An Integrative Framework for Model-Driven Systems Engineering : Towards the Co-Evolution of Simulation, Formal Analysis and Enactment Methodologies for Discrete Event Systems

Hamzat Olanrewaju Aliyu

N° d'ordre: D. U : 2777
E D S P I C : 786

UNIVERSITÉ BLAISE PASCAL - CLERMONT-FERRAND II
ÉCOLE DOCTORALE DES SCIENCES POUR L'INGÉNIEUR DE CLERMONT-FERRAND

PhD Thesis

Submitted by:

HAMZAT OLANREWAJU ALIYU

To obtain the degree of:

PhD in Computer Science

Title of the thesis:

An Integrative Framework for Model-Driven Systems Engineering: Towards the Co-Evolution of Simulation, Formal Analysis and Enactment Methodologies for Discrete Event Systems

Publicly defended on December 15, 2016 before the jury:

Jean-Pierre Müller	Referee
Grégory Zacharewicz	Referee
Mamadou Kaba Traoré	Thesis Supervisor
David Hill	Examiner and President of the Jury
Henri Pierreval	Examiner
Andreas Tolk	Examiner

N° d'ordre : D. U : 2777
E D S P I C : 786

UNIVERSITÉ BLAISE PASCAL - CLERMONT-FERRAND II

ÉCOLE DOCTORALE DES SCIENCES POUR L'INGÉNIEUR DE CLERMONT-FERRAND

Thèse

Présentée par

HAMZAT OLANREWAJU ALIYU

Pour obtenir le grade de:

DOCTEUR D'UNIVERSITÉ

SPÉCIALITÉ: Informatique

Titre de la thèse :

Un cadre d'intégration pour l'ingénierie dirigée par modèle des systèmes - Vers la coévolution de la simulation, de l'analyse formelle et de l'émulation des systèmes à événements discrets

Soutenue publiquement le 15 Décembre, 2016 devant le jury:

Jean-Pierre Müller	Rapporteur
Grégory Zacharewicz	Rapporteur
Mamadou Kaba Traoré	Directeur de thèse
David Hill	Examinateur et Président du Jury
Henri Pierreval	Examinateur
Andreas Tolk	Examinateur

Abstract

Model-based systems engineering methodologies such as Simulation, Formal Methods (FM) and Enactment have been used extensively in recent decades to study, analyze, and forecast the properties and behaviors of complex systems. The results of these analyses often reveal subtle knowledge that could enhance deeper understanding of an existing system or provide timely feedbacks into a design process to avert costly (and catastrophic) errors that may arise in the system. Questions about different aspects of a system are usually best answered using some specific analysis methodologies; for instance, system's performance and behavior in some specified experimental frames can be efficiently studied using appropriate simulation methodologies. Similarly, verification of properties such as, liveness, safeness and fairness are better studied with appropriate formal methods while enactment methodologies may be used to verify assumptions about some time-based and human-in-the-loop activities and behaviors. Therefore, an exhaustive study of a complex (or even seemingly simple) system often requires the use of different analysis methodologies to produce complementary answers to likely questions. There is no gainsaying that a combination of multiple analysis methodologies offers more powerful capabilities and rigor to test system designs than can be accomplished with any of the methodologies applied alone. While this exercise will provide (near) complete knowledge of complex systems and helps analysts to make reliable assumptions and forecasts about their properties, its practical adoption is not commensurate with the theoretical advancements, and evolving formalisms and algorithms, resulting from decades of research by practitioners of the different methodologies. This shortfall has been linked to the prerequisite mathematical skills for dealing with most formalisms, which is compounded by little portability of models between tools of different methodologies that makes it mostly necessary to bear the herculean task of creating and managing several models of same system in different formalisms. Another contributing factor is that most of existing computational analysis environments are dedicated to specific analysis methodologies (i.e., simulation or FM or enactment) and are usually difficult to extend to accommodate other approaches. Thus, one must learn all the formalisms underlining the various methods to create models and go round to update all of them whenever certain system variables change. The contribution of this thesis to alleviating the burdens on users of multiple analysis methodologies for exhaustive study of complex systems can be described as a framework that uses Model-Driven Engineering (MDE) technologies to federate simulation, FM and enactment analysis methodologies behind a unified high-level specification language with support for automated synthesis of artifacts required by the disparate methodologies. This framework envelops four pieces of contributions: i) a methodology that emulates the Model-Driven Architecture (MDA) to propose an independent formalism to integrate the different analysis methodologies. ii) Integration of concepts from the three methodologies to provide a common metamodel to unite some selected formalisms for simulation, FM and enactment. iii) Mapping rules for automated synthesis of artifacts for simulation, FM and enactment from a common reference model of a system and its requirements. iv) A framework for the enactment of

discrete event systems. We use the beverage vending system as a running example throughout the thesis.

A methodology that emulates the Model-Driven Architecture (MDA) to propose an independent formalism to integrate the different analysis methodologies: The application of MDA approach to software development has recorded significant advantages such as reduced coding time and efforts, automated synthesis of error-free code and increased throughput. To replicate these advantages in the process of modeling complex systems for complementary analyses with simulation, FM and enactment, we propose a framework that mirrors the MDA by generating the models for the different analysis methodologies from a reference model, which is independent of any of the methodologies. With this approach, the modeler needs only to be concerned with the correct specification of the reference model while the artifacts required for each of the analysis methodologies can be (re)generated with little efforts. By this approach, the domain expert is shielded, largely, from the efforts, time and mathematical rigor required to specify (and update) models for each of the different methodologies.

Integration of concepts expressed in selected simulation, FM and enactment formalisms to provide an independent meta-model for the reference models: Language engineering is one of the cornerstones of MDE. Researchers in this field have proposed techniques to integrate different languages to form new languages in some suitable circumstances; of particular interest in this context is the integration of different meta-models to produce a new coherent one. In this thesis, we use some of these techniques to integrate concepts from some considerably universal formalisms in system theory and software engineering to define a unified language, which is expressive enough to be at the kernel of an MDA-mirrored framework that support the automated generation of artifacts for three analysis methodologies: simulation, FM, and enactment.

Mapping rules for automated synthesis of artifacts for simulation, FM and enactment from a common reference model of a system and its requirements: Using MDE techniques, particularly model transformation techniques, to define separate mappings of the concepts described in the aforementioned unified language to the underlying formalisms of selected for simulation, FM and enactment methodologies. i.e., the semantics of the constructs of the unified language are given by the selected formalisms. Therefore, we take benefit of model transformation techniques to map the common meta-model to each of the formalism to achieve the automated synthesis of the artifacts required for each of the disparate methodologies.

A preliminary framework for the enactment of discrete event systems: Unlike simulation and formal analysis, both of which have well established formalisms and operational/logical protocols that are accepted by considerably large communities, enactment methodology has yet to permeate significantly into the systems engineering practice with discrete event systems in general. The current practices of enactment are mostly based on UML (Unified Modeling Language), SysML (System Modeling Language) and their profiles. We propose an enactment

framework for discrete event systems, which adopts truly system-theoretic concepts to express a system model and the object-oriented observer design pattern to define its enactment semantics.

Keywords: Model-Driven Systems Engineering, SimStudio, HiLLS, Simulation, Formal Analysis, Enactment, DEVS, Z, Temporal Logic.

List of publications

The following are the journal and conference papers produced in the course this thesis:

1. Aliyu, H. O., Maïga, O., Traoré, M. K. (2016). The high level language for system specification: A model-driven approach to systems engineering. *International Journal of Modeling, Simulation, and Scientific Computing*, *7*(01), 1641003.
2. Djitog, I., Aliyu, H. O., Traoré, M. K. (2017). Multi-Perspective Modeling of Healthcare Systems. *International Journal of Privacy and Health Information Management*, ***in press***.
3. Aliyu, H. O., Traoré, M. K. (2016). Integrated Framework for Model-Driven Systems Engineering: A Research Roadmap. In *Proceedings of the 2016 Spring Simulation Multi-Conferences* - SpringSim'16(p. 28), April 3-6, 2016, Pasadena, CA, USA, SCS International.
4. Aliyu, H. O., Traoré, M. K. (2015). Toward an Integrated Framework for the Simulation, Formal Analysis and Enactment of Discrete Event Systems Models. In *Proceedings of the 2015 Winter Simulation Conference*- WSC'15 (pp. 3090-3091), December 6-9, 2015, Huntington Beach, CA, USA. IEEE Press.
5. Aliyu, H. O., Traoré, M. K. (2015). Towards a unified framework for holistic study and analysis of discrete events systems. In *Proceedings of The AUST International Conference on Technology*-AUSTECH'15*,* October 12-13, 2015, Abuja, Nigeria
6. Aliyu, H. O., Maïga, O., Traoré, M. K. (2015). A framework for discrete events systems enactment. In Proceedings of 29th *European Simulation and Modeling Conference* - ESM'15 (pp. 149-156)*,* October 26-28, 2015, Leicester, UK, EUROSIS-ETI.
7. Maïga, O., Aliyu, H. O., Traoré, M. K. (2015). A new approach to modeling dynamic structure systems. In Proceedings of 29th *European Simulation and Modeling Conference* - ESM'15 (pp. 141-148)*,* October 26-28, 2015, Leicester, UK, EUROSIS-ETI.
8. Djitog, I., Aliyu, H. O., Traoré, M. K. (2015). Towards a framework for holistic analysis of healthcare systems. In Proceedings of 29th *European Simulation and Modelling Conference - ESM'2015,* October 26-28, 2015, Holiday Inn, Leicester, United Kingdom.

Résumé

Les méthodes d'ingénierie dirigée par modèle des systèmes, telles que la simulation, l'analyse formelle et l'émulation ont été intensivement utilisées ces dernières années pour étudier et prévoir les propriétés et les comportements des systèmes complexes. Les résultats de ces analyses révèlent des connaissances qui peuvent améliorer la compréhension d'un système existant ou soutenir un processus de conception de manière à éviter des erreurs couteuses (et catastrophiques) qui pourraient se produire dans le système. Les réponses à certaines questions que l'on se pose sur un système sont généralement obtenues en utilisant des méthodes d'analyse spécifiques ; par exemple les performances et les comportements d'un système peuvent être étudiés de façon efficace dans certains cadres expérimentaux, en utilisant une méthode appropriée de simulation. De façon similaire, la vérification de propriétés telles que la vivacité, la sécurité et l'équité sont mieux étudiées en utilisant des méthodes formelles appropriées tandis que les méthodologies d'émulation peuvent être utilisées pour vérifier des hypothèses temporelles et des activités et comportements impliquant des interactions humaines. Donc, une étude exhaustive d'un système complexe (ou même d'apparence simple) nécessite souvent l'utilisation de plusieurs méthodes d'analyse pour produire des réponses complémentaires aux probables questions. Nul doute que la combinaison de multiples méthodes d'analyse offre plus de possibilités et de rigueur pour analyser un système que ne peut le faire chacune des méthodes prise individuellement. Si cet exercice (de combinaison) permet d'aller vers une connaissance (presque) complète des systèmes complexes, son adoption pratique ne va pas de pair avec les avancées théoriques en matière de formalismes et d'algorithmes évolués, qui résultent de décennies de recherche par les praticiens des différentes méthodes. Ce déficit peut s'expliquer parles compétences mathématiques requises pour utiliser ces formalismes, en combinaison avec la faible portabilité des modèles entre les outils des différentes méthodes. Cette dernière exigence rend nécessaire la tâche herculéenne de créer et de gérer plusieurs modèles du même système dans différents formalismes et pour différents types d'analyse. Un autre facteur bloquant est que la plupart des environnements d'analyse sont dédiés à une méthode d'analyse spécifique (i.e., simulation, ou analyse formelle, ou émulation) et sont généralement difficiles à étendre pour réaliser d'autres types d'analyse. Ainsi, une vaste connaissance de formalismes supportant la multitude de méthodes d'analyse est requise, pour pouvoir créer les différents modèles nécessaires, mais surtout un problème de cohérence se pose lorsqu'il faudra mettre à jour séparément ces modèles lorsque certaines parties du système changent. La contribution de cette thèse est d'alléger les charges d'un utilisateur de méthodes d'analyse multiples, dans sa quête d'exhaustivité dans l'étude des systèmes complexes, grâce à un cadre qui utilise les technologies d'Ingénierie Dirigée par les Modèles (IDM) pour fédérer la simulation, l'analyse formelle et l'émulation. Ceci est rendu possible grâce à la définition d'un langage de spécification unifié de haut niveau, supporté par des capacités de synthèse automatiques d'artéfacts requis par les différentes méthodes d'analyse. En fait, ce travail de thèse propose quatre contributions majeures, qui sont : i) un cadre opérationnel qui utilise l'Architecture Dirigée par les Modèles

(ADM) pour supporter ce formalisme de haut niveau, ii) l’intégration de concepts d’horizons multiples pour créer un métamodèle commun qui unifie la simulation, l’analyse formelle et l’émulation, et qui définit la syntaxe abstraite de ce formalisme, iii) des règles de transformation pour la synthèse automatique d’artéfacts à destination de la simulation, de l’analyse formelle et de l’émulation, à partir de tout modèle de système accompagné d’un modèle de ses exigences, et iv) un domaine sémantique pour l’émulation des systèmes à évènements discrets. Tout au long de la thèse, un cas d’école de distribution automatique de boisson est utilisé comme exemple.

Un cadre opérationnel qui utilise l'Architecture Dirigée par les Modèles (ADM):l'application de l’ADM pour le développement de logiciels a enregistré des avantages importants tels que la réduction du temps et des efforts de codage, la synthèse automatisée de code sans erreur et l'augmentation des performances. Pour reproduire ces avantages dans le processus de modélisation multi-analyse des systèmes complexes, nous proposons un cadre qui reflète l’architecture ADM en générant des modèles pour les différentes méthodes d'analyse à partir d'un modèle de référence, qui est indépendant de ces méthodes. Avec cette approche, le modélisateur doit seulement se préoccuper de la spécification correcte du modèle de référence, tandis que les artefacts nécessaires pour chacune des méthodes d'analyse peuvent être générés avec peu d'efforts. Par cette approche, l'expert du domaine est déchargé, en grande partie, de l'effort, du temps et de la rigueur mathématique requise pour spécifier (et mettre à jour) des modèles pour chacune des méthodes.

L'intégration des concepts d’horizons multiples pour créer un méta-modèle unificateur: l’ingénierie des langages est l'une des pierres angulaires de l’IDM. Les chercheurs dans ce domaine ont proposé des techniques pour intégrer différents langages en un nouveau langage, ce sous certaines circonstances appropriées; l'intégration de différentes méta-modèles pour produire un nouveau méta-modèle cohérent est d'un intérêt particulier dans ce contexte. Dans cette thèse, nous utilisons certaines de ces techniques pour intégrer des concepts issus de la théorie des systèmes et du génie logiciel. Le langage unifié résultant est assez expressif pour servir de noyau à notre cadre opérationnel, et permettre la génération automatique des artefacts destinés respectivement à la simulation, à l’analyse formelle, et à l’émulation.

Des règles de transformation pour la synthèse automatique d’artéfacts: en utilisant les techniques de l’IDM, particulièrement les techniques de transformation de modèles, nous définissons les sémantiques du langage unifié construit, en projetant son méta-modèle dans des domaines sémantiques correspondant à des formalismes adéquats respectivement pour la simulation, l’analyse formelle et l’émulation. Ce faisant, nous rendons la synthèse automatique des artéfacts requis possibles.

Un domaine sémantique pour l’émulation des systèmes à évènements discrets : contrairement à la simulation et à l’analyse formelle qui bénéficient, toutes les deux, de formalismes bien définis (syntaxe et sémantique) et bien acceptés par de large communautés scientifiques, l’émulation n’a pas un grand niveau de maturité dans l’ingénierie des systèmes à évènements discrets. Les pratiques courantes d’émulation sont basées sur UML (Unified

Modeling Language), SysML (System Modeling Language) et leurs profils. Nous proposons un cadre sémantique d'émulation pour les systèmes à évènements discrets, qui adopte vraiment les concepts de la théorie des systèmes pour décrire un système, et qui fait usage de patrons de conception orientée-objet.

Mots clés: Ingénierie Dirigée par les Modèles, SimStudio, HiLLS, Simulation, Analyse Formelle, Emulation, DEVS, Z, Logique Temporelle.

Liste de publications

Les articles suivants sont des contributions faites pendant cette thèse :

1. Aliyu, H. O., Maïga, O., Traoré, M. K. (2016). The high level language for system specification: A model-driven approach to systems engineering. *International Journal of Modeling, Simulation, and Scientific Computing*, *7*(01), 1641003.

2. Djitog, I., Aliyu, H. O., Traoré, M. K. (2017). Multi-Perspective Modeling of Healthcare Systems. *International Journal of Privacy and Health Information Management*, ***in press***.

3. Aliyu, H. O., Traoré, M. K. (2016). Integrated Framework for Model-Driven Systems Engineering: A Research Roadmap. In *Proceedings of the 2016 Spring Simulation Multi-Conferences* - SpringSim'16 (p. 28), April 3-6, 2016, Pasadena, CA, USA, SCS International.

4. Aliyu, H. O., Traoré, M. K. (2015). Toward an Integrated Framework for the Simulation, Formal Analysis and Enactment of Discrete Event Systems Models. In *Proceedings of the 2015 Winter Simulation Conference*- WSC'15 (pp. 3090-3091), December 6-9, 2015, Huntington Beach, CA, USA. IEEE Press.

5. Aliyu, H. O., Traoré, M. K. (2015). Towards a unified framework for holistic study and analysis of discrete events systems. In *Proceedings of The AUST International Conference on Technology*-AUSTECH'15*,* October 12-13, 2015, Abuja, Nigeria

6. Aliyu, H. O., Maïga, O., Traoré, M. K. (2015). A framework for discrete events systems enactment. In Proceedings of 29th *European Simulation and Modeling Conference* - ESM'15 (pp. 149-156)*,* October 26-28, 2015, Leicester, UK, EUROSIS-ETI.

7. Maïga, O., Aliyu, H. O., Traoré, M. K. (2015). A new approach to modeling dynamic structure systems. In Proceedings of 29th *European Simulation and Modeling Conference* - ESM'15 (pp. 141-148)*,* October 26-28, 2015, Leicester, UK, EUROSIS-ETI.

8. Djitog, I., Aliyu, H. O., Traoré, M. K. (2015). Towards a framework for holistic analysis of healthcare systems. In Proceedings of 29th *European Simulation and Modelling Conference - ESM'2015,* October 26-28, 2015, Holiday Inn, Leicester, United Kingdom.

Acknowledgments/Remerciements

All praises are due to Allah the Almighty, the Creator and Sustainer of the Worlds, for having been seeing me through in all my endeavors, and especially for giving me the privilege to attain this new feat in my academic pursuits.

I am greatly indebted to many people, who have helped me in different capacities during my journey to obtaining the PhD degree. To begin with, I am profoundly grateful to my supervisor, Professor Mamadou Kaba Traoré of Université Blaise Pascal, who in addition to his excellent guidance, his care and understanding were sometimes all that helped to maintain my focus during the work on this thesis.

I would like to thank all members of my thesis committee - Professors Jean-Pierre Müller of Université de Montpellier, Grégory Zacharewich of Université de Bordeaux, David Hill of Université Blaise Pascal, Henri Pierreval of Université Blaise Pascal, and Andreas Tolk of MITRE Corporation, USA - for accepting to be in my committee, and for their inspiring comments during my defense. Thanks, also, to Prof. Hans Vangheluwe of the University of Antwerp, Belgium for his insightful questions, during our discussions at SprinSim'16, which led to the provision of support for high-level specification of required properties in HiLLS.

I am grateful to all members of our GReP research group for their contributions to the success of my work. Many thanks to Oumar Maïga for the wonderful moments we shared during our several discussions on the specification of HiLLS, and for helping with the French version of the abstract of this thesis. Thanks to Ignace Djitog, Shaowei Wang, Youssouf Konné, Doyin Adegoke, Yoro Diouf, Kehinde Eli-Ake, Hajarah AbdulWahab, Hamidou Togo, Moussa Koita, Hawa Bado, Alpha Bazemo, Yan Wang and Yanhong Wang for all their comments and insightful questions during our weekly GReP sessions.

The love and invaluable supports received from members of my family have always been significant in my endeavors. I am especially grateful to my mother and sister, Sideeqoh, for all their sacrifices, right from the onset, to see to my academic pursuits. Thank you also Mama for enduring my absence during the period of this work; may Allah grant you a healthy long life to enjoy the good fruits of your labor. I am also very grateful to my wife, Ma'rufah, and son, Abdur-Rahman for sharing my attention with the thesis while boosting my morale and praying passionately for my success.

I am profoundly grateful to the friends and colleagues, whose unalloyed supports and magnanimities have helped to keep many things in place behind my back. Many thanks to Mustapha Magaji, Razaq Lasisi, Luqman AbdulAzeez, Qasim Adeoye, Murtadha Ibrahim, Mr. Shefiu Ganiyu and Dr Qasim Salako. Jazakumul-Laahu khaeran.

This work would not have been possible without the PhD scholarship offered me by the Nigerian National Information Technology Development Agency (NITDA) through which the work was fully sponsored. I am profoundly grateful to the management and staff of NITDA for always

responding to my financial requests throughout the period of working on this thesis. I appreciate the courage and kindness of Dr Kunle O. Babalola of the University of Ilorin for accepting to stand as my guarantor to NITDA to pave way for the release of the grant.

Hamzat Olanrewaju ALIYU

December 2016.

Contact: alilanre@yahoo.com

To my Mother

Table of Contents

List of Figures

List of Tables

List of Acronyms

ARMI	Asynchronous Remote Method Invocation
ATL	ATLAS Transformation Language
AToMPM	A Tool for Multi-Paradigm Modeling
BPM	Business Process Management
CIL	Computation-Independent Layer
CIM	Computation-Independent Model
CM	Conceptual Model
CTL	Computation Temporal Logic
CZT	Community Z Tools
DDML	DEVS-Driven Modeling Language
DES	Discrete Event System
DEVS	Discrete Event System Specification
DEVSDSOL	DEVS Distributed Simulation Object Library
DEVSML	DEVS Modeling Language
DEVS-MS	DEVS-Meta Simulator
DML	DEVS Markup Language
DSL	Domain-Specific Languages
DSM	Domain-Specific Model
DSML	Domain-Specific Model Language
DUNIP	DEVS Unified Process
EF	Experimental Frame
EMF	Eclipse Modeling Framework
EMP	Eclipse Modeling Project

FM	Formal Methods
GME	Generic Modeling Environment
GUI	Graphical User Interface
HiLLS	High Level Language for System Specification
INCOSE	International Council on Systems Engineering
ISO	International Standards Organization
LTL	Linear Temporal Logic
M&S	Modeling and Simulation
M2M	Model-to-Model Transformation
M2T	Model-to-Text Transformation
MBSE	Model-Based Systems Engineering
MDA	Model-Driven Architecture
MDD4MS	Model-Driven Development for Modeling and Simulation
MDE	Model-Driven Engineering
MDSE	Model-Driven Systems Engineering
MIC	Model-Integrated Computing
MIM	Methodology-Independent Model
MOF	Meta-Object Facility
MSM	Methodology-Specific Model
MTL	Model Transformation Language
MVC	Model-View-Controller
OMG	Object Management Group
PDM	Platform Description Model
PIL	Platform-Independent Layer
PIM	Platform-Independent Model

PISM	Platform-Independent Simulation Model
ProMoBox	Properties and (design) Models developed (Boxed) in concert
PSL	Platform-Specific Layer
PSM	Platform-Specific Model
PSSM	Platform-Specific Simulation Model
SUS	System under study
SysML	System Modeling Language
TL	Temporal Logic
TS	Technological Space
UML	Unified Modeling Language
XMI	XML Metadata Interchange
XML	eXtensible Markup Language
ZML	Z Markup Language

1 GENERAL INTRODUCTION

1.1 MODEL-DRIVEN SYSTEMS ENGINEERING

Model-Based Systems Engineering (MBSE) is defined by the International Council on Systems Engineering (INCOSE) as "*the formalized application of modeling to support system requirements, design, analysis, verification, and validation activities beginning in the conceptual design phase and continuing throughout development and later life cycle*"[Est07]. It was primarily envisioned to address the limitations of the traditional Document-Based Systems Engineering process that relies on text documents and engineering data in many formats to drive the design, implementation and maintenance of complex systems. Mittal and Martín [MM13] first defined Model-Driven Systems Engineering (MDSE) as "*a discipline that applies Model-Driven Engineering (MDE) practices to MBSE paradigm*" to automate MBSE processes. According to Bocciarelli and D’Ambrogio [BD14], MDSE stimulates a radical shift from a merely contemplative use of models - in MBSE - to a productive and more effective use by the application of meta-modeling techniques and automated model transformations to the systems engineering domain, thus boosting the advantages of the MBSE approach.

An efficient design and development of a complex system requires an iterative process of modeling, performance evaluation, logical analysis for requirement verifications and implementation (prototype) for run-time testing [HK06]. This iterative process is necessary to reveal subtle knowledge about the systems, which are, in most cases, beyond intuition. Moreover, a violation of requirement(s) or undesired behavior at this stage can be a signal of a fundamental flaw in the design that must be resolved early to forestall costly errors in the final system [ZN16]. Hence, the knowledge gained from each of the iterations may serve two purposes: 1) as a guide for deeper understanding of the system's behavior and the influences it may have on its environment or which the environment may have on it. 2) as a feedback to revise the designed model and/or requirements until an acceptable level of satisfaction of critical requirements is guaranteed before committing time and resources to the implementation of the system.

Figure 1.1is a schematic illustration of a typical MDSE methodology. The system under study (SUS) is represented as a model, possibly consisting of interacting components, using an appropriate formalism. SUS may be a physical system or a non-existent conceptualized system. Solver refers to the protocols and algorithms for analyzing the system model to generate some results. The choice of an appropriate formalism to write the system model is determined by a number of factors such as the properties of SUS to be analyzed, the capabilities of solver available, the analyst's experience, etc. The solver does the manipulation of the system model and the results obtained provide some feedback as more knowledge about SUS to the analyst.

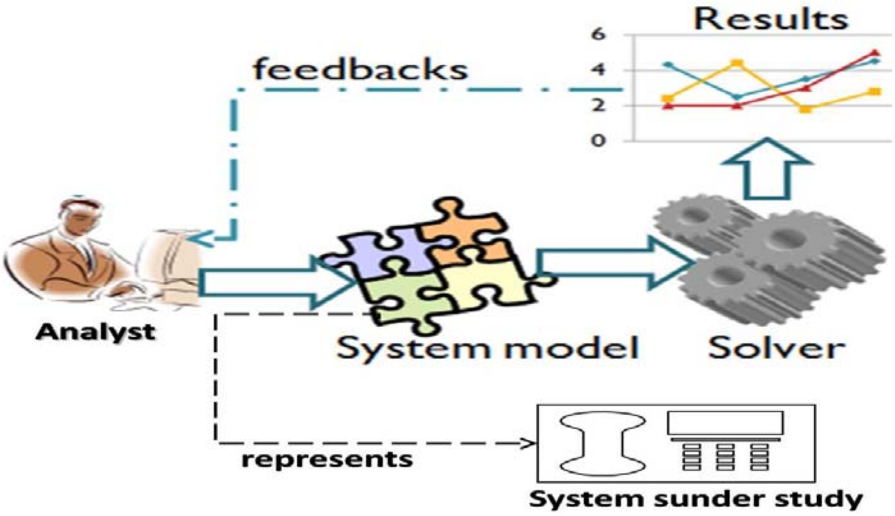


Figure 1.1 MDSE methodology
The information represented in the model and the chosen modeling formalism and solver all depend on the questions to be answered about the system under study

Depending on the questions to be answered about SUS, MDSE approaches based on theoretically sound methodologies like simulation, Formal Methods or enactment are employed in the iteration loops to mine the desired knowledge of the SUS from the model.

This thesis explores the integration of MDSE theories and technologies along three dimensions of design analysis and verification methodologies: simulation, formal methods and enactment. The goal is to harness the synergy of diverse tools and experiences for an encompassing investigation of properties in the design space of complex Discrete Event Systems (DES). The three methodologies promote reasoning about systems from somewhat divergent viewpoints; they are, however, often necessarily required in combinations for sound system designs. A viewpoint can be defined, in the general context of software and systems engineering, as a description of appropriate machinery consisting of domain, languages, specifications and methodologies to capture and process one or more related engineering or technical concerns about a system and the information associated with such concern(s) [FKN+92, KW07]. The overall objective of the thesis is to put them together under the umbrella of a unified high-level viewpoint to make them accessible to non-experienced users as well as ease the tasks of experienced users.

A typical simulation methodology allows the practitioner to *compress time* (i.e., use logical time approximation) and evaluate or analyze a model over a specified period and under scenarios or environment defined by the experimental frame. In Modeling and Simulation (M&S), an

Experimental Frame (EF) defines the objective(s), assumptions and constraints of a simulation study and the context within which a system is observed or the validity of the model is evaluated [ZPK00, TM06]. The results obtained from a simulation study may be used to infer or forecast system's behavior and performance, identify problems and their causes, etc. Comprehensive lists of problems that are suitable for simulation as well as possible uses of simulation results are provided in [Mar97, Car04].

Formal Methods (FM) are mathematically-based languages, techniques and tools that permit the specification, verification and development of software and hardware systems in a systematic manner [Win90, CW+96]. The goal of FM is to unveil and correct subtle and very expensive errors that may result in occasional mysterious system failures [Lam77, CW+96]. The likelihood of having such inconspicuous errors in a system design increases as the system grows in scale and functionality; hence the need for a systematic use of FM at some strategic phases of a systems development process to enhance the developers' understanding of the system through early revelations of ambiguities, inconsistencies and violations of specification requirements. The overall motivation for this will be the eventual construction of complex systems that operate reliably [CW+96]. Similar to the way an EF defines the objectives and context of a simulation study of a system, the *requirement* proposes the premise on which the logical and symbolic reasoning about a system is based using FM. In essence, the requirement is an abstract specification of a collection of properties (or evaluation criteria) that need to be satisfied by a system under study [Lam00]. It serves as a contract between the client and system developers [Win90]; and logical analysis of the combined operational model of the system and requirement helps to verify that the former satisfies the latter. Counterexamples are generated (in some cases) to illustrate the violations where requirements are not met. An empirical survey of the whys and wherefores of using FM is presented in [Hal05, Hal07].

Enactment is a system analysis (cum implementation) methodology often used in Business Process Management (BPM) [VTW03, JN14] for the execution of a business workflow [KGJ10] - the (semi-) automation of business processes during which information and work lists are passed from one participant to another for necessary actions [OF07]. In service engineering and Human-Computer Interaction, enactment refers to the manifestation of the functionalities represented by a system's prototype [HE07]. A system prototype is described as a representation of the functionality, but not the appearance, of a finished artifact which can be used as a proof that a certain theory or concept or technology works or otherwise [Hol05]. In a more general software engineering context, enactment is described as the mechanism for the execution or interpretation of software process definitions, which may involve live interactions with the environment and external actors like human-in-the-loop, to provide supports that are consistent with the process definitions [DF94]. In the context of DES, we describe enactment as the mechanism for executing an operational model (i.e., one that can be executed in a suitable software environment [BA95]) of a system to act out its behavior by using the physical clock time as the reference for the scheduling and execution of events. An enactment model should practically stand in for the real system in a physical environment through the manifestation of its expected characteristics. Enactment is different from mainstream DES simulation in three ways: 1) scheduling and execution of events in the former should be based on physical time as against logical (compressed or approximated) time in the latter; 2) the former should allow for the observable manifestation of state activities (operations that do not lead to state changes) as against instantaneous operations in the latter; and 3) the former should allow for live (runtime) interaction between operational model and its physical environment (e.g., human- or hardware- or software-in-the-loop) as against the interaction with virtual environment defined by the EF in the latter. Results obtained from an enactment process can give further insights into the system's behavior for verification of timing correctness in real-time systems as well as point out certain inconsistencies, missing requirements. A preliminary discussion of enactment as applied to DES can be found in [AMT15].

It is important to point out here that the scope of the work presented in this thesis is within the confines of Discrete Event Systems (DES). A DES is a dynamic system whose state evolves with the instantaneous occurrences, at possibly unknown irregular intervals, of events over time [RW89, CL09]. An event is an instantaneous occurrence that results into a change of state of the system. For instance, in an elevator, the receipt of a floor request and arrival at a floor are both events. While the former event may lead to a state in which the system is moving up or down with its door closed, the latter leads to a state in which the system is stationary with its door opened. Thus, the reader can safely take term "system" to mean DES henceforth except where it is stated otherwise.

1.2 PROBLEM STATEMENTS

A salient point to be noted from the previous section is that no single MDSE methodology is sufficient to study all aspects of a complex system since each one is most suitable to provide answers to some specific kinds of questions. Thus, an exhaustive study of a complex system requires a synergy of different analysis methodologies; this synergy can be achieved through a disciplined combination of the methodologies so that they provide complimentary, rather than competitive, answers to evolving design questions. Recent publications such as [TH14, BD14, ZN16, Shu11, Tra08] suggest that there are growing interests, both in academia and industry, in the systematic combinations of disparate MDSE approaches to maximize the synergy between the different disciplines.

The practical adoption of this collaborative approach to computational analysis of systems is, however, being constrained directly or indirectly by the same forces inhibiting the wide adoption of individual analysis methodologies especially by non-expert users. The gap between research outputs in terms of theoretical advancements cum evolving formalisms and their practical adoptions by domain experts is widely acknowledged by practitioners of both simulation and formal methods methodologies. A number of reasons have been identified, in the literature, for this shortfall; chief among them is the lack of requisite logic and mathematical skills to deal with most formalisms. More details on this so-called chief constraint and two more challenges, which together form the premises on which the research questions of this thesis are based, are provided below:

P1. ***Lack of requisite logic and mathematical skills to deal with most formalisms:*** Computational analysis methodologies often rely on some mathematics- and/or logic-based formalisms for the specification and manipulation of systems and their frames (experimental frames or requirements). This is necessary to ensure formal reasoning with models with precise semantics and devoid of ambiguities and inconsistencies. Domain users, however, seldom have the requisite skills to deal with such formalisms; they are considered, somewhat, to be some kinds of low level expressions that do not match with the high level artifacts which domain users are often accustomed to. This problem has been continuously acknowledged and addressed through the development of high-level modeling interfaces on top of the formalisms to make them accessible to non-expert users through automated synthesis of low-level artifacts from the high-level models by courtesy of MDE techniques. Surveys of some of such interfaces for discrete event simulation [FBT+14] and FM [KES03] highlight their features. It is important to note, however, that the tools vary in their capabilities to express different aspects of complex systems; hence accessibility to non-expert users is still open to further research at the time of writing this thesis.

P2. ***Little chances of portability of models between computational analysis methodologies:*** Another source of concern identified in the course of this thesis [AT15a, AT15b, AT16], and which directly constrains the study of a system using multiple analysis methodologies, is that there are usually little chances of portability of models between methodologies. This usually requires manual, or at best semi-automated, creation and updating of separate models, in different formalisms, of yet the same system for different kind of analysis. This task can be herculean and error-prone.

P3. ***Little coexistence of disparate methodologies in the same environment:*** Arguably as a consequence of the limited chances of sharing system models among disparate analysis methodologies, different techniques rarely coexist in the same environment; rather most existing computational analysis environments dedicated to specific analysis methodologies and they are usually difficult to extend to accommodate other approaches. In addition to laborious task of managing and maintaining consistencies between the different models, having multiple disconnected views of the same system model in separate MDSE environments has the potential to create miscommunication among the development teams [BSD+12]. The importance of this problem can be attributed to, among others, the prospect of enhancing the portability of system models among analysis methodologies through their co-existence and co-evolution in the same environments.

Though recent publications (e.g., [TH14, BD14, Shu11, ZN16]) suggest that these problems, especially **P2** and **P3**, are growing in importance, and getting more research attentions, the breakthroughs have yet to manifest significantly; hence there is need for concerted research efforts in this direction to consolidate the findings so far and deploy same to enhance the industrial adoption of the approaches.

1.3 RESEARCH QUESTIONS AND MOTIVATIONS

Sequel to the problem statements in Section1.2 above and the extent to which they have been addressed by the current practices of MDSE, we formulate the main research question of this thesis as follows:

RQ1. Is it possible to build an integrative framework that can be continuously populated with best practices in MDSE for simulation, formal methods and enactment such that the various components are federated through a seamless sharing of high-level system model?

The idea is not to reinvent all the MDSE components; rather it is to federate legacy simulation, formal analysis and enactment tools in a unified framework in such a way that one high-level system model (that is understood by all stakeholders) can be technically used to drive the MDSE processes in all the three dimensions.

The motivations for this approach would include:

- The use of a shared model will reduce the task of creating and updating models for the different analysis purposes.
- Using a unified language to write the shared model can enhance communication among the stakeholders in the development of a system.
- Possibility to crosscheck that a property verified with one of the analysis methodologies actually continues to hold in analysis with other methodologies since they are all based on the same base model.
- Possibility to check the effects (if any) of model changes initiated by one of the three methodologies on the properties being verified using other methodologies. For instance, if a simulation practitioner modifies some system variables in the shared model in order to satisfy some properties, the formal analysis practitioner can immediately check

whether or not the new changes in the model lead to the violation of already proven requirements and vice versa.

- There is likelihood of discovering some useful research directions towards enhancing the co-evolution of the different MDSE methodologies.

These motivations could lead us to subject the research question itself to more queries like:

RQ2. Which formalism should we adopt to write the shared unified model?

RQ3. How can the disparate concerns of the different methodologies be captured in the so-called unified model

RQ4. In what order should the process of the different methodologies be executed?

This thesis proposes answers to these research questions in subsequent chapters; the next section presents an overview of the answers as a summary of the thesis' contribution while detailed discussions will be provided in later chapters.

1.4 THESIS CONTRIBUTIONS

We believe that the abstract global answer to RQ1 above is in the affirmative; of course, the creation of liaisons between the disparate approaches will require intensive efforts since they are independent bodies of knowledge developed by different research communities and, largely, for different purposes. Interestingly, MDE is an open and integrative approach that emphasizes building bridges between technological spaces [KBA02, FN05]. This statement gives impetus to the overall contribution of this thesis to answering the research questions since the technological spaces are themselves bodies of knowledge originating from different research communities. Hence, we can take advantage of the openness of the MDE approach to explore the possibility of replacing technological spaces in MDE with the simulation, formal analysis and enactment methodologies towards developing an integrative framework to harness the synergy between the disparate analysis methodologies for an exhaustive analysis of DES designs.

Con 1. A multi-layered framework that emulates the Model-Driven Architecture (MDA) by defining a unified model specification layer on top of the layers containing the specific analysis methodologies: In answering RQ1, we propose a methodological framework with a multi-level architecture similar to the MDA as described in Figure 1.2. Unlike the conventional MDA that primarily targets software development; the framework depicted in Figure 1.2has MDSE components at the different layers. The topmost layer of the framework supports the high-level specification of unified system and requirement models which are decoupled from the three MDSE methodologies but expressive enough to drive each of them. The modeling layer sits on top of a layer containing the underlying formalisms for the three methodologies so that a model in the former can be used to drive the automated synthesis of models required in the latter.

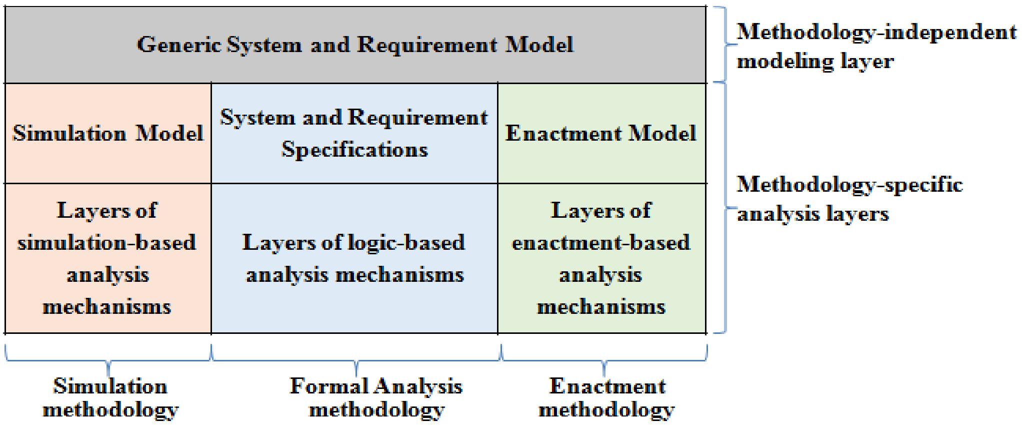

Generic System and Requirement Model
Methodology-independent modeling layer
Simulation Model
System and Requirement Specifications
Enactment Model
Methodology-specific analysis layers
Layers of simulation-based analysis mechanisms
Layers of logic-based analysis mechanisms
Layers of enactment-based analysis mechanisms
Simulation methodology
Formal Analysis methodology
Enactment methodology

Figure 1.2 An MDA-like integrative MDSE framework

This approach provides a global framework to address the three problems identified previously in Section 1.2. It consolidates on existing solutions to problem P1through a high level modeling interface that shields the user, to a large extent, from the requisite mathematical skills to deal with the underlying formalisms of the different analysis methodologies, especially simulation and FM. It also addresses problems P2 and P3through the automated generation of artifacts for the three methodologies from the same unified model thereby fostering their co-evolution through sharing of models; in this way, the task of creating and updating models for the three methodologies will be greatly reduced since the users have to deal manually with only one model - the unified model. Lastly, the idea is not to override existing solutions for the different MDSE methodologies; we envision the support for plugging legacy frameworks for each methodology into the layer below the modeling layer of our framework as long we can formally map the concepts captured in the unified model to those of the underlying formalisms of the plugged-in frameworks.

The architecture of the proposed framework is accompanied by a process model, as an answer to RQ4, which describes the workflow to be followed as a guide to using the framework.

Con 2. A high level language whose syntax uniformly combines the DES concepts for simulation, FM and enactment to support the specification of unified models for the three methodologies: This contribution directly answers the research question RQ2 and technically provides the answers toRQ3. Since none of the three MDSE methodologies federated in the framework proposed in above is based on a formalism that subsumes all the concepts expressed in others, we have considered defining a new language, **High Level Language for System Specification (HiLLS)**, that methodically combines the general concepts underlying the three methodologies to specify the unified model described in the framework. HiLLS' abstract syntax is built from the integration of general DES concepts from DEVS (Discrete Event System Specification) [ZPK00], Object-Z [Smi12] and UML (Unified Modeling Language) [RJB04] using metamodel integration techniques proposed by Emerson, and Sztipanovits [ES06].HiLLS adopts symbols similar to those of UML class diagram and (resp.state charts [Har87]) for graphical presentation of system's structure (resp.behavior). It is important to note, however, that HiLLS is not an extension of neither UML nor state chart, it only reuses parts of their notations for communicability.

HiLLS' abstract syntax is mapped to three semantics domains for simulation, formal analysis and enactment. Hence HiLLS is, in fact, at the kernel of the framework we propose, serving as the seam that holds the disparate analysis methodologies together in the framework.

Con 3. A preliminary framework for the enactment of DES: Enactment methodology has yet to permeate significantly into the MDSE practice with DES unlike simulation and formal analysis which both have well established formalisms and operational/logical protocols that are accepted by considerably large communities. The current practices of enactment for DES are mostly based on UML and SysML (System Modeling Language) [FMS14] and their profiles. We propose in this thesis, a "*preliminary*" framework for the enactment of DES which uses some object-oriented design patterns [GHJ+95] to express DEVS concepts for the scheduling and execution of events based on physical clock time.

Con 4. Formal mappings of HiLLS concepts to simulation, FM and enactment semantics domains: To consolidate the answers provided inCon 1, RQ2 and RQ3 above to the research questions, we define the mappings of HiLLS concepts to DEVS (resp. Z[Spi88] and Temporal Logic [Pnu77, CES86])for discrete event simulation (resp. logical analysis by exhaustive state explorations) and a mapping to the enactment framework mentioned in Con 3 above for an operational execution of a software prototype of the system under study. Given a software environment for editing HiLLS models, these mappings can be followed to implement the model transformations to automated synthesis of the different artifacts from a given HiLLS model.

Prior to writing this thesis, some of the outputs obtained in the course of the work have been discussed in journal and conference publications as contributions to the literature: abstract descriptions of the framework in Con 1and preliminary research directions were presented in [AT15a, AT15b]; these were extended with details and a simple application example discussed in [AT16]. The preliminary description of the concept of enactment in the context of the work and the enactment framework mentioned in Con 3was presented in [AMT15] while early results leading to Con 2 and a part of Con 4 were discussed in [MAT15, AMT16]. In all cases, this thesis gives a definitive account of the different components of, and the entire work.

1.5 THESIS OUTLINE

In Chapter 2, we will present the state of the art in combining multiple MDSE approaches for system design and analysis and highlight the extent to which they have attempted to solve the problems addressed in this thesis as identified in Section 1.2.

Chapter 3 provides backgrounds on the theories, formalisms, techniques, technologies and tools to be used in subsequent chapters for detailed presentations of the contributions of this thesis. This chapter also introduces the synopsis of the beverage vending system, which is used as a running example throughout the thesis to illustrate the contributions.

We will present, in Chapter 4, a detailed discussion of the framework proposed in Con 1. Recall that Con 1 specifically provides an answer to the research question RQ1; RQ1 itself gets inspirations from a decade-long research initiative [Tra08] which envisioned a multi-faceted MDSE project; a previous Doctoral thesis [Tou12] has explored a branch of this project. Therefore, we will first present an introduction of the research initiative and the aspect studied in [Tou12]. This will be followed by an abstract presentation of the methodology of the present thesis, the architecture of the proposed framework and the process model that describes the workflow to be used as a guide to use it.

The preliminary framework for the enactment of DES described under Con 3 will be presented in details in Chapter 5.

Chapter 6 will provide detailed accounts of Con 2. We will present in this chapter how the abstract syntax of HiLLS was built by methodical integration of concepts from system-theoretic and software engineering formalisms to define a language that supports the specification of unified models for disparate MDSE approaches. This will be followed by a presentation of the concrete notations used by the modeler to render the language's concepts in models.

In Chapter7, we will elaborate on Con 4, which is the mapping of HiLLS concepts to the three semantics domains.

Finally, we will conclude the thesis in Chapter 8with a summary of the thesis, highlights of problems addressed and the proposed solutions, and plans for future work.

2 LITERATURE REVIEW

2.1 INTRODUCTION

In this chapter, we present the state of the art in MDSE practices vis-à-vis the challenges which this thesis seeks to address. We classify the approaches in the literature into two categories based on whether they are targeted at facilitating specific analysis methodologies or at integrating multiple methodologies; these are:

- Methodology-specific approaches
- Pair wise integrations of methodology-specific models

The next two sections of this chapter elaborate on the two categories. In each category, we present the fundamental principles of some selected approaches and highlight the extents to which they have addressed the problems - P1, P2 and P3 - identified in Chapter 1, Section 1.2.Though there are numerous formalisms underlying the approaches and MDSE environments in each of the two categories, we give preference to DEVS (Discrete Event System Specification) as simulation formalism, Z language and its extensions as formalisms to specify systems for formal analysis and Temporal Logic (Logic) as formalism for defining required properties for formal verifications. We will provide, in details, the premises upon which our preferences are based in later chapters; one general consideration, which is applicable to all of the three formalisms, is that they are considerably universal in their respective domains. If necessary, we invite the reader to have a glance at Section 3.2 for introductory discussions of these formalisms.

The chapter concludes with a comparative discussion of the selected approaches, and highlight of the lessons learned from the state of the art and the persistent issues that necessitate the work reported in this thesis.

2.2 METHODOLOGY-SPECIFIC APPROACHES

This arguably comprises most of the prominent approaches to the practice of MDSE. It involves the development of a model-driven environment to provide tooling supports for a specific formalism and targeted at a particular analysis methodology, i.e., simulation, formal analysis or enactment. Essentially, such environments offer high-level notations, which are graphical in most cases, for creating and editing system models, which are used to start the systematic and progressive synthesis of low-level artifacts until executable codes are obtained for the required execution platform or environment.

The fundamental objective, which is shared by most approaches in this category, is usually to alleviate the complexity and rigor of direct system specification with the underlying formalism through high level modeling interfaces. It is particularly meant to address the problem identified in P1 - *Lack of requisite logic and mathematical skills to deal with most formalisms* - in the previous chapter (see Section1.2).Moreover, these approaches are generally motivated, inter alia, by the possibility of making the underlying formalisms and their capabilities accessible to wider communities including non-expert users, ease of communication among stakeholders and (semi-) automated synthesis of executable analysis codes from the high-level models.

Generally, MDSE environments under this category are often focused on specific analysis methodologies and largely isolated from others. This general characteristic is depicted in Figure2.1 with each of the three methodologies considered in this thesis - simulation, formal methods and enactment - residing in isolation in one of the three orthogonal planes of the three-dimensional (3D) coordinate. i.e., simulation, formal methods and enactment processes reside in isolation in the X-Y, Y-Z and X-Z planes respectively. The criteria for choosing formalism(s) within the plane of interest may include the nature of the system under study, the kind of property to be studied and the experience of the analyst among others.

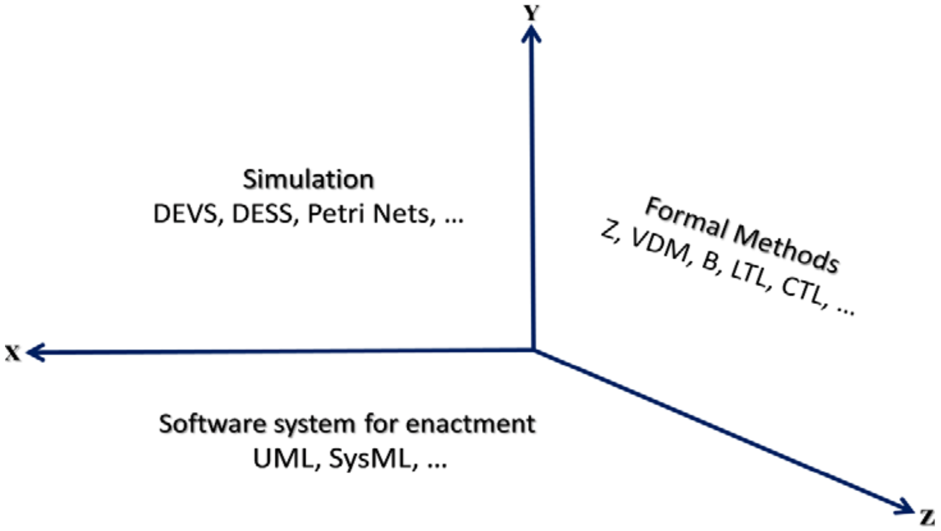


Figure2.1 Schematic illustration of isolated MDSE practices
The X-Y, Y-Z and X-Z planes of the 3D coordinates depict isolated MDSE frameworks based on specific formalisms for simulation, formal methods and enactment respectively.

One significant benefit of this approach is the separation of concerns. i.e., the modeler in each plane of Figure2.1 creates a formal specification of the system under study from the point of view of the plane while abstracting away from the peculiarities of the methodologies in the other two planes thereby leading to a considerably simplified and focused model in each plane.

Unfortunately, however, since a comprehensive study of a system often requires combinations of multiple analysis methodologies, one would likely need to create separate models of the same system in each of the three orthogonal planes. The implication of this is that disparate models of the same system independent MDSE environments would be needed; this phenomenon of dealing with multiple disconnected views of the system in the different planes can be susceptible to miscommunication among domain experts [BSD+12]. Moreover, the creation, and more importantly, the repeated update of the different models during the iterations of analysis processes can be herculean since any change in the system variable may require that all models in the different planes be manually updated. In other words, the approach ignores problems P2 and P3 identified in the previous chapter.

Usually, the development of the modeling interfaces for MDSE environments in this category follow one of two prevalent styles:

- Domain-Specific Language (DSL) style
- UML/SysML profile style

The UML or SysML profile-based modeling style rely on the universality of UML for software systems modeling, its wide acceptability in the industry and availability of supporting tools to leverage the complexities of abstract formalisms and make them accessible to wider communities of users. Another advantage is that it minimizes the cost and risk of adoption by integrating with existing methods and tools [SB06]. Modeling interfaces in this category often use some restricted stereotypes of the different kinds of formalisms in the UML or SysML families of diagrams to describe system's structures and behaviors. Particularly they use UML class and component diagrams (resp. SysML block diagrams) to describe systems' structures where hierarchical constructions of composite system models are realized with UML sub-components (resp. SysML's Block Definition Diagram) and formalisms like the activity, sequence and state diagrams to model systems' behaviors.

The DSL style promotes the creation of a DSL specifically based on an existing formalism. The DSL's abstract syntax is specified to capture the concepts described in the underlying formalism; the concrete syntax is developed with high-level cognitive notations to shield the DSL's users from the complexity of the underlying formalism. Intuitively, the operational/execution/logical framework of the underlying formalism provides semantics domain for the DSL. Proponents of the dedicated DSL style argue that it allows the language engineer to create more suitable notations to effectively and concisely represent the concepts of the underlying formalism rather than relying on notations of some universal language that have been defined originally for some other purposes [SK10, TB11].

There is a plethora of methodology-specific MDSE environments and we cannot possibly give an exhaustive list of existing ones; we would rather limit our references to those that are most relevant to the work in this thesis. Examples of DSL-based environments for discrete event simulation include PowerDEVS [KLP03], DUNIP [MM13], CD++ [WCD01, Wai02, BWC13], DDML [Tra09, IMT12, MIT12], MS4Me [ZS13, SZC+13], DEVS Diagram [SK10] and DEVSML 2.0 [MD12]. UML/SysML-based environments for discrete event simulation include euDEVS [RJM+09] and a host of others including[Zin05, NDK07, NDM+08, HRM07]; light surveys of some of these environments comparing their relative strengths in expressing different aspects of systems for DEVS-based simulation can be found in[FBT+14, AMT16].Prominent DSL-based MDSE environments for formal methods include UPAAL [BDL04]and CZT [MU05, MFM+05]; there are also a host of UML/SysML-based environments such as UML-B [SB06], ModelicaML [PAF07,Sch09, SFP+09], SysML4Modelica [RPC+12], TURTLE-P [ASK06], TEPE [KAS11], etc.

While many of the methodology-specific MDSE environments are built to be compatible with unique implementations of the operational semantics of their underlying formalisms, some adopt an integrative approach with considerably generic modeling notations to harness the synergy of intra-disciplinary tools towards boosting the utilities of research outputs in their respective methodologies; examples of such frameworks include the DEVS Unified Process (DUNIP) [MM13a, MM13b, Mit07], Community Z Tools (CZT)[MU05, MFM+05] for simulation and formal methods respectively. The rest of this subsection elaborates on these integrative frameworks.

2.2.1 DEVS Unified Process (DUNIP)

DEVS Unified Process (DUNIP) was first proposed by Mittal in his Doctoral thesis [Mit07], then revised and further developed by Mittal and Martín [MM13a, MM13b] to explore the integration of various concepts that had been developed through decades of research in DEVS-based simulation methodology and apply same to the design and analysis of systems of systems in a full systems engineering life cycle. The overall objective of the framework is to harness the benefit of automated transformations in MDE to bind different phases of a rigorous MBSE process backed by the DEVS theory for a transparent simulation of systems of systems in a net-centric M&S setup.

The fundamental visions of, and MDSE processes in DUNIP are comprehensively captures in the DEVSML 2.0 stack [MM13a], which is a multi-layered architecture with infrastructures at the different layers that work together to realize the modeling and simulation-based verification of DES on a stand-alone or net-centric simulation platform. The top layer of DEVSML 2.0 stack contains the DEVS Modeling Language version 2.0 (DEVSML 2.0) [MD12], a textual DSL for expressing systems' structure and behavior based on DEVS theory. A Model written in DEVSML 2.0 is persisted in XML and considered as a Platform-Independent Model (PIM) to make it compatible with DEVS-based simulators implemented in multiple platforms. The PIMs in the DEVSML 2.0 layer is transmitted through some DEVS-compliant middleware and APIs (in the lower layers) to a net-centric infrastructure that generate and deploy platform-specific simulation codes on a distributed multi-platform federation of simulation engines based on Java, C++, etc. at the bottom of the stack. DEVSML 2.0 also serves as the interface to integrate DUNIP with DSLs and languages such as Business Process Modeling Notations (BPMN) [Whi04], UML, SysML so that domain experts can create system models in the problem domains and transform them (semi-)automatically to DEVSML 2.0-compatible format.

In essence, DUNIP fosters the federation of diverse DEVS-based simulation engines to provide a transparent simulation support for DSLs via a net-centric virtual machine. Hence, it addresses problem P1 identified in this thesis by shielding the modeler from the rigor of direct system specification with raw DEVS constructs through high-level concrete syntax for DEVSML 2.0 and the possibility of transforming DSMs (domain-specific models) into the XML format of DEVSML 2.0 for onward transformation to DEVS-based simulation codes. DUNIP partially addresses the problem of interoperability and co-existence of tools identified in P3, albeit only within the realm of simulation. The model portability and tool interoperability we referred to in P2 and P3is not limited to intra-disciplinary scope as provided in DUNIP, it is, in fact, more of inter-disciplinary between disparate analysis methodologies like simulation, formal analysis and enactment.

2.2.2 Community Z Tools (CZT)

The Community Z Tools (CZT) [MU05, MFM+05] is an open-source integrated framework for building formal methods tools for standard Z and Z dialects/extensions with the aim of providing a comprehensive development environment for formal specifications written in Z and its extensions such as Object-Z and Circus from typesetting to verification. CZT promotes the integration of existing and new Z tools via the Z Markup Language (ZML) [UTS+03], a standard XML interchange format for Z.

Z [Spi88] is a formal specification language suitable to precisely specify state-based systems, and analyze them via proof, animation, test generation, etc. Z is widely used by FM practitioners in the domain of state-based systems and it has been approved as an ISO standard in the year 2002 [ISO02]. However, few of existing tools for *editing, parsing, type checking, analyzing and animating formal specifications in Z and its extensions* conform to the Z ISO standard; they are based on diverse variants of the formalism thereby making it difficult for them to interoperate. CZT proposed to solve this problem with the ZML XML schema at the core of the framework, which serves as the interchange format that can be used to transmit parsed and type-checked specifications between the various tools federated in the framework.

CZT is implemented in jEdit and Eclipse, both of which are open-source integrated development environments that provide convenient platforms for progressive development and integration of tools for general and specific purposes. The Eclipse CZT offers graphical Z editor plug-ins for authoring Z specifications with the Unicode symbols based on the Z ISO standard and plug-ins for tools offering the various operations and analyses on the specification. It also allows for integration with the Eclipse plug-in for theorem proverbs like Z/EVES [MS97, Saa03] and CadiZ [TM95], refinement formalisms like Circus [WC01] and possible transformation of the Z specifications into B specifications for formal analysis capabilities that are specifically possible with the B method [Lan12].

By the integration of disparate Z tools in the same framework through the sharing of a unified specification, CZT proffers a *partial* solution to problems P2 and P3 identified in the previous chapter. It is a partial solution because the tools and methodologies integrated in the framework only help the user to explore system's properties using different FM approaches. i.e., it does not provide support for simulation and enactment. Though CZT offer the interface for system specification, the user needs the skills and rigor to deal with raw Z specifications in order to use the symbols provided in ZML for system specification; hence problem P1 is not addressed in CZT.

2.3 PAIR WISE INTEGRATIONS OF METHODOLOGY-SPECIFIC APPROACHES

This approach involves the pair wise integration of system models and MDSE processes in the different planes of Figure2.1above, mostly by model transformations. This is usually done by identifying correspondences between elements of chosen formalisms in the different planes in order to define the mapping rules between them so that a model in one plane may be used to drive the (semi-) automated synthesis of models in other planes to take benefit of the different analysis capabilities in the different planes. Examples of such combined use of simulation and formal analysis include [KCS03, Tra05, Tra06a, Tra06b, TFH09, TSF09, YHF14]. Similarly, several proposals have been made to bridge the gap between software and enactment models with FM [LCA04, LP99, SAB09, LSM+10, Men16, BMC+12,KNT+14] while some efforts to achieve pair wise transformation between simulation formalisms and software and enactment methodologies are reported in [NDA10, RMZ07, SFP+09,SFP+09, SV11, Hou12].

Through building of bridges between disparate formalisms, this approach addresses some of the shortcomings of the isolated use of different methodologies presented in the previous section. Particularly, it promotes model reusability and helps to reduce the task of creating and updating models during analysis cycles through (semi-)automated model transformation, which is also a way to hide the complexities of system specifications with the target formalisms. Unfortunately, we cannot always guarantee a *surjective* mapping between the source and target formalisms of the transformations; that is, there is no guarantee that every element of the target formalism has corresponding element(s) from which it can be derived in the source formalism. For instance, the authors of [ZN16] noted that formalisms that are oriented to verification tend to lack many functions that exist in simulation formalisms. Hence the synthesis of the target models are, in many cases, *semi-automatic* thereby making it a necessity to still introduce some manual processes of completing the update of some models during the analysis cycles.

The rest of this subsection presents some pair wise combinations of simulation, FM and enactment in the literature.

2.3.1 Z-DEVS

Z-DEVS was defined by Traoré in a series of publications [Tra05, Tra06a, Tra06b] as an approach to integrate FM with the DEVS simulation framework towards making DEVS simulation models amenable to formal analysis and symbolic reasoning. Traoré was motivated by the benefits of rigorous proofs of properties of simulation models with respect to their design and use requirements, and the likelihood of better understanding of concepts such as verification, validation, reuse and composability, which are important issues in M&S practice. He proposed the Z-DEVS to extend the DEVS paradigm with a logical framework that permits the use of suitable tools for the exploration of properties for early detection and resolution of conflicts between requirements and missing assumptions.

The formulation of the Z-DEVS approach was inspired by Paige's meta-method for FM integration [Pai97], which postulates the full translation of the concepts of formalism to FM in order to take benefit of all possible kinds of analyses available through the target FM. Hence, Z-DEVS provides formal semantics for the DEVS paradigm through the expression of the concepts described in DEVS model and simulator as modular and hierarchical Z schemas. This offers the grand benefit of using all relevant Z tools and techniques to analyze the simulation model and simulation protocols. According to Traoré, the Z-DEVS approach also opens an additional prospect of methodically investigating the suitability of a given experimental frame for a given simulation model.

The Z-DEVS approach was illustrated with a case study, which presents the Z equivalent (specified with the Z/EVES editor) of a given DEVS specification and its simulator for some logical analysis. This makes Z-DEVS unique in that similar approaches only deal with the logical analysis of system model but not its simulator.

In the context of this thesis, the Z-DEVS approach proffers a partial solution to our thesis problem P2since it proposes the integration of simulation and FM but not enactment. Unfortunately, it does not address P1; apparently, it remains focused on its prime objective - to provide a formal framework for the DEVS paradigm - with little attention to the ease of dealing with the combined formalisms. In fact, the user of the approach will have to contend with the rigor of both DEVS and Z formalisms. An automated synthesis of the target Z schemas from a high level DEVS specification will likely alleviate this constraint.

2.3.2 DEVS Compiler

The DEVS Compiler was recently proposed by Trojet and Berradia [TB15] to improve the quality of DEVS simulation models through integration with FM; the idea is to subject the simulation models to formal verification for early detection and resolution of eventual errors before the commencement of simulation processes.

DEVS Compiler is yet another integration of DEVS with Z but with focus on verifying two specific properties of models: *determinism* and *completeness*. It proposes to precede the simulation process with a two-step formal verification procedure: 1) automatically transform a DEVS model into an equivalent Z specification, and 2) verify the consistency of DEVS model via the generated Z specification using available Z tools.

The implementation of DEVS Compiler is embedded within the LSIS-DME M&S environment for DEVS [HZ07]. Essentially, LSIS-DME offers a graphical modeling interface for creating DEVS models, which are persistent in XML format. With an XML representation of a DEVS model as source, an XML-based equivalent Z specification is automatically generated upon the invocation of the DEVS Compiler. The Z specification so generated is fed into the Z/EVES theorem prover [Saa97, MS97] to prove (or reveal the violation of) the determinism and completeness properties of the model.

DEVS Compiler proffers a solution to problem P1 of this thesis by taking benefit of the graphical modeling interface offered by LSIS-DME, its host MDSE environment, and the automated synthesis of the Z specification to shield the user from the rigor of the underlying formalisms. Unfortunately, DEVS Compiler's support for formal analysis is very limited in its current state since only the verifications of completeness and determinism properties are possible; hence, it is fair to say that the approach proffers only *partial* solutions to problems P2 and P3.

2.3.3 Constraints-Based DEVS Framework (ϕDEVS)

Trojet, Frydman and Hamri [TFH09] proposed the Constraints-based DEVS framework ($\phi DEVS$) to achieve a *lightweight* introduction of Z to the classic DEVS framework for formal verification of static functional properties of DEVS models. Essentially, the intent of the framework is to permit its user to describe the behavior of DESs with DEVS and capture static constraints with predicate logic so that the combined behavior and constraints specification can be translated into Z to verify whether the behavior will satisfy the constraints at runtime or not.

To achieve this objective, the authors propose the introduction of an additional symbol,ϕ into the mathematical structure of classic DEVS. ϕ represents a set of static constraints, specified with predicate logic, on the system's state variables and the resulting $\phi DEVS$ is referred to as the *Constraints-based DEVS*. According to them, their choice of Z for lightweight extension of the DEVS framework for formal analysis is premised on the fact that both DEVS and Z are used to specify sequential systems with a state/transition approach and both are widely used in their respective domains of application.

$\phi DEVS$ has been implemented as a component called the $\phi DEVS - PROCESSOR$ in the LSIS-DME M&S environment [HZ07]. $\phi DEVS - PROCESSOR$ provides a graphical interface to load a DEVS model specified with the model editor of the host environment (LSIS_DME), specify a set of static constraints (ϕ) on its state variables inside the *constraints box* offered by the interface and automatically generate a Z equivalent of the resulting $\phi DEVS$ in XML format. The XML file is used as input to the Z/EVES theorem prover, which does some type checking and performs the proof obligations to determining whether all the constraints specified in ϕ are respected by the DEVS model.

By courtesy of its host environment $\phi DEVS$ is able to contribute a solution to problem P1through automated synthesis of the Z specification to shield the user from the rigor of the underlying formalisms. It is important to note also that $\phi DEVS$' support for formal analysis is limited to the verification of static properties of DES; for instance, it does not support the specification and verification of temporal requirements to check properties like liveness, safety and fairness.

2.3.4 DEVS and Temporal Logic of Action+ (DEVS-TLA+)

Cristiá [Cri07, Cri08] defined the DEVS-TLA+ on the one hand, to project the DEVS formalism into the FM community so that it may be adopted by FM researchers for the specification and verification of event-based systems; and on the other hand, to provide the basis for formal semantics for rigorous logical reasoning with DEVS models. He argued that DEVS and Temporal Logic of Action+ (TLA+) [Lam02] share a common conceptual foundation and went ahead to discuss the conceptual equivalences between the elements of the two formalisms. Sequel to that, he made an initial proposal of a procedure to encode atomic DEVS models in TLA+ modules in [Cri07]; which was revised and extended with the procedure to encode coupled DEVS models in TLA+ specifications in [Cri08].

One interesting difference between the DEVS-TLA+ approach and most other combinations of DEVS with FM is that it supports the specification and verification of temporal properties like safety, liveness and fairness. According to Cristiá, having a TLA+ specification of a DEVS model allows for formal verification of the model with the tools already available for TLA+ specifications.

Unfortunately, the translation of DEVS model to TLA+ specification is done manually. The implication of this is that a potential user with limited skills in dealing with the FM notations will most likely not be able to use the approach. In fact, someone must combine the skills required in dealing with the mathematical rigors of both DEVS and TLA+ to use the approach. Thus, the DEVS-TLA+ approach preserves the problems P1 and P3 of this thesis but provides a partial solution to P2.

2.3.5 ProMoBox

ProMoBox (Properties and design Models developed Boxed in concert) [MWB+13, MDL+14, Mey16] is an environment that provides high-level formal verification supports for Doman-Specific Languages (DSLs). Apparently inspired by the architecture of domain-specific model checker of Visser et al. [VDW12], the authors of ProMoBox argued that the conventional practice of translating Domain-Specific Models (DSMs) into mathematical formalisms before specifying the required properties with logic-based formalisms violates the principle of domain-specific modeling, which promotes the use of domain notations for system modeling. As earlier suggested by Visser et al., the specification of models of systems, requirements, environment/input and analysis results should be achievable with high level domain notations while all low level artifacts should be hidden from the domain expert. Meyers et al. proposed a solution in the form of ProMoBox to this problem.

ProMoBox is a framework that integrates the definition and verification of temporal properties in discrete-time behavioral DSMLs, whose semantics can be described as a schedule of graph rewrite rules. Essentially, ProMoBox seeks to raise the specification of temporal properties and the interpretation of verification traces (e.g., counterexamples) to the same abstraction level as the system model in order to make FM techniques readily accessible to domain experts. Therefore, ProMoBox allows its user to design a DSML for modeling not only the system, but also its properties, input model, run-time state and output trace. ProMoBox's DSML development process allows for semi-automated synthesis, from an annotated metamodel, of five sublanguages, which share a domain-specific syntax while the DSML's operational semantics is specified as a transformation annotated with input and output information.

In ProMoBox, a system's model and its associated requirement model are translated to Promela, and the properties are verified with the Spin model checker [Hol97]. The traces of the verification are transformed into the problem domain as a trace model to animate the counterexamples if any.

From the standpoint of this thesis, ProMoBox proffers an interesting solution to problem P1, particularly through the modeling of temporal properties and visualization of verification results using notations of the problem domain as this will help to shield domain experts from the rigors of the underlying notations and techniques. However, its primary target user is the language engineer that develops the DSML. Hence, a domain expert will always require the service of the language engineer to build a DSML specifically for the problem to be solved; this may be a disadvantage if the problem is one that hardly reoccurs to take full benefits of the efforts of developing the DSML. On the contrary, the context of the problems investigated in this thesis is to build the MDSE environment based on considerably universal formalisms for FM, simulation and enactment so that it can be applicable to many problems. ProMoBox has been implemented in AToMPM (A Tool for Multi-Paradigm Modeling) [SVM+13], framework for designing domain-specific modeling environments, performing model transformations, manipulating and managing models. It could be possible to extend a developed DSML with simulation support by specifying a simulation-based operational semantics for it; this is however, not stated explicitly by the authors of ProMoBox.

2.3.6 Model-Driven Development for Modeling and Simulation (MDD4MS)

The Model-Driven Development for Modeling and Simulation (MDD4MS) was proposed by Çetinkaya in her Doctoral thesis [Çet13]. She argued that though the importance of simulation conceptual model (CM) to the accurate development of M&S models was widely acknowledged in the literature, the systematic transformation of CMs, through intermediate models, to executable simulation models had not been sufficiently studied. She described the CM and two other models required in an M&S process as follows:

i. A CM, which is non-executable higher-level abstraction of the system under study that represents the structure of the system and what will be modeled in the future executable simulation model.

ii. A Platform-Independent Simulation Model (PISM) is a mathematical description of the processes and activities in the CM so that mathematical or computational analyses can be manually conducted.

iii. A Platform-Specific Simulation Model (PSSM) is derived from the PISM and the details of a specific execution platform towards the synthesis of an executable model that can allow the simulation to be carried out on a machine.

Çetinkaya argued that non-consumption of the CM in any development iteration involving other models creates a semantics gap between CM and PISM that may lead to lack of model continuity in all stages of the model development. To address this problem, she proposed the MDD4MS framework to manage the simulation process that encompasses the three stages of model development. Fundamentally, MDD4MS mirrors the layered architecture of the OMG's MDA framework with CM, PISM and PSSM in the place of MDA's CIM, PIM and PSM respectively.

This generic framework was concretized with the CM, PISM and PSSM created based on BPMN, DEVS and a Java-based DEVS simulation framework known as DEVS Distributed Simulation Object Library (DEVSDSOL) [SV09] respectively. With transformation rules written in ATLAS Transformation Language (ATL) [JAB+06], substantial parts of the PISM - which is manually refined - can be obtained from the CM through a *partial* model transformation process; though not explicitly stated, the transformation is "partial" apparently because the mapping of BPMN (source formalism) to DEVS (target formalism) is not *surjective*. The refined PISM is used in another partial model transformation process to generate a PSSM based on DEVSDSOL.

Though not explicitly stated or claimed by Çetinkaya, the BPMN-based CM can arguably be independently refined to an enactment model; if it is considered in this sense, then we can reasonably say that an enactment model is being transformed into a DEVS-based model for simulation. Consequently, it will be fair to say that MDD4MS proffers a partial solution to problem P1with the graphical editor for BPMN that is subsequently used to drive the synthesis of a substantial part of the DEVS simulation model. We could also claim that MDD4MS has the potential to partial solutions to problems P2if the BPMN-based CM was developed to a full enactment model and consumed in an enactment process. Furthermore, the framework does not address any form of formal analysis of the system under study with FM. In fact, it will be very difficult, if not impossible, to make the models amenable to formal analysis because most system concepts defined in the metamodels provided are of type *String*; hence, it will be difficult for any tool to recognize the true types of the model artifacts since they are all seen as pure strings.

2.3.7 Homomorphic Extension of Formal Analysis Model for Simulation

Zeigler and Nutaro [ZN16] recently proposed a framework to integrate the FM and simulation methodologies through an incremental model development process, primarily for the verification and validation of simulation models of System of Systems. They argued "*that the combination of simulation and formal verification gives a much more powerful capability to test designs than can be achieved with either alone"*. To combine the two approaches, Zeigler and Nutaro hypothesized that: "*System morphisms can map a model expressed in a formalism suitable for analysis (e.g. timed automata or hybrid automata) into the DEVS formalism for the purpose of simulation. Conversely, it is also possible to go from DEVS to a formalism suitable for analysis for the purposes of model checking, symbolic extraction of test cases, and reachability, among other analysis tasks"*.

The authors proposed that a design process to incorporate FM and simulation methodologies should start with an abstract verification model, V_m, which can be logically analyzed using FM to obtain absolute answers about a system's behavior under an ideal condition. The failure of V_m to satisfy the required properties at this stage is an indication of fundamental flaws in the system design that must be identified and corrected. Once it is verified that V_m satisfies the requirements, it is formally extended into a simulation model, S_m, with operational details to explore scenarios that are outside the scope of the formal verification. The V_m-to- S_m model extension or refinement should be done such that the source V_m is a homomorphic simplification of the target S_m. The assumption is that S_m retains the proven properties of the V_m, through system morphisms. i.e., any property proved to hold in V_m would also hold for S_m in the same context and the latter can subsequently be used to verify the operational practicality of the ideal properties of the former.

The authors, however, acknowledged that the homomorphism between V_m and S_m might not necessarily imply the preservation of properties in all cases especially when the experimental frame of S_m evolves. Thus, we can arguably conclude that more research work is needed to provide a concrete demonstration of where it would be most convenient to apply this approach.

From the perspective of this thesis, this approach offers a partial solution to problem P2 in that it allows a formal extension of an abstract verification model into an operational model for simulation. However, since the verification model lacks many functions that exist in the simulation formalism as acknowledged by the author, there is little chance of automating the extension process; this can be more herculean if the modeler has to deal with the raw verification and simulation formalisms.

2.4 CONCLUSION

In this chapter, we have presented a literature review of the state of the art in MDSE frameworks and environments with particular focus on DEVS-based simulation and the specifications of systems (resp. requirement properties) with Z (resp. Temporal Logic) for formal verifications. We discussed existing approaches under two categories: 1) methodology-specific approaches, which comprise the approaches that are dedicated to specific analysis methodologies; and 2) pair wise integration of methodology-specific approaches, which comprise the various approaches to discipline combinations of two disparate analysis methodologies to get more benefits than could possibly be obtained with either of the two methodologies in isolation.

A summary of the literature review and comparative view of the studied approaches in relation to the extent to which they have addressed the challenges identified in this thesis and the overall visions of the thesis is presented in Table2.1below.

Table2.1 Comparison of existing MDSE approaches

‡: Full support; †: Partial support; – : No support;✓: Yes;✕: No
Cat.: Category; Sim.: Simulation; FM: Formal Methods; Ena.: Enactment;
Auto.: Automated model transformation between integrated models

Approaches			MDSE methodologies supported			Thesis problems solved			Auto.	Unified formalism
Cat.	**Example(s)**	**Objectives**	**Sim.**	**FM**	**Ena.**	**P1**	**P2**	**P3**		
Methodology-specific	DUNIP[MM13a, MM13b]	To integrate the results of DEVS-based researches for a full systems engineering life cycle.	‡	–	–	‡	–	–	†	✓[1]
	CZT [MU05, FM+05]	To integrate legacy and new analysis tools for formal specifications written in Z and Z extensions.	–	‡	–	–	†	†	‡	✓[2]
Pair wise integration	Z-DEVS [Tra05, Tra06a, Tra06b]	To provide a logical framework for rigorous formal analysis of models and simulators in the DEVS paradigm.	‡	†	–	–	†	–	–	✕
	DEVS Compiler [TB15]	To integrate DEVS with Z for the verification of completeness and determinism properties of the former prior to simulation.	‡[3]	†[4]	–	‡	†	†	†	✕
	$\phi DEVS$ [TFH09]	To extend the DEVS framework with the	‡[5]	†[6]	–	‡	†	†	†	✕

[1] DEVSML 2.0 is the unified formalism that facilitates the integration of disparate DEVS simulators
[2] ZML serves as the unified formalism for the integration of analysis tools for Z and its extensions
[3] The host environment, LSIS-DME, provides simulation support
[4] Supports for verification of consistency and completeness properties only
[5] The host environment, LSIS-DME, provides simulation support
[6] Support for specification and verification of static constraints on system's state variables only

Approaches			MDSE methodologies supported			Thesis problems solved			Auto.	Unified formalism
Cat.	Example(s)	Objectives	Sim.	FM	Ena.	P1	P2	P3		
		capability to specify and verify static constraints on system's state variables.								
	DEVS-TLA+ [Cri07, Cri08]	To project DEVS into the FM domain in order to promote its adoption by FM practitioners for the specification and verification of the behaviors of DES. To provide a basis for formal semantics for DEVS.	‡	†[7]	–	–	†	–	–	✕
	ProMoBox [MWB+13, MDL+14, Mey16]	To integrate verification with DSMLs for high level modeling of a system and its requirement properties as well as the visualization of counterexamples using notations defined for the problem domain.	†[8]	‡	–	‡	†	–	†	✕
	MDD4MS [Çet13]	To realize model continuity in M&S processes through the consumption of conceptual models in successive transformations for the synthesis of simulation codes.	‡	–	†	‡	†	†	†	✕
	Model extension	To integrate FM and	‡	†	–	–	†	–	–	✕

[7] Specifically focused on verification of temporal properties

[8] Possibility to support simulation through the specification of simulation-based operational semantics for a DSML in the host environment, AToMPM [SVM+13]

Approaches			MDSE methodologies supported			Thesis problems solved			Auto.	Unified formalism
Cat.	Example(s)	Objectives	Sim.	FM	Ena.	P1	P2	P3		
	by system morphism [ZN16]	simulation methodologies through an incremental model development process, for the verification of simulation models of System of Systems.								
Visions of this thesis		To build an integrative framework that can be continuously populated with best practices in MDSE for simulation, formal methods and enactment such that the various components are federated through a seamless sharing of high-level system model.	‡	‡	‡	‡	‡	‡	‡	✓

2.4.1 Lessons Learned

We have learnt from the analysis of the information in Table2.1 that a unified and generic formal representation of models is essential for the systematic integration of disparate tools; this has been demonstrated by DUNIP and CZT each of which relies on a generic system representation to facilitate the portability of models between disparate tools. We have also observed from the presentations of the fundamental principles of approaches in the pair wise integration category that whenever a framework integrates FM with any other methodology, formal verification activities always take the lead in the overall process of the MDSE framework. Obviously, this is often done to give credence to the models used in subsequent analysis activities since a formally correct and consistent model is a prerequisite for a reliable simulation result.

2.4.2 Perspectives

We can conclude from our analysis of the state of the art so far that none of the existing MDSE approaches or environments fully addresses problems P1, P2 and P3 as envisioned in this thesis. That is, none of them fully supports the complementary application of simulation-, FM-, and enactment-based analysis methodologies for the study and investigation of the properties of DESs. In most cases, either the modeling notations lack supports for modeling some of the aspects of systems or they are less suitable for the intended users especially when they require dealing with multiple notations, each of which is considerably complex on its own.

In this thesis, we will demonstrate that a high level modeling of a DES, its context and required properties in a unified formalism and the systematic derivations of low-level artifacts for simulation, formal analysis and enactment is realizable within an integrated MDSE framework. Though this claim may sound too ambitious, as it requires nontrivial research efforts, we will provide the theoretical foundation for the tasks and set the guidelines for its technological implementations as well as open the perspectives for further research in the same direction.

We begin by presenting, in the next chapter, some theoretical and technological background to aid our presentations in subsequent chapters.

3 BACKGROUND

3.1 INTRODUCTION

This chapter presents the theoretical and technological backgrounds of the contributions of this thesis, which will be presented in subsequent chapters and a running example that will be used in the rest of this document. Essentially, the thesis' contributions revolve around the application of MDE techniques on the one hand to facilitate the use of MDSE formalisms through high level modeling interfaces, and on the other hand to realize the integration of disparate MDSE methodologies behind a unified and generic modeling interface for complementary analysis of static and dynamic properties of systems. These formalisms and MDE techniques are introduced in this chapter.

In the sequel, we start with background on relevant formalisms in the next section; we also introduce, in the same section, a beverage vending system as a running example to aid our discussions of the formalism and to serve as a running example in subsequent chapters of this document. This is followed by background on MDE in Section 3.3. Section 3.4 presents the concept of megamodeling, which provides the basis for defining the formal relationships between the different artifacts in an MDE-based architecture. We then present an overview of the essential elements of a language engineering process in Section3.5 before concluding the chapter in Section3.6.

3.2 MDSE FORMALISMS

In this section, we introduce the key underlying modeling formalisms to the contributions of this thesis in subsequent chapters. We introduce the Discrete Event System Specification (DEVS), a system-theoretic simulation formalism for DES; Z notations and its Object-Oriented extension Object-Z, both of which are used to specify state-based systems for formal verification of properties by symbolic reasoning; and Temporal Logic, a logic-based language for formal specification dynamic system properties to be verified. We will use the beverage vending system as a running example throughout this section to illustrate each of the different formalisms.

3.2.1 The Beverage Vending System: A Running Example

This subsection presents the synopsis of a beverage vending system (BVS), which will be used as a running example in this chapter and subsequent chapters of this document.

As illustrated in Figure 3.1, BVS comprises two sub-systems, *Beverage Vending Machine* (BVM) and *User* (U), interacting with each other via their input and output interfaces to effect automated transactions. BVM is configured to dispense cocoa, coffee, orange and apple drinks to U at the cost of €1, 00, € 0, 80, €1,20, and €1,30 respectively.

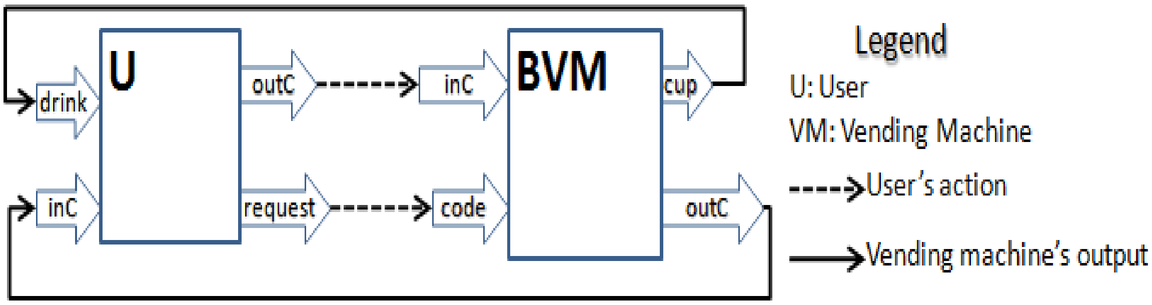


Figure 3.1 Schematic illustration of the beverage vending system

BVM has two input slots: a slot for receiving payments with coins and a keypad. The keypad has five code buttons numbered 1-5 for choosing drinks and canceling ongoing transactions; buttons 1, 2, 3 and 4 are used to choose cocoa, coffee, orange and apple respectively while button 5 cancels a transaction in progress. BVM recognizes only the following Euro coins for payment: € 0,10, € 0,20, € 0,50, € 1,00 and € 2,00; any coin outside this set is treated as "invalid" and rejected forthwith. BVM has two output slots: a slot for dispensing cups of drinks and a slot for returning coins to U; it returns coins in the form of balance for completed transactions, refund for canceled transactions or rejected coins.

U can receive two kinds of inputs from BVM: cups of drinks and coins as balance for completed transactions, as refund for canceled transactions or as rejected invalid coins. It also has two outputs: coin as payment for drinks and request to place an order for a drink or to cancel a transaction in progress.

U starts a transaction on BVM by choosing a drink (i.e., cocoa coffee, orange or apple) with the keypad; BVM responds to this request by prompting U to repeatedly insert acceptable coins, which are added to the *credit* (the cumulated amount of coins inserted for current transaction), until the *price* of the selected drink is reached or just exceeded. U takes a period of fifteen (15) seconds to pick a coin and insert it into the coin slot of BVM.BVM gives a maximum interval of two (2) minutes between the insertions of coins to fund a transaction; failure of which the transaction is automatically cancelled and the inserted coins (if any) are returned to U. U may also choose to cancel the transaction and immediately get a refund of all inserted coins before the termination of the *charging stage*.

Once *credit* is equal or greater than *price*, the *charging* stage terminates and the transaction cannot be canceled any more. This stage will be followed immediately by the *dispense stage* during which the requested drink is delivered to U and the inserted coins will be delivered to BVM's vault. If *credit* was greater than the price of the selected drink, the *dispense stage* is preceded by a *return stage* in which the balance will also be taken form vault and returned to U in a moment. BVM lasts for a non-interruptible period of one (1)minute in the *dispense stage* before delivering the requested cup of drink and transiting to the *idle* stage to wait indefinitely for the next user actions. U *waits* for a maximum period of ninety (90) seconds, after completing the payment, to receive an ordered drink; otherwise it requests to cancel the transaction. It also waits for the same period to receive the coins after canceling a transaction. After completing or canceling a transaction on BVM, U goes *away* for a random period in the range of 0-10 minutes before it develops the urge to take some drink and then goes back to initiate another transaction on BVM.

Design requirements for BVM

i. BVM must not dispense unless enough coins are inserted to pay for the selected drink

ii. BVM should always refund the balance whenever excess coins are inserted, i.e., when the amount inserted is greater than the price of the selected drink.

iii. Once the payment for a drink is complete, the transaction cannot be canceled any longer

We use this as a running example, in this thesis. It is used first in this chapter to illustrate system specification with DEVS simulation formalism (Section 3.2.3), Z notations (Section 3.2.4), Object-Z (Section 3.2.5) and Temporal Logic (Section 3.2.6). We also use it in Chapter 5 and Chapter 6as a case study for the enactment framework and system specification with HiLLS respectively.

3.2.2 Discrete Event System Specification (DEVS)

DEVS[Zei76, ZPK00]is a system-theoretic mathematical formalism for specifying DESs as abstract mathematical objects for simulation. It supports the specification of a full range of DESs as other formalisms for systems in this category have been proven to have equivalent DEVS representations [Van00].It, however, does not provide any concrete syntax to express system constructs; we can take advantage of this freedom by providing concrete specifications in a DSL with formal semantics that adopt the DEVS simulation protocol as its operational semantics.

DEVS defines two kinds of models -*atomic* and *coupled* models - for modular and hierarchical construction of system models. While a DEVS atomic model (AM) describes the structure of the so-called smallest unit of an autonomous DES and its behavior based on states and transition functions, aDEVS coupled model (CM)is an hierarchical composition of atomic and/or coupled DEVS models to construct more complex systems as illustrated in Figure 3.2. As shown in the diagram, the set of components of a CM may comprise only AMs or mixtures of AMs and CMs; for instance, CM2 has three components, CM1, AM4 and AM5, while CM1 itself composes AM1, AM2 and AM3. Components of a CM treat one another as black boxes with input and/or output ports through which they influence one another by exchanging events. This exchange of events between elements of a CM is facilitated by the *couplings* between their ports.

A coupling in CM can be one of three kinds: Input Coupling (IC), External Input Coupling (EIC) and External Output Coupling (EOC). As Figure 3.2 depicts, an IC is a coupling from an output port of a component to an input port of a peer component of the same CM; an EIC is a coupling from an input port of a CM to an input port of one of its components and an EOC is a coupling from an output port of a component of a CM and an output port of the CM itself. It is important to note, however, that a "self-loop coupling" is not allowed in DEVS; i.e., it is illegal to have a coupling between the input and output ports of the same system.

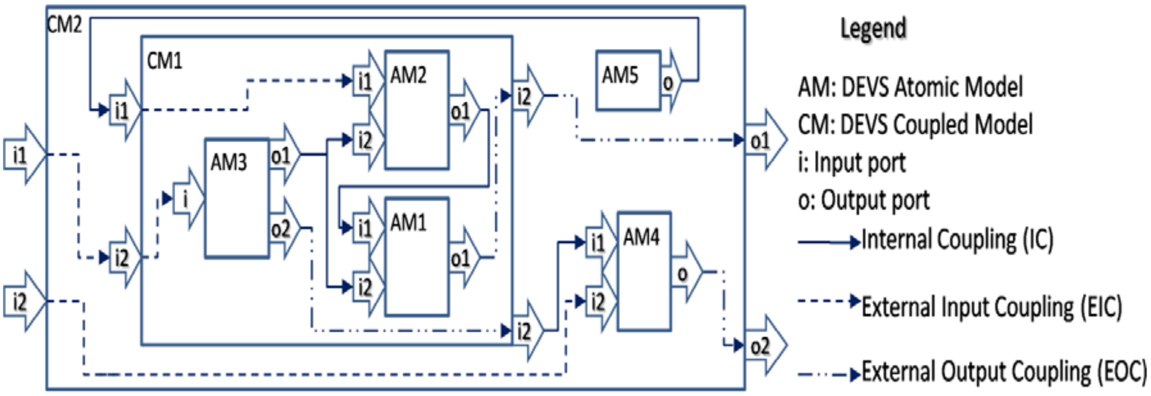


Figure 3.2Schematic illustration of hierarchical description of systems with DEVS

Conventionally, DEVS exists in two major forms - *classic DEVS* (CDEVS) and *parallel DEVS* (PDEVS) - the main difference being the support for concurrent state transitions within imminent components of a CM. CDEVS was defined by Zeigler in the mid-seventies [Zei76]. Note that every component of a CM is autonomous. i.e., its internal state evolves independently. However, due to the limitation of computing systems to sequential executions at that time, Zeigler's definition of CDEVS provides a "tiebreaker" function with which to set the priorities (sequence)to be followed whenever two or more components of a CM have imminent internal state transitions since the computing system would not be able to process them concurrently. With the support for concurrent and parallel computation in modern computing, PDEVS [CZ94, ZPK00] has been defined, which supports concurrent state transitions in imminent components of a CM. We will use the PDEVS throughout this thesis and refer to it simply as DEVS. Next, we present the mathematical descriptions of AM and CM and then use them to model the running example as an illustration.

3.2.2.1 DEVS atomic model (AM)

An atomic DEVS model, AM, has a time base and abstract sets of states, transitions, inputs and outputs to describe system's structure and behaviour. Mathematically, AM is defined as:

$$AM = < X, Y, S, \delta_{int}, \delta_{ext}, \delta_{conf}, \lambda, ta > \quad (1)$$

$X = \{(p, v), p \in IPort \wedge v \in dom(p)\}$ is the set of input events where each event is a port-value pair (p, v) such that every value $v$received is addressed to a port p in the set $IPort$ of input ports. A port p is nothing but an abstract variable whose domain $dom(p)$ defines the set of allowable input values; thus, every input v received on port p must respect the constraint $v \in dom(p)$. Similarly, $Y = \{(q, v), q \in OPort \wedge v \in dom(q)\}$ is the set of output events with $OPort$ as the set of output ports.

S is an abstract set of sequential states. At every instant, AM is in a state $s \in S$.

$ta: S \rightarrow \mathbb{R}^{+}_{0,\infty}$ is the time advance function which assigns a "time advance" to every state of the system. $\forall s \in S$, the time advance of s (i.e.,$ta(s)$) is the maximum period for which AM can remain in state s without external impulse before a scheduled state transition occurs. The time advance of a state can take real values in the range $0 \leq ta(s) \leq +\infty$. Based on the value of its time advance, a state is classified into one of three categories according to the following pars: (transient state, $ta(s) = 0$), (active state, $0 < ta(s) < +\infty$) and (passive state, $ta(s) = +\infty$).

$\delta_{int}: S \rightarrow S$ is the internal state transition function. When the system stays in a state s for a period equal to $ta(s)$ (the time advance of s) without receiving any external event, the system outputs the value $v = \lambda(s)$ and changes to state $s^{'} = \delta_{int}(s)$.

$\delta_{ext}: \{(s, e) | s \in S, e \in [0, ta(s)]\} \times X^{b} \rightarrow S$ is the external state transition function. The pair (s, e) is called the total state of the system at any instant where $0 \leq e < ta(s)$ is the elapsed time since the last state transition. If an external event $x \in X^{b}$is received while $e < ta(s)$, an external state transition is triggered which transits the system to a state$s^{'} = \delta_{ext}(s, e, x)$.

$\delta_{conf}: S \times X^{b} \rightarrow S$ is the confluent state transition function, which defines the system's behavior when the conditions for both internal and external state transitions are satisfied in coincidence. i.e., when an external event $x \in X^{b}$ is received at exactly $e = ta(s)$. In this situation, the system outputs the value $v = \lambda(s)$ and changes to state $s^{'} = \delta_{conf}(s)$.The superscript *b* on *X* signifies a *bag* of input values.

$\lambda: S \rightarrow Y^{b}$ is the output function that defines the value that may be produced as output when the system is in specific states. The superscript *b* on *Y* signifies a *bag* of output values. Note that, in DEVS, outputs are only allowed just before internal or confluent transitions.

3.2.2.2 DEVS coupled model (CM)

Mathematically, CM is defined as:

$$CM = < X, Y, D, \{M_d\}_{d \in D}, EIC, EOC, IC > \qquad (2)$$

X and Y are as defined for AM.

D is the set of references to the component of CM; $\forall d \in D$, M_d is the full specification referenced by d.In Figure 3.2 for instance, for CM1, set $D_{CM1} = \{AM1,\ AM2,\ AM3\}$and for CM2, $D_{CM2} = \{CM1,\ AM4, AM5\}$.

$EIC = \{((CM, ip_{CM}), (d, ip_d)) |\ ip_{CM} \in IPorts_{CM}, d \in D, ip_d \in IPorts_d\}$ is the set of External Input Couplings.

$EOC = \{((d, op_d), (CM, op_{CM})) |\ op_{CM} \in OPorts_{CM}, d \in D, ip_d \in OPorts_d\}$ is the set of External Output Couplings.

$IC = \left\{\left(\left(a, op_a\right), (b, ip_b)\right) | a, b \in D\ \wedge a \neq b, op_a \in OPorts_a\,, ip_b \in IPorts_b\right\}$ is theset of internal couplings.

The essence of the couplings is to allow for interactions between system components. Given an element $((S, p_S), (R, p_R))$ of any of the relations EIC, $EOC$and IC; the sender Sinfluences the receiver R by sending a message (event) through port p_S of S to port $p_R$of R. It is importantto note here that *CM* defines a logical boundary between all its components and the environment; therefore, any of the components can only influence (or be influenced by) the environment through the interfaces of *CM*, hence the need for$EIC$and EOC. Full details on DEVS and its operational semantics can be found in [ZPK00].

3.2.3 DEVS specification of the beverage vending system

In this subsection, we present the DEVS specification of the running example, which has been introduced in Section 3.2.1, to demonstrate system specification with DEVS. We see from the illustration in Figure 3.1that the beverage vending system can be studied as consisting of two subsystems interacting with each other: the beverage vending machine itself and the user. We can specify this in DEVS as a coupled model, *Beverage vending system* (BVS), comprising two atomic model components *Beverage vending machine* (BVM) and *User* (U). We will discuss the specification in a bottom-up fashion by first presenting the atomic models (i.e., BVM and U), and then their composition in BVS.

3.2.3.1 Beverage vending machine (BVM): a DEVS atomic model

Following the definition of DEVS atomic model in Section 3.2.2.1, we define BVM as follows:

$$BVM = < X_{BVM}, Y_{BVM}, S, \delta_{int}\,, \delta_{ext}\,, \delta_{conf}\,, \lambda, ta >$$

The elements of the mathematical structure are defined as follows:

A. Input interface

$$X_{BVM} = \{(inC, \mathbb{C}), (code, \{1,2,3,4,5\})\}.$$

BVM has two input ports as shown where $\mathbb{C} = \{1, 2, 5, 10, 20, 50, 100, 200\}$denotes the set of all Euro coins (values in cents). Then the input port inC (short hand for input coin) receives a coin, one at a time, from outside BVM. Input port $code$ describes the keypad; hence, its domain is modeled by the set {1,2,3,4,5} to receive requests for the different options described earlier. i.e., $code\ \in \{1, 2, 3, 4\}$to order beverages and $code\ \in \{5\}$to cancel an ongoing transaction.

B. Output interface

$$Y_{BVM} = \{(cup, \{cocoa, coffee, orange, apple\}), (outC, \mathbb{C}^b)\}$$

Set $\{cocoa, coffee, orange, apple\}$ describes the domain of output port cup to indicate the kinds of drinks dispensed by BVM. Let $\mathbb{C}^{\mathrm{b}}$ denote a "bag" of coins, then the output port $outC$ (short hand for output coins) allows BVM to return any number of coins (including identical coins) to the U.

C. State set

$$S = \{(vault, \mathbb{C}^b), (escrow, \mathbb{C}^b), (credit, \mathbb{N}), (badC, \mathbb{C}), (price, \{0, 100, 80, 120, 130\}), \\ (current, \{0, 1, 2, 3, 4, 5\}), (\phi, \{idle, charge, dispense, cancel, return, reject\})|$$

$$\phi = \begin{cases} idle, & if\ current = 0 \land credit = price = 0 \land badC = null \land escrow = [\![\ ]\!] \\ charge, & if\ current \in \{1, 2, 3, 4\} \land\ price > 0 \land credit < price \land badC = null \\ dispense, & if\ current \in \{1, 2, 3, 4\} \land\ price > 0 \land credit =\ price \land badC = null \\ & \qquad \land\, escrow = [\![\ ]\!] \\ return, & if\ current \in \{1, 2, 3, 4\} \land\ price > 0 \land credit >\ price \land badC = null \\ & \qquad \land\, escrow = [\![\ ]\!] \\ cancel, & if\ current \in \{5\}\ \land\ credit < price \\ reject, & if\ current \in \{1, 2, 3, 4\} \land badC\ \neq null \end{cases} \}$$

The state set S defines the system's state space. Technically, the system's state at any instant is determined by the combination of the values of its state variables. Table 3.1 describes the information held by the state variables defined in *S*.

Design Assumption: we assume the machine always has unlimited stock of drinks, hence our model abstracts away from the level of stock, and particularly, a state of being out of stock.

Table 3.1 State variables in DEVS specification of the BVM

State variable	Domain	Information held
$vault$	$\mathbb{C}^b$	A *bag* of coins accumulated in the safe of the machine from previous transactions. We use "bag" instead of "set" of coins to allow for the storage of identical coins in the vault.

$escrow$	$\mathbb{C}^b$	A *bag* of coins temporarily holding the coins accumulated for a transaction in progress. *Escrow* is a bag for the same reason as *vault*. It is emptied into the *vault* whenever a transaction runs to completion; if the transaction is cancelled, it will be emptied into the output port $outC$.
$credit$	$\mathbb{N}$	Cumulative value of coins received for the current transaction.
bad$C$	$\mathbb{C}$	A temporary holder for an "invalid/non-acceptable" coin before it is sent out of BVM. Let $\nu = \{10, 20, 50, 100, 200\}$ be the subset of $\mathbb{C}$ that contains the set of valid coins in BVM. Thus, a coin *c* is valid if $c \in \nu$; otherwise, it is invalid.
$price$	$\{0, 100, 80, 120,130\}$	Price of selected drink (in Euro cents). Recall from Section 3.2.1 that cocoa, coffee, orange and apple drinks cost €1,00, €0,80, €1,20, and €1,30 respectively. $price = 0$ when no transaction is in progress.
$current$	$\{0, 1, 2, 3, 4, 5\}$	Code for selected drink in current transaction. 1, 2, 3, 4 represent *cocoa*, *coffee*, *orange* and *apple* respectively; *current* = 0 is the default value when no transaction is in progress.
ϕ	$\{idle, charge, dispense, return, cancel, reject\}$	Identifiers of instantaneous states of the system. ϕ is indeed a derived variable whose values are pointers to sets of (ranges of) values of the other state variables. The corresponding predicates on the state variables for the different values of ϕ are as defined in the predicate part of set S above. *idle* =>BVM is idle. *charge* =>BVM is accepting and cumulating coins for current transaction. *dispense* =>BVM has cumulated enough credit for the requested drink and is now dispensing. *return* => returning the balance of a transaction when accumulated credit is greater than the price of the selected drink. *cancel* => canceling ongoing transaction to refund cumulated credit. *reject* => returning an invalid coin to the user.

D. Time advance function

$$ta(\phi) = \begin{cases} +\infty, & if\ \phi = idle \\ 2.0, & if\ \phi = charge \\ 1.0, & if\ \phi = dispense \\ 0.0, & if\ \phi = return \\ 0.0, & if\ \phi = cancel \\ 0.0, & if\ \phi = reject \end{cases}$$

The time advance function, $ta(\phi)$, defines the maximum periods for which BVM may stay in state ϕ before firing a scheduled internal transition. Recall from the synopsis of the system in Section 3.2.13.2.1 above that BVM stays in an idle state indefinitely in waiting for an initiation of a transaction on the keypad; therefore$ta(idle) = +\infty$. When a drink is selected, it gives the user a maximum of 2 minutes to insert enough coins to complete the transaction; this is modeled as the *charge* state with $ta(charge) = 2.0$. Once enough credit is received, BVM enters the *dispense* state to prepare and deliver a cup of drink in 1 minute; hence $ta(dispense) = 1.0$.If the received *credit* is greater than the *price* of the requested drink, then BVM enters the transient state *return*, prior to entering *dispense*, to return the balance of the transaction to the user; $ta(return) = 0.0$. We consider that the cancelation of transactions and rejection of invalid coins are instantaneous events; hence, we model their respective states as cancel and reject with $ta(cancel) = ta(reject) = 0.0$.

E. Internal state transition function

Before we specify the internal transition functions, let us define two mathematical functions, $bal: \mathbb{N} \longrightarrow \mathbb{N}$and $\kappa: \{n: \mathbb{N} | n \geq 0\} \longrightarrow \mathbb{C}^b$, which will be used in the specifications. Given a number $m \in \mathbb{N} bal(m) = m - price$returns the excess of m over the instantaneous value of state variable $price$. Given a natural number $n \geq 0$, function $\kappa(n)$returns a bag of coins whose total numerical (in cents) value is $n.\kappa$ generates a bag of coins for its numerical argument according to the recursive definition below:

$$\kappa(n: \mathbb{N} | n \geq 0.00) = \begin{cases} [\![200]\!] \cup \kappa(n - 200), & if\ n \geq 200 \\ [\![100]\!] \cup \kappa(n - 100), & if\ 100 \leq n < 200 \\ [\![50]\!] \cup \kappa(n - 50), & if\ 50 \leq n < 100 \\ [\![20]\!] \cup \kappa(n - 20), & if\ 20 \leq n < 50 \\ [\![10]\!] \cup \kappa(n - 10), & if\ 10 \leq n < 20 \\ [\![5]\!] \cup \kappa(n - 5), & if\ 5 \leq n < 10 \\ [\![2]\!] \cup \kappa(n - 2), & if\ 2 \leq n < 5 \\ [\![1]\!], & if\ n = 1 \\ [\![\ ]\!], & if\ n = 0 \end{cases}$$

For clarity purpose, since DEVS does not prescribe any definite format for presenting different elements of a specification, we will be presenting state transition behaviors in the format:

$\delta\ \ (\phi_{source}) = \phi_{target} : \langle op_1, op_2, .., op_n\rangle$ such that δ can be δ_{int}, δ_{ext} or δ_{con}

ϕ_{source} and ϕ_{target} are the source and target states respectively.$\langle op_1, op_2, .., op_n\rangle$ is the finite sequence of reconfiguration operations on the state variables that result in the transition from the source state to target state.

The internal state transition function $\delta_{int}(\phi)$of BVM is defined as:

$$\delta_{int}(\phi) = \begin{cases} cancel\colon \langle current = 5\rangle, & if\ \phi = charge \\ idle\colon \langle current = 0, price = 0, credit = 0, escrow = [\![\ ]\!], badC = null\ \rangle, & if\ \phi = cancel \\ charge\colon \langle badC = null\rangle, & if\ \phi = reject \\ idle\colon \langle current = 0, price = 0, credit = 0\rangle, & if\ \phi = dispense \\ dispense\colon \langle credit - bal(credit), vault \setminus \kappa \circ bal(credit)\rangle, & if\ \phi = return \end{cases}$$

$\delta_{int}(\phi)$ defines the system's behavior when it has stayed in state$\phi$for a period $e = ta(\phi)$ without receiving any external event. The function defines five cases of internal state transition behaviors for BVM:

Case $\phi = charge$specifies a sequence of reconfiguration of state variable *current* (i.e.,$\langle current = 5\rangle$) leading to an automatic transition to state$\phi = cancel$.

Case $\phi = cancel$specifies transition to state $\phi = idle$following the reset of all state variables to their default values in $\langle current = 0, price = 0, credit = 0, escrow = [\![\ ]\!], badC = null\ \rangle$.

Case $\phi = reject$specifies a reconfiguration of state variable *badC* (i.e., $\langle badC = 0\rangle$) leading to an automatic transition to state$\phi = charge$.

Case $\phi = dispense$ specifies the system's transition behavior at the successful processing of a request for a drink. The transition to state $\phi = idle$follows the sequence of reconfiguration of state variables specified in $\langle current = 0, price = 0, credit = 0, escrow = [\![\ ]\!]\rangle$.

Case $\phi = return$specifies the system's transition behavior preceding the *dispense* state when the accumulated credit is greater than the price of the selected drink. As specified if the sequence of reconfiguration of state variables,$\langle vault \setminus \kappa \circ bal(credit), credit - bal(credit)\rangle$, the balance of the transaction is deducted from the accumulated credit and a bag of coins equivalent to the balance is withdrawn from the *vault*. The composite operation $\kappa \circ bal(credit)$ computes the bag of coins whose total value is equivalent to the excess of *credit* over *price*. The delivery of this bag of coins to the user will be defined later in the λ function.

Since $ta(idle) = +\infty$, which will never elapse, there is no internal transition from state $\phi = idle$.

F. External state transition function

Prior to the presentation of the external transition function, let us define a mathematical function $\rho\colon CODE \longrightarrow PRICE$, which sets the price of a transaction based on the drink code selected on the keypad. $CODE = \{1, 2, 3{,}4\}$ and $PRICE = \{0, 100, 80, 120{,}130\}$are the sets of keypad codes and prices respectively of cocoa, coffee, orange and apple drinks. The function is defined as:

$$\rho(code \in \{1, 2, 3, 4\}) = \begin{cases} price = 100, & if\ code = 1\ //cocoa \\ price = 80, & if\ code = 2\ //coffee \\ price = 120, & if\ code = 3\ //orange \\ price = 130, & if\ code = 4\ //apple \end{cases}$$

We present the external transitions in the format:

$$\delta_{ext}\,(\phi_{source}\,, e, (p, in)) = \phi_{target} : \langle op_1, op_2, .., op_n \rangle$$

Where $e < ta(\phi_{source}\,)$ is the time elapsed since transition to the source state and the pair (p, in) is the input event that triggers the transition when input value *in* is received on input port p. Thus, $in \in dom(p)$ as defined previously in the input set of a DEVS model in Section 3.2.2.1.

The external state transition function of BVM is defined as:

$$\delta_{ext}\,(\phi, e, (p, in)) \\ = \begin{cases} charge\colon \langle current = in, \rho(in), credit = 0\rangle, if\ \phi = idle \\ \qquad \wedge\, p = code \wedge in \in \{1, 2, 3, 4\} \\ charge\colon \langle escrow \cup [\![in]\!], credit + in\rangle, if\ \phi = charge \\ \qquad \wedge\, p = inC \wedge in \in \nu \wedge credit + in < price \\ dispense\colon \langle vault \cup escrow \cup [\![in]\!], escrow = [\![\ ]\!], credit + in\rangle, if\ \phi = charge \\ \qquad \wedge\, p = inC \wedge in \in \nu \wedge credit + in = price \\ return\colon \langle vault \cup escrow \cup [\![in]\!], escrow = [\![\ ]\!], credit + in\rangle, if\ \phi = charge \\ \qquad \wedge\, p = inC \wedge in \in \nu \wedge credit + in > price \\ reject\colon \langle badC\ = in\rangle, if\ \phi = charge \\ \qquad \wedge\, p = inC \wedge in \notin \nu \\ cancel\colon \langle current = in\rangle, if\ \phi = charge \\ \qquad \wedge\, p = code \wedge in = 5 \end{cases}$$

The function defines one case (resp. five cases) of external transition behavior from state $\phi = idle$ (resp. $\phi = charge$).

Case $\phi = idle \wedge p = code \wedge in \in \{1, 2, 3, 4\}$ specifies a transition to state $\phi = charge$ upon receiving an input $in \in \{1, 2, 3, 4\}$ on input port $code$ while the system is in state $\phi = idle$. The operation $\rho(in)$ sets the value of state variable *price*, as defined earlier, based on the input value.

Case $\phi = charge \wedge p = inC \wedge in \in \nu \wedge credit + in < price$ specifies a transition back to state $\phi = charge$ if an "acceptable" coin $in \in \nu$is received on input port inC while the system is in state $\phi = charge$such that the sum of the value of the received coin and the subsisting value of *credit* is less than *price*.

Case $\phi = charge \wedge p = inC \wedge in \in v \wedge credit + in = price$ is similar to the previous case except that $credit + in = price$. This leads to a transition to state $\phi = dispense$ while the received coin *in* and the content of *escrow* are transferred to the *vault*.

Case $\phi = charge \wedge p = inC \wedge in \in v \wedge credit + in > price$ is also similar to the two previous cases with the difference being that$credit + in > price$. This is followed by a transition to state $\phi = return$ in which the balance of the transaction will be returned to the user.

Case $\phi = charge \wedge p = inC \wedge in \notin v$defines the system's behavior when it receives an "invalid/non-acceptable" coin $in \notin v$on input port inCwhile in state $\phi = charge$. This results into a transition to state $\phi = reject$ in which the received coin will be returned, forthwith, to the user.

Lastly, case $\phi = charge \wedge p = code \wedge in = 5$ specifies a transition to state $\phi = cancel$if an input $in = 5$ is received on input port $code$ while the system is in state $\phi = charge$.

G. Confluent transition function

The confluent transition behaviors will be specified in the format:

$$\delta_{conf}\,(\phi_{source}\,, (p, in)) = \phi_{target} : \langle op_1, op_2, .., op_n \rangle$$

All variables are as described previously.

$$\delta_{conf}\,(\phi, (p, in)) = \begin{cases} dispense\colon \langle credit + in, vault \cup escrow \cup [\![in]\!], escrow = [\![\ ]\!] \rangle, if\ \phi = charge\ \wedge p = inC \\ \qquad \wedge\, in \in v \wedge credit + in = price \\ return\colon \langle credit + in, vault \cup escrow \cup [\![in]\!], escrow = [\![\ ]\!] \rangle, if\ \phi = charge\ \wedge p = inC \\ \qquad \wedge\, in \in v \wedge credit + in > price \\ charge\colon \langle credit + in, escrow \cup [\![in]\!] \rangle, if\ \phi = charge\ \wedge p = inC \\ \qquad \wedge\, in \in v \wedge credit + in < price \\ cancel\colon \langle badC = in, current = 5 \rangle, if\ \phi = charge\ \wedge p = inC \\ \qquad \wedge\, in \notin v \end{cases}$$

As specified in the function above, there are four cases of confluent state transition behavior defined for BVM, each of which defines a state transition that occurs when an input event *in* is received on input port p at exactly when the system has been in state ϕ for a period $e = ta(\phi)$. The different cases can be read in the same manner as those explained earlier for external transitions δ_{ext}.

H. Output function

Prior to specifying the output function, let us define a mathematical function $\psi\colon \{1, 2, 3, 4\} \longrightarrow \{cocoa, coffee, orange, apple\}$, which generates a cup of drink based on the value of a variable *x*. Recall that current holds the drink code selected from the set $\{1, 2, 3, 4\}$ at the beginning of the transaction. Hence ψmaps a code number to a cup of drink as follows:

$$\psi(x \in \{1, 2, 3, 4\}) = \begin{cases} cocoa, & if\ x = 1 \\ coffee, & if\ x = 2 \\ orange, & if\ x = 3 \\ apple, & if\ x = 4 \end{cases}$$

The output function λ is defined as:

$$\lambda(\phi) = \begin{cases} cup = \ \psi(current), & if \phi = dispense \\ outC = \kappa \circ bal(credit), & if \phi = return \\ outC = [\![badC]\!], & if \phi = reject \\ outC = escrow \cup [\![badC]\!], & if \phi = cancel \wedge \\ \qquad (escrow \neq [\![\ ]\!] \vee badC \neq null) & \\ \{\}, \quad otherwise & \end{cases}$$

The function $\lambda(\phi)$defines five cases of output events that occur just before internal and/or external state transitions from stateϕ.

Case $\phi = dispense$ defines the output event that occurs just before leaving the *dispense* state. In this case a cup of the requested drink, $\psi(current)$, is produced as output on the *cup* port.

Case $\phi = return$defines the output event that occurs just before leaving the *return* state. In this case, the composite operation $\kappa \circ bal(credit)$ computes the bag of coins whose total value is equivalent to the excess of *credit* over *price*. The bag of coins so generated is produced as output on port *outC*.

Case $\phi = reject$states that before leaving the *reject* state, the rejected invalid coin, *badC,* is sent out on port *outC*.

Case$\phi = cancel \wedge (escrow \neq [\![\ ]\!] \vee badC \neq null)$is a composition of three cases associated with state$\phi = cancel$:$escrow \neq [\![\ ]\!] \wedge badC = null$, $escrow = [\![\ ]\!] \wedge badC \neq null$and $escrow \neq [\![\ ]\!] \wedge badC \neq null$. In any of the cases, all coins that may be contained in either *escrow* or *badC* or both are sent out as output on port *outC*.

3.2.3.2 Beverage vending machine user (U): a DEVS atomic model

The DEVS atomic model of the user, U, is defined as:

$$U = < X, Y, S, \delta_{int}, \delta_{ext}, \delta_{conf}, \lambda, ta >$$

We describe the elements of the structure in a similar way with BVM

A. Input interface

Let *Beverage* be the set of cups of all kinds of beverage drink including cocoa, coffee, orange and apple.

$X_U = \{(drink, Beverage),\ (inC, \mathbb{C}^{\mathrm{b}})\}$.

U has two input ports as shown. Since the set *Beverage*, the domain of port *drink* subsumes all kinds of drinks, U can receive cups of cocoa, coffee, orange, apple and many more. U can also receive any kind of coin on port *inC*.

B. Output interface

$Y_U = \{(request, \{1,2,3,4,5\}),\ (outC, \mathbb{C})\}$.

U has two output ports *request* and *outC*. The domain of port *request*models the choices that U could make during a transaction; while outputs of numbers 1-4 on port *request* correspond to the choices of drinks as described previously, an output of 5 signifies the choice to cancel an ongoing transaction. Port *outC* models the coins that can be sent out by U.

C. State set

$$S = \{(wallet, \mathbb{C}^b), (bill, \mathbb{N}), (advance, \mathbb{N}), (cup, Beverage), (purse, \mathbb{C}^b), (choice, \{0..5\}), \quad (\phi, \{away, inserting, ordering, canceling, waiting\})|$$

$$\phi = \begin{cases} away, & if\ bill = advance = choice = 0 \\ ordering, & if\ choice \in \{1,2,3,4\} \wedge bill = advance = 0 \wedge cup = null \\ inserting, & if\ choice \in \{1,2,3,4\} \wedge 0 \le advance < bill \wedge cup = null \\ waiting, & if\ (choice \in \{1,2,3,4\} \wedge 0 < bill \le advance) \\ & \quad \vee\ (choice = 0 \wedge advance < bill) \\ canceling, & if\ choice \in \{5\} \end{cases} \}$$

The purpose of each state variable is briefly explained in Table 3.2below.

Table 3.2 DEVS state variables of beverage vending machine user

State variable	Domain	Information held
$bill$	$\mathbb{N}$	The bill (in cents) of the current transaction. Its value is set based on the chosen drink (i.e., based on the value of variable *choice*).Recall from Section 3.2.1 that cocoa, coffee, orange and apple drinks cost €1,00, €0,80, €1,20, and €1,30 respectively. Let us define a mathematical function $\beta: \{0,1,2,3,4,5\} \longrightarrow \mathbb{R}$that sets the value of *bill* as:

		$\beta(choice \in \{0,1,2,3,4,5\}) = \begin{cases} 100, & if\ choice = 1 \\ 80, & if\ choice = 2 \\ 120, & if\ choice = 3 \\ 130, & if\ choice = 4 \\ 0, & otherwise \end{cases}$
$choice$	$\{0, 1, 2, 3, 4, 5\}$	Code to choose transaction. 1, 2, 3, 4 represent *cocoa*, *coffee*, *orange* and *apple* respectively. *current* = 0 is the default value when no transaction is in progress.
$advance$	$\mathbb{N}$	Cumulative value of the coins expended for a transaction. Unlike BVM, U does not discriminate against any kind of coin.
$wallet$	$\mathbb{C}^b$	A finite *bag* of coins from which U makes transaction on BVM and probably elsewhere.
$purse$	$\mathbb{C}^b$	A finite *bag* of coins to store coins rejected by BVM. Technically, these are coins (1 cent, 2 cents, and 5 cents) returned while payment for an ongoing transaction is in progress.
cup	$\{cocoa, cofee, orange, apple, null\}$	A variable to hold the cups of drinks that may be received at the end of the transactions. The domain of the variable denotes the kinds of expected drinks. It has a *null* value when no cup has been received.
ϕ	$\{away, ordering, inserting, waiting, canceling\}$	Identifiers of instantaneous states of the system. ϕ is indeed a derived variable whose values are pointers to sets of (ranges of) values of the other state variables. The corresponding predicates on the state variables for the different values of ϕ are defined in the predicate part of set S above. The states may be interpreted as follows: *away* => no transaction in progress, it leaves this state only when it has the urge to take some drink. *ordering* => U is placing an order for a drink on BVM. *inserting* => U is inserting coins into BVM to pay an initiated transaction. *waiting* => U is waiting to receive an order after completing the payment or to receive a refund of advance payments for a canceled transaction. *canceling* => U is canceling an initiated order before completing the payment.

D. Time advance function

The time advance function is specified as:

$$ta(\phi) = \begin{cases} rand[0.0, 100.0], & if\ \phi = away \\ 0.0, & if\ \phi = ordering \\ 0.25, & if\ \phi = inserting \\ 1.5, & if\ \phi = waiting \\ 0.0, & if\ \phi = canceling \end{cases}$$

Recall that U remains in state$\phi = away$ until it has the urge to take some drink; we consider that this urge is an internal event for which frequency we cannot precisely define (or at least, it would be difficult to define the frequency precisely). For the purpose of our discussion in this example, let us take the frequency to be a random value in the range $[0, 100]$minutes. Hence, $ta(away) = rand[0.0, 100.0]$. Placing an order and canceling a transaction in this system are instantaneous events as they involve just pressing a button, hence $ta(ordering) = ta(canceling) = 0.0$. It takes 15 seconds (0.25 minutes) to pick a coin from the wallet and insert it in BVM, so $ta(canceling) = 0.25$. Lastly, U waits for a maximum of 90 seconds to take delivery of a drink or get refund of coins from BVM, hence, $ta(waiting) = 1.5$.

E. Internal state transition function

Prior to the external state transition function, let us define two mathematical functions that will be used in the specification of the transition functions:

First, let $\sigma: \mathbb{C}^b \longrightarrow \mathbb{R}$ be the function that computes the sum of the numerical values of all the coins in a given non-empty bag of coins.σ is defined as:

$$\sigma(C \in \mathbb{C}^b) = \sum_{i=1}^{i=|C|} C_i$$

Secondly, let function$\tau: \mathbb{C}^b \longrightarrow \mathbb{C}$denote the operation of randomly picking a coin$c \in \mathbb{C}$ from a non-empty bag of coins.

The internal state transition function of U is defined as follows:

$\delta_{int}: \phi \longrightarrow \phi$Where$\phi = \{away, inserting, canceling, waiting\}$

For clarity, we present the internal transition specifications from each of the seven states separately. The transitions will be presented in the same format as BVM:

$$\delta_{int}(\phi_{source}) = \phi_{target} : \langle op_1, op_2, .., op_n \rangle$$

ϕ_{source} and ϕ_{target} are the source and target states respectively while $\langle op_1, op_2, .., op_n \rangle$ is the sequence of reconfiguration operations on the state variables that result in the transition from the source state to target state.

The internal transitions from the different states are as follows:

Specification of internal transition from state $\phi = away$

$$\delta_{int}(away) = ordering{:}\,\langle choice = rand[1..4], cup = null\rangle$$

This specifies an internal transition from state $\phi = away$ to state $\phi = ordering$following the assignment of a randomly generated natural number in the interval [1..4] to state variable *choice* and *null* to state variable *cup*.

Specification of internal transition from state $\phi = ordering$

$$\delta_{int}(ordering) = inserting{:}\,\langle bill = \beta(choice)\rangle$$

This specifies an internal transition to state $\phi = inserting$ from $\phi = ordering$ following the assignment of a value to *bill* based on the instantaneous value of *choice*. Recall that we defined, in Table 3.2above, the function β, which assigns values to bill based on the price of the drink corresponding to the subsisting value of *choice*.

Specification of internal transition from state $\phi = inserting$

There are three cases of internal transition from state $\phi = inserting$ as specified below:

$$\delta_{int}(inserting) = \begin{cases} inserting{:}\,\langle advance + \tau(wallet)\rangle, & if\ wallet \neq [\![\ \ ]\!] \\ & \wedge\ advance + \tau(wallet) < bill \\ waiting{:}\,\langle advance + \tau(wallet)\rangle, & if\ wallet \neq [\![\ \ ]\!] \\ & \wedge\ advance + \tau(wallet) \geq bill \\ canceling{:}\,\langle choice = 5\rangle, & if\ wallet = [\![\ \ ]\!] \end{cases}$$

We have defined the function $\tau(wallet)$, which returns a coin picked at random from the *wallet*. The explanations of the three transition cases are as follows:

The first case specifies the behavior when *wallet* is not empty and the sum of the current value of *advance* and the numerical value of the coin picked from the *wallet* is still less than *bill*. This implies that the payment for the requested drink is not yet complete; since *wallet* is not empty, a transition back to state $\phi = inserting$occurs following the operation specified in the angle bracket.

The condition of the first case is similar to the first except that the sum of the current value of *advance* and the value of the coin picked from *wallet* is at least equal to *bill*. Hence, the target of the transition event is the state $\phi = waiting$ following the specified variable reconfiguration operations.

The third case specifies the behavior when *wallet* becomes empty while in state $\phi = inserting$. The implication of this situation is that there are no sufficient acceptable coins to pay the *bill* of the selected drink. Hence a transition to state $\phi = canceling$ occurs following the assignment operation $choice = 5$.

Specification of internal transition from state $\phi = canceling$

We specify two cases of transition behavior from state $\phi = canceling$ below:

$$\delta_{int}(canceling) = \begin{cases} waiting: \langle choice = 0 \rangle, & if advance > 0 \\ away: \langle choice = 0, \quad bill = 0 \rangle, & if advance = 0 \end{cases}$$

Case $advance > 0.00$ implies that some coins had already been released before entering the *canceling* state; hence, the target of the transition is the state $\phi = waiting$ to get a refund of the coins. Case $advance = 0.00$, however, implies that either no coin had been released so far or all released coins have been refunded (rejected), hence a transition to state $\phi = away$ occurs following the reset of other state variables to their default values.

Specification of internal transition from state $\phi = waiting$

$$\delta_{int}(waiting) = canceling: \langle choice = 5 \rangle$$

The only possible internal transition from state $\phi = waiting$ targets state $\phi = canceling$ following the assignment operation $choice = 5$

F. External transition function

Before we discuss the different cases, recall that function σdefines the operation of computing the cumulative numerical value of all the coins in a given bag of coins. Hence $\sigma(bag)$ is the operation of computing the total value of the coins stored in the *bag* variable at any instant.

The external transition function is specified as follows:

$$\delta_{ext}\big(\phi, e, (p, in)\big)$$
$$= \begin{cases} inserting: \langle purse \cup in, advance - \sigma(in) \rangle, & if\ \phi = inserting \wedge p = inC \\ waiting: \langle wallet \cup in, advance - \sigma(in) \ \rangle, & if\ \phi = waiting \wedge p = inC \\ & \wedge\ advance > 0 \\ away: \langle wallet \cup in, bill = 0, choice = 0 \ \rangle, & if\ \phi = waiting \wedge p = inC \\ & \wedge\ advance = 0 \\ away: \langle cup = in, bill = 0, choice = 0, advance = 0 \rangle, & if\ \phi = waiting \wedge p = drink \end{cases}$$

The external state transition function specifies the system's behavior when it receives an input event in(i.e., a trigger) on input port p while having being in state ϕ for a period $0 \leq e < ta(\phi)$.Four cases are specified in the external transition function above:

The receipt of a bag of coins on input port inCwhile the system is in state $\phi = inserting$triggers an external transition back to state $\phi = inserting$ while the bag of coins received is kept in the *purse* and its numerical value is deducted from *advance*; the coin received in this state is considered to have been rejected.

When a bag of coins is received in state$\phi = waiting$and $advance > 0$, an external state transition back to state $\phi = waiting$ occurs while the coins are returned to the *wallet*; this is considered the balance of a successful transaction, hence user returns to *waiting* to await the ordered item. However, if $advance = 0$, a transition to state $\phi = away$ occurs; it is considered, in this case, that *user* is not expecting anything more since *advance* =0.

If a cup of drink is received on input port *drink* while the system is in state $\phi = waiting$a transition to state $\phi = away$ occurs.

G. Confluent transition function

The confluent state transition function specifies the system's behavior when it receives an input event in (i.e., a trigger) on input port p while having being in state ϕ for a period of exactly $e = ta(\phi)$. Incidentally, this system has identical behaviors for external and confluent transition events.

H. Output function.

The output function specifies three cases of output events just before internal and/or confluent transitions from the states defined by ϕ.

$$\lambda(\phi) = \begin{cases} request = choice, & if\ \phi = ordering \vee \phi = canceling \\ outC\ = \tau(wallet), & if\ \phi = inserting \wedge advance < bill \wedge wallet \neq [\![\ ]\!] \\ \{\ \}, & otherwise \end{cases}$$

Case $\phi = ordering \vee \phi = canceling$ is a disjunction of two cases: $\phi = ordering$ and $\phi = canceling$. In each case, the instantaneous value of state variable *choice* is sent out as output port *request*. Recall that clause $choice \in \{1, 2, 3, 4\}$ (resp. $choice \in \{5\}$) is always true whenever the system is in state $\phi = ordering$ (resp. $\phi = canceling$). Hence, the output in the first case places an order for a drink while that of the second case requests a cancelation of an ongoing transaction.

Case $\phi = inserting \wedge advance < bill \wedge wallet \neq [\![\ ]\!]$specifies that a coin picked at random from *wallet* is produced as output on port $outC$ just before an internal or confluent transition from state $\phi = inserting$ on the condition that *advance* (i.e., total amount of "acceptable" coins released so far for the ongoing transaction) is less than *bill* (i.e., the price of the order) and *wallet* is not empty.

3.2.3.3 Beverage vending system (BVS): a DEVS coupled model

Based on the definition of DEVS coupled model, we describe BVS as:

$$BVS = < X, Y, D, \{M_d\}_{d \in D}, EIC, EOC, IC >$$

The elements of the structure are defined as follows:

- **Input and output interfaces**

$$X = \{\}; Y = \{\}$$

In BVS, exchanges of events exist only between its components, BVM and U. Since we do not consider interactions with anything outside the coupled model, BVS is a close system and thus has no input or output port.

- **Components**

The set of component references of BVS, $D = \{BVM, U\}.\forall d \in D$, M_d is the complete DEVS specification referred to by *d*. i.e., M_{BVM} and M_U are the DEVS models presented previously in Section 3.2.3.1 and Section 3.2.3.2 respectively.

- **Couplings**

$$IC = \{\big((U, request), (BVM, code)\big), \big((U, outC), (BVM, inC)\big), \big((BVM, cup), (U, drink)\big),$$

$$\big((BVM, outC), (U, inC)\big)\}; EIC = \{\}; EOC = \{\}$$

Recall that coupling set EIC (resp. EOC) is a relation involving the input (resp. output) ports of a coupled model. Thus for BVS, $EIC = \{\}$ and $EOC = \{\}$. There are, however, four internal couplings as specified in the set IC above. Each coupling in the set is a pair of pairs where the first pair specifies the sending system and the port through which the event is being sent while the second pair specifies the receiving system and the port through which the event is received. For instance, coupling $\big((U, request), (BVM, code)\big)$ specifies that the port *request* of U is coupled with the port *code* of BVM so that the former can send events to the latter. Other couplings in the set can be read similarly.

3.2.4 Z Language

The Z language/notation [Spi88, Spi92],pronounced as "Zed", is a formal specification language founded on set theory and predicate logic used for specifying software and hardware systems for logical analysis. Z is widely used by the FM community both in academia and in industry for its considerable universality in describing state-based systems and amenability to symbolic manipulations by diverse tools. Z is specially suited to model systems' data and state changes [SWA+05].

Our discussion of Z will be limited to what is required to follow the work in this thesis; for detailed presentation of the language, we invite the reader to consult any of its numerous texts (e.g., [Spi88, Spi92, WD96, Jac97, ISO02]).

One of the fundamental features of Z, which is of particular interest in this thesis, is that every element of a specification has a unique and precisely defined type. This makes Z specifications specifically amenable to type-checking and diverse logical reasoning; interestingly, there exist diverse tools that practically support such analyses; examples include UPAAL [BDL04], CZT [MU05, MFM+05],Z/EVES[MS97, Saa03], CadiZ [TM95], and Circus [WC01].

Z supports modular and hierarchical specification of complex state-based systems using schema calculus. A Z schema may be described as a collection of declarations and (optional) constraints to describe a mathematical object and its properties. It can be used to describe the state space of a system and the ways in which the state can possibly change, as well as to describe and reason about some static properties and possible refinements of the system. Essentially, a Z specification is made up of paragraphs; a Z paragraph can be a basic type definition, a free type definition, an axiomatic definition, or a schema. The next subsections elaborate a bit on these kinds of Z paragraphs and schema calculus, the composition of schemas with logical connectives for hierarchical construction of complex schemas.

3.2.4.1 Basic type definition

As its name implies, a basic type definition introduces one or more basic type(s). A basic type is a unique name, which has not been used previously for any global declaration in the same specification, used to name an abstract set without giving details of the objects it contains. It is specifically used when it does not matter, for the purpose of the specification, what form or structure is taken by the objects been represented.

The syntax of basic type definitions is as follows:

$$[\boldsymbol{IDENTIFIER_1,..., IDENTIFIER_n}]$$

Each "*IDENTIFIER*" is a basic unique name representing a basic type and its scope extends globally from the point of specification to the end of the specification. As an example, imagine we want to specify system that contains variables such as names of people and towns we may introduce the basic types **[*NAME, TOWN*]** to model the types of names of people and towns respectively. In this case, we consider the types to be infinite sets of names people and towns respectively and we are not interested in the forms taken by the names.

3.2.4.2 Free type definition

Free types are sets with explicit structuring information that can be used for the specification of a variety of data structures such as lists, arrays, or trees of elements drawn from one or more basic types [WD96].A free type is especially suitable to model enumerated collections, compound objects, and recursively defined structures.

The format of a free type can be described as$FREETYPE ::= consant_1|\ldots|constant_n$ or $FREETYPE ::= constant|constructor \ll source \gg$. The former can be used in particular to define a finite set of distinguishable elements while the latter is most suitable to define some recursive structures [DW96, Spi92]. The scope of a free type starts from the point of its definition to the end of the specification.

As examples of free type definitions, let us begin the Z specification of our Beverage Vending Machine (BVM) running example with the introduction of free types *COIN*, *CUP* and *CODE,* to represent the set of euro coins, set of cups of beverages and set of input codes respectively.

The types are defined as follows:

$$COIN ::= 1cent|\ 2cent|\ 5cent|\ 10cent|\ 20cent|\ 50cent|\ 1euro\ |\ 2euro$$

$$CUP ::= cocoa|coffee|orange\ |\ apple$$

$$CODE ::= 1\ |\ 2\ |\ 3\ |\ 4$$

Type *COIN* defines a set of eight distinguishable kinds of euro coins; any variable declared with this type henceforth can only take its value from this set. Similarly, types *CUP* and *CODE* define finite sets of distinguishable beverages and input codes respectively. Detailed descriptions, with examples, of the two formats of free type definition are provided in [DW98, p. 132-145].

3.2.4.3 Axiomatic definition

An axiomatic definition is used in a Z specification to introduce one or more global variable(s), constant(s) or function(s) possibly accompanied by constraints on their values. Just like a basic type definition, the variables declared in an axiomatic definition must be unique and their scopes extend from the points of declaration to the end of the specification. The predicates defining constraints in an axiomatic definition may define relationships between its variables and/or between its variables and other variables that have been defined previously in the specification. Such predicates can be regarded as global properties of the system [Spi92].

Figure 3.3below describes two possible templates for specifying axiomatic definitions. The format on the left side of the figure shows two segments separated by a horizontal dividing line (which can be interpreted to mean "such that"); while the upper segment contains a finite number of declarations of variables or constants, the predicates specifying constraints on them, if any, are specified in the lower segment. The template on the right depicts that the dividing line and the lower segment can be absent if no predicate is defined; the predicate in this case is the logical value *true* [Spi92].

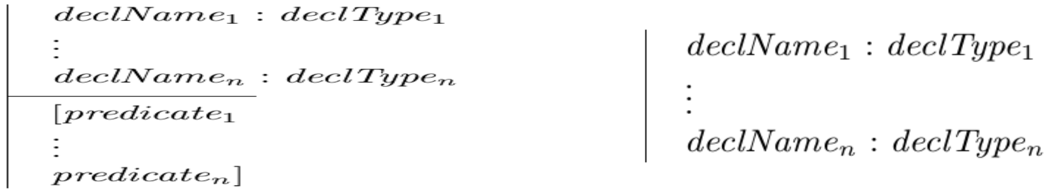

$declName_1 : declType_1$
⋮
$declName_n : declType_n$
$[predicate_1$
⋮
$predicate_n]$
$declName_1 : declType_1$
⋮
$declName_n : declType_n$

Figure 3.3Templates for Z axiomatic definition

Figure 3.4presents a sample axiomatic definition from the Z specification of the BVM, which reuses the free types *COIN*, *CUP* and *CODE* defined previously.v is an injective function which maps a given euro coin to its numerical value in cents; the value of each coin is precisely defined in the first two lines on the predicate part of the axiomatic definition. ρ is also an injective function that determines the price of a transaction based on the code chosen on the keypad while ψ injectively maps a number from the set {1, 2, 3, 4}, selected on the keypad, to the corresponding beverage. Function σ computes the sum of the numerical values of all coins in a given bag of coins. Function κ is the inverse of σ; it specifies the generation of a bag of coins whose total numerical value is equivalent to a given natural number. More details about each of the functions are provided in the predicate part of the axiomatic definition.

$$
\begin{array}{|l}
v : COIN \longrightarrow \{1, 2, 5, 10, 20, 50, 100, 200\} \\
\rho : CODE \longrightarrow \{100, 80, 120, 130, 0\} \\
\psi : \{1, 2, 3, 4\} \longrightarrow CUP \\
\sigma : [\![COIN]\!] \longrightarrow \mathbb{N} \\
\kappa : \mathbb{N} \twoheadrightarrow \mathbb{P}[\![COIN]\!] \\
\hline
v : 1cent \mapsto 1 \wedge v : 2cent \mapsto 2 \wedge v : 5cent \mapsto 5 \wedge v : 10cent \mapsto 10 \wedge \\
v : 20cent \mapsto 20 \wedge v : 50cent \mapsto 50 \wedge v : 1euro \mapsto 100 \wedge v : 2euro \mapsto 200 \\
\rho : 1 \mapsto 100 \wedge \rho : 2 \mapsto 80 \wedge \rho : 3 \mapsto 120 \wedge \rho : 4 \mapsto 130 \wedge \rho : 5 \mapsto 0 \\
\psi : 1 \mapsto cocoa \wedge \psi : 2 \mapsto coffee \wedge \psi : 3 \mapsto orange \wedge \psi : 4 \mapsto apple \\
\forall\, C \in [\![COIN]\!] \bullet \sigma(C) = \sum_{i=1}^{i=|C|} v(C_i) \\
\forall (n, C) \in \kappa \bullet \sigma(C) = n
\end{array}
$$

Figure 3.4 A sample axiomatic definition in the BVM specification

3.2.4.4 State schema

The state schema defines the state space of a system by declaring its state variables and the constraints on their values, if any. These constraints are also called the *state invariants* as they are relationships must always remain valid in all states of the system.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\_ StateSchemaName \_\_\_\_} \\
declName_1 : declType_1 \\
\vdots \\
declName_n : declType_n \\
\hline
[predicate_1 \\
\vdots \\
predicate_n] \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{\_\_ StateSchemaName \_\_\_\_} \\
declName_1 : declType_1 \\
\vdots \\
declName_n : declType_n \\
\hline
\end{array}
$$

$$
SchemaName \mathrel{\widehat{=}} [declName_1 : declType_1; \ldots; declName_n : declType_n \mid predicate_1; \ldots; predicate
$$

Figure 3.5 Z state schema templates

Both the vertical and horizontal formats describe the same system elements.

Figure 3.5presents three templates for specifying state schemas in Z; the two formats at the top of the figure are called the vertical format while the one at the bottom is the horizontal format.

A state schema in Z must have a unique name that has not occurred previously in the specification. The vertical formats at the top-left and top-right respectively describe state schemas with and without the predicate part.

We present a sample state schema, *BVMState*, in Figure 3.6. *BVMState* describes the state space of the BVM based on the state variables defined in the DEVS model of BVM in Section 3.2.3.1. BVM reuses the type *COIN* and function *v* defined previously in the basic type definition and axiomatic definition respectively.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{BVMState}} \\
\hline
vault : [\![COIN]\!] \\
escrow : [\![COIN]\!] \\
badC : \mathbb{P}\, COIN \\
credit, price, current : \mathbb{P}\, \mathbb{N} \\
\Delta \\
\phi : \{idle, charge, dispense, cancel, return, reject\} \\
\hline
v(badC) \notin \{10, 20, 50, 100, 200\} \\
price \in \{0, 80, 100, 120, 130\} \\
current \in \{0, 1, 2, 3, 4, 5\} \\
\phi = idle \Rightarrow credit = current = price = 0 \land badC = null \land escrow = [\![]\!] \\
\phi = charge \Rightarrow current \in \{1, 2, 3, 4\} \land price > 0 \land credit < price \land badC = null \\
\phi = dispense \Rightarrow current \in \{1, 2, 3, 4\} \land price > 0 \land credit = price \land badC = null \land escrow = [\![]\!] \\
\phi = return \Rightarrow current \in \{1, 2, 3, 4\} \land price > 0 \land credit > price \land badC = null \land escrow = [\![]\!] \\
\phi = cancel \Rightarrow current = 5 \land credit < price \\
\phi = reject \Rightarrow current \in \{1, 2, 3, 4\} \land badC \neq null \\
\hline
\end{array}
$$

Figure 3.6 State space of the beverage vending machine

As explained previously, each of state variables *vault* and *escrow* is a bag of coins. Variable *badC* has type *COIN* while the first predicate in the schema specifies a constraint that helps to define, precisely, the subset of the set *COIN* of all coins to which it belongs; *badC* specifies the so-called invalid coin described in the synopsis of BVM in Section 3.2.1. Similarly, variables *credit*, *price* and *current* are all of type ℕ while the second (resp. third) predicate of the schema sets a constraint on the subset of ℕwithin which legal values of *price* (resp. *current*) must be taken. Variable ϕ is a *secondary* or *derived* variable whose values depend on the instantaneous values of other variables as specified by the remaining predicates in the schema.

The starting (initial) state of the BVM is specified as in Figure below. This schema simply specifies that BVM will initialize to state $\phi = idle$when started.

$$\begin{array}{|l}
\hline InitBVM \\
BVMState \\
\hline \phi = idle \\
\hline
\end{array}$$

Figure 3.7 Initial state of BVM

3.2.4.5 Operation schema

An operation schema specifies one or more operations on the state variables defined in the state schema. Essentially, an operation schema *includes* the state schema(s) upon which it operates in its specification by referencing them.

The reference to a state schema from an operation schema specifies whether the referencing is for read-only or for modification; the former and latter cases are described in the templates presented on the right and left respectively of Figure 3.8. The symbols Δ (delta) in the operation schema *StateChangingOperationSchema* (see left of Figure 3.8) denotes that the state schema referred to by the reference *StateSchemaRef* will be modified after the execution of the present operation schema. i.e., it will lead to a change of state. In contrast, the symbol Ξ (Xi) on the right of the figure denotes that the reference to *StateSchemaRef* is read-only and will not lead to a change of state. i.e., the values of all variables in the state schema before and after the execution of the operation schema will remain the same.

$$\begin{array}{|l}
\hline StateChangingOperationSchema \\
\Delta StateSchemaRef \\
[declName_1 : declType_1 \\
\vdots \\
declName_n : declType_n] \\
[inputVariable_1? : varType_1 \\
\vdots \\
inputVariable_n? : varType_n] \\
[outputVariable_1! : varType_1 \\
\vdots \\
outputVariable_n! : varType_n] \\
\hline [precondition_1 \\
\vdots \\
precondition_n] \\
[postcondition_1 \\
\vdots \\
postcondition_n] \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline StatePreservingOperationSchema \\
\Xi StateSchemaRef \\
[declName_1 : declType_1 \\
\vdots \\
declName_n : declType_n] \\
[inputVariable_1? : varType_1 \\
\vdots \\
inputVariable_n? : varType_n] \\
[outputVariable_1! : varType_1 \\
\vdots \\
outputVariable_n! : varType_n] \\
\hline [precondition_1 \\
\vdots \\
precondition_n] \\
[postcondition_1 \\
\vdots \\
postcondition_n] \\
\hline
\end{array}$$

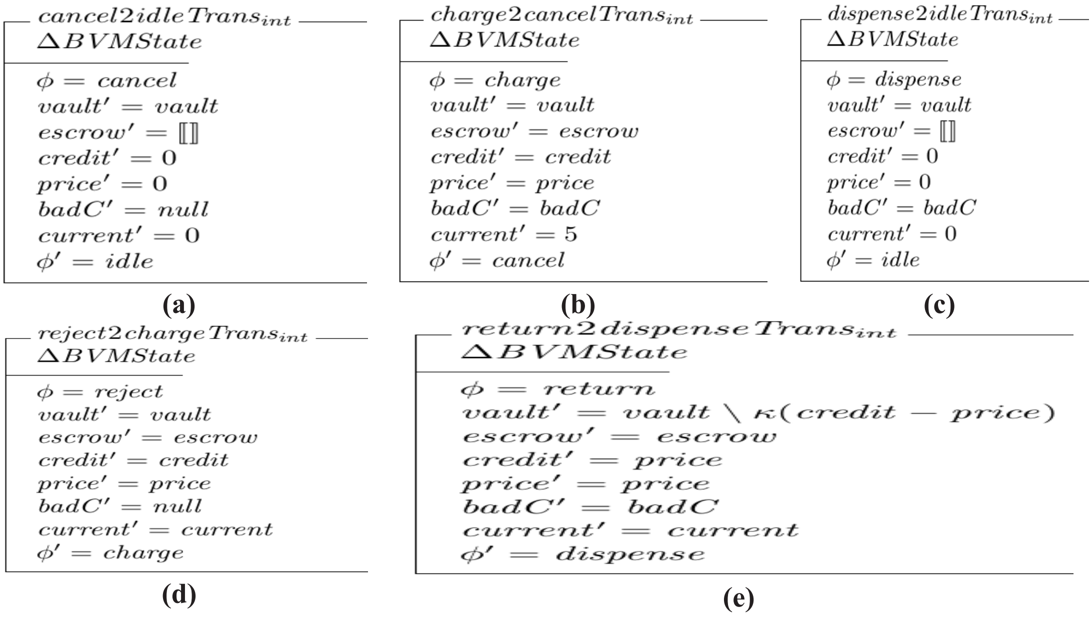
Figure 3.8Templates for Z operation schema

Δ in the template on the left denotes that *StateSchemaRef* will be modified by the operation schema while Ξ on the right indicates that the operation schema only reads the contents of *StateSchemaRef* without modifying it.

As shown in Figure 3.8, an operation schema has a unique name specified on top of the schema and may declare some local variables. In addition to the local variables, it may also declare input and/or output variables; input and output variable names end with the question mark '?' and exclamation '!' symbols respectively. The predicate part allows for the specification of the pre- and post-conditions of the operation. On the one hand, the preconditions specify the constraints, on the included state schema, local variables or input variables or any combination of these three, which must be satisfied before the operation can be executed. The post-conditions, on the other hand, specify the effects of executing the operation on its local variables, the included state schema and the output variables if any. Conventionally, the final values of the different variables, in the post-conditions, are represented by "primed" variable names. i.e., they are differentiated by appending the prime (′) symbol to the variable names.

The operation schema is particularly useful to specify the behavior of a state-based system by describing the state transition operations. As examples to illustrate the specification of operation schemas, we will present the Z equivalent of the state transition and output functions of the BVM as described in its DEVS models in Section 3.2.3.1.

Figure 3.9 (a-e) presents the internal state transition operations of BVM. Each of the operations includes the state schema, *BVMState* defined previously with the Δ reference; this indicates that each operation has full access to all variables declared in the state schema and that the system's state is modified whenever the operations is executed. We have chosen to specify the different cases of internal transition separately because they have disparate pre- and post-conditions.



$cancel2idleTrans_{int}$
$\Delta BVMState$
$\phi = cancel$
$vault' = vault$
$escrow' = [\![ ]\!]$
$credit' = 0$
$price' = 0$
$badC' = null$
$current' = 0$
$\phi' = idle$

(a)

$charge2cancelTrans_{int}$
$\Delta BVMState$
$\phi = charge$
$vault' = vault$
$escrow' = escrow$
$credit' = credit$
$price' = price$
$badC' = badC$
$current' = 5$
$\phi' = cancel$

(b)

$dispense2idleTrans_{int}$
$\Delta BVMState$
$\phi = dispense$
$vault' = vault$
$escrow' = [\![ ]\!]$
$credit' = 0$
$price' = 0$
$badC' = badC$
$current' = 0$
$\phi' = idle$

(c)

$reject2chargeTrans_{int}$
$\Delta BVMState$
$\phi = reject$
$vault' = vault$
$escrow' = escrow$
$credit' = credit$
$price' = price$
$badC' = null$
$current' = current$
$\phi' = charge$

(d)

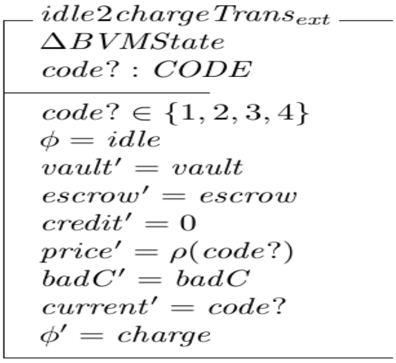
$return2dispenseTrans_{int}$
$\Delta BVMState$
$\phi = return$
$vault' = vault \setminus \kappa(credit - price)$
$escrow' = escrow$
$credit' = price$
$price' = price$
$badC' = badC$
$current' = current$
$\phi' = dispense$

(e)

Figure 3.9 Internal state transition operations of the BVM

Recall from our previous explanation that the post-conditions are identified in the predicate part of the schema by the "primed" variable names and they specify the relationships between the values of the state variables before and after the execution of the operation without necessarily giving the details of how we arrive at such relationships.

Operation *return2dispenseTrans_{int}* has the pre-condition $\phi = return$, which implies that all the constraints imposed by this state must be *true* before the operation can be executed. All other references to the state variables in this predicate part specify the post-condition of the operation since they are "primed". Predicate $vault^{'} = vault\backslash\kappa(credit - price)$ specifies that the state, $vault^{'}$, of variable *vault* after the execution of the operation is the set minus of the bag of coins returned by $\kappa(credit - price)$ from the state, *vault*, of the variable before the operation is executed. From our previous definition of function κ, $\kappa(credit - price)$ returns a bag of coins whose total value is equivalent to the balance, *credit-price*, of the transaction in progress. State variables *credit* and ϕ also take new values (i.e., $credit^{'} = price$and $\phi^{'} = dispense$ respectively) while *escrow*, *price*, *badC* and *current* remain unchanged after executing the operation. Other operations in the figure can be read in similar manner.

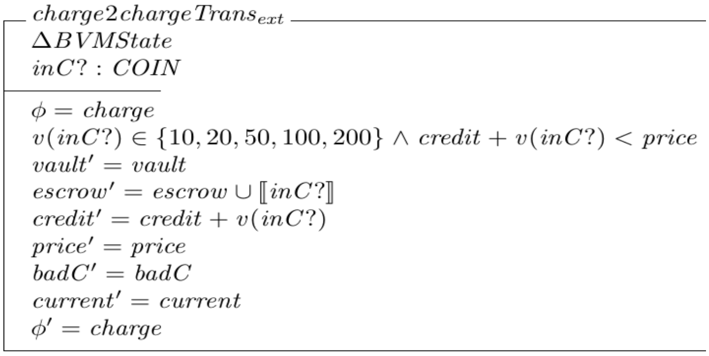


$idle2chargeTrans_{ext}$

$\Delta BVMState$
$code? : CODE$

$code? \in \{1, 2, 3, 4\}$
$\phi = idle$
$vault' = vault$
$escrow' = escrow$
$credit' = 0$
$price' = \rho(code?)$
$badC' = badC$
$current' = code?$
$\phi' = charge$

(a)

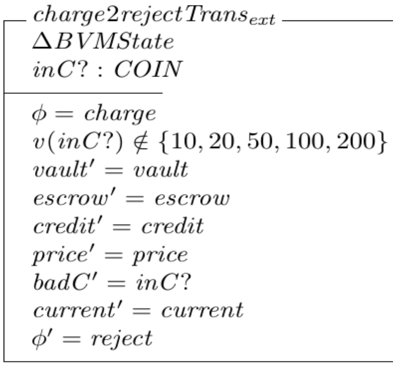


$charge2chargeTrans_{ext}$

$\Delta BVMState$
$inC? : COIN$

$\phi = charge$
$v(inC?) \in \{10, 20, 50, 100, 200\} \wedge credit + v(inC?) < price$
$vault' = vault$
$escrow' = escrow \cup [\![inC?]\!]$
$credit' = credit + v(inC?)$
$price' = price$
$badC' = badC$
$current' = current$
$\phi' = charge$

(b)

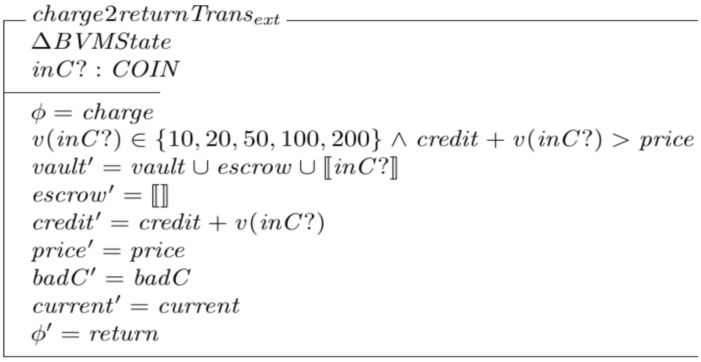


$charge2rejectTrans_{ext}$

$\Delta BVMState$
$inC? : COIN$

$\phi = charge$
$v(inC?) \notin \{10, 20, 50, 100, 200\}$
$vault' = vault$
$escrow' = escrow$
$credit' = credit$
$price' = price$
$badC' = inC?$
$current' = current$
$\phi' = reject$

(c)

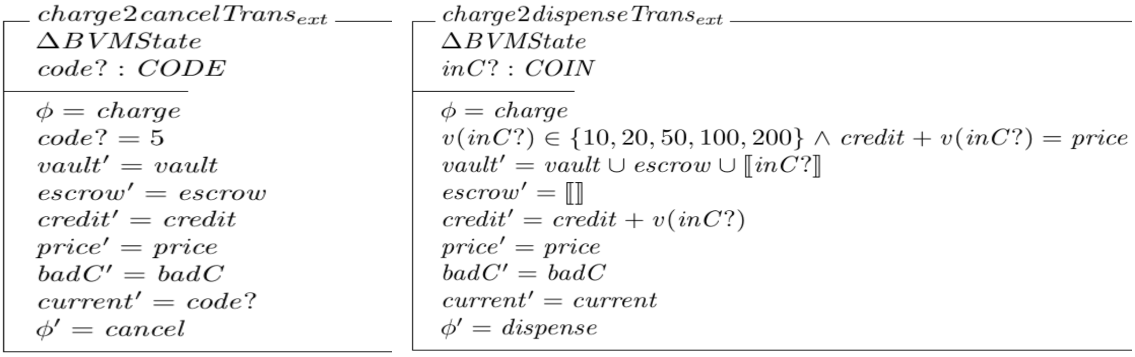


$charge2returnTrans_{ext}$

$\Delta BVMState$
$inC? : COIN$

$\phi = charge$
$v(inC?) \in \{10, 20, 50, 100, 200\} \wedge credit + v(inC?) > price$
$vault' = vault \cup escrow \cup [\![inC?]\!]$
$escrow' = [\![]\!]$
$credit' = credit + v(inC?)$
$price' = price$
$badC' = badC$
$current' = current$
$\phi' = return$

(d)



$charge2cancelTrans_{ext}$

$\Delta BVMState$
$code? : CODE$

$\phi = charge$
$code? = 5$
$vault' = vault$
$escrow' = escrow$
$credit' = credit$
$price' = price$
$badC' = badC$
$current' = code?$
$\phi' = cancel$

(e)

$charge2dispenseTrans_{ext}$

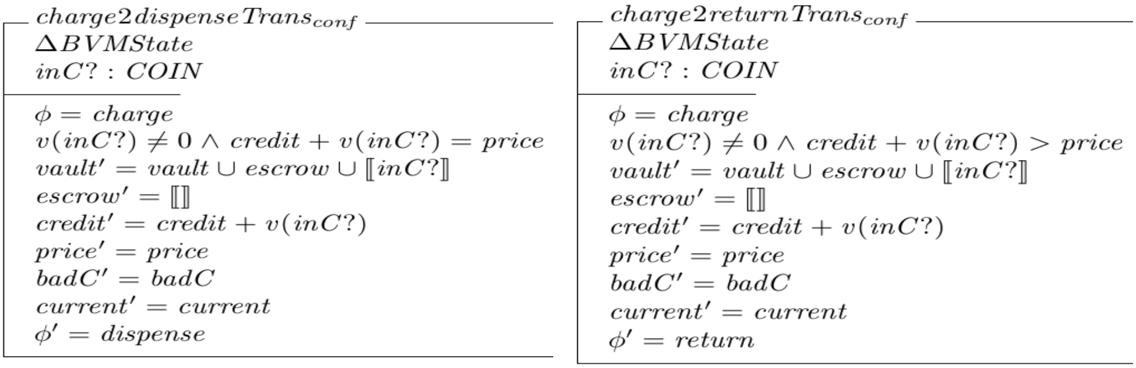
$\Delta BVMState$
$inC? : COIN$

$\phi = charge$
$v(inC?) \in \{10, 20, 50, 100, 200\} \wedge credit + v(inC?) = price$
$vault' = vault \cup escrow \cup [\![inC?]\!]$
$escrow' = [\![ ]\!]$
$credit' = credit + v(inC?)$
$price' = price$
$badC' = badC$
$current' = current$
$\phi' = dispense$

(f)

Figure 3.10External transition operations of the BVM

Figure 3.10 (a-f) presents another set of operation schemas that specifies the state transition behaviors of BVM when it receives different kinds of input events while in different states. The schema in (a) describes the transition behavior when an input value *code* in the set {1, 2, 3, 4} while the system is in state $\phi = idle$. Schema (b) describes the transition behavior upon the receipt of a "valid" coin (indicated by $v(inC?) \in \{10, 20, 50, 100, 100\}$) that satisfies the constraint specified by the second line of predicate while in state $\phi = charge$. The transition specified in schema (c) describes the transition behavior that is triggered by the receipt of an "invalid" coin (indicated by $v(inC?) \notin \{10, 20, 50, 100, 100\}$) while in state $\phi = charge$. The transition behavior exhibited when the cancel button (*code?=5*) is pressed while the system is in state $\phi = charge$ is described in schema (e). All post-conditions can be read in the same manner as the schemas for the internal state transitions.



$charge2dispenseTrans_{conf}$

$\Delta BVMState$
$inC? : COIN$

$\phi = charge$
$v(inC?) \neq 0 \wedge credit + v(inC?) = price$
$vault' = vault \cup escrow \cup [\![inC?]\!]$
$escrow' = [\![ ]\!]$
$credit' = credit + v(inC?)$
$price' = price$
$badC' = badC$
$current' = current$
$\phi' = dispense$

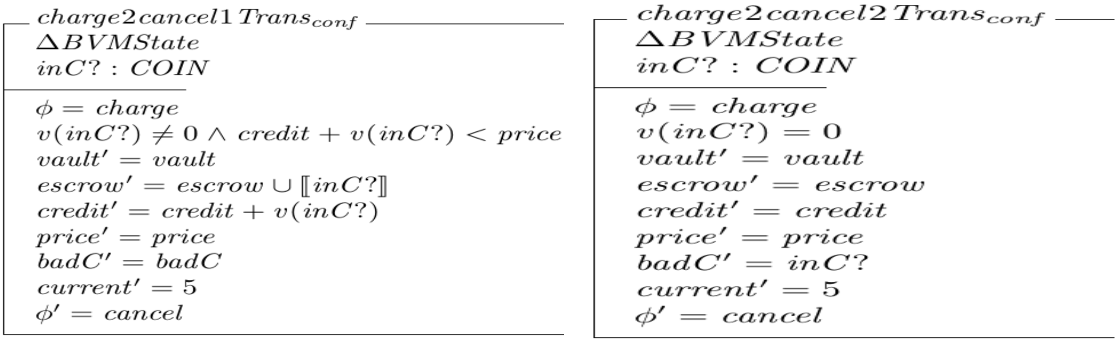
(a)

$charge2returnTrans_{conf}$

$\Delta BVMState$
$inC? : COIN$

$\phi = charge$
$v(inC?) \neq 0 \wedge credit + v(inC?) > price$
$vault' = vault \cup escrow \cup [\![inC?]\!]$
$escrow' = [\![ ]\!]$
$credit' = credit + v(inC?)$
$price' = price$
$badC' = badC$
$current' = current$
$\phi' = return$

(b)



$charge2cancel1\,Trans_{conf}$

$\Delta BVMState$

$inC? : COIN$

$\phi = charge$

$v(inC?) \neq 0 \wedge credit + v(inC?) < price$

$vault' = vault$

$escrow' = escrow \cup [\![inC?]\!]$

$credit' = credit + v(inC?)$

$price' = price$

$badC' = badC$

$current' = 5$

$\phi' = cancel$

(c)

$charge2cancel2\,Trans_{conf}$

$\Delta BVMState$

$inC? : COIN$

$\phi = charge$

$v(inC?) = 0$

$vault' = vault$

$escrow' = escrow$

$credit' = credit$

$price' = price$

$badC' = inC?$

$current' = 5$

$\phi' = cancel$

(d)

Figure 3.11 Confluent transition operations of BVM

The last in the series of sets of state transition schemas of BVM is that of the confluent state transitions shown in Figure 3.11. All schemas in the set can be read same way as the previous ones as well.

We present the specification of the output operations of BVM in Figure3.12. Unlike the state transition schemas which all have full access to the variables declared in the included state schema, *BVMOutput* includes state schema *BVMState* as *read-only* as denoted by the preceding symbol Ξ. This implies that all state variables remain unchanged by the execution of the operations specified in this schema.

Another feature that differentiates this schema from the previous ones is the declaration of output variables - *cup*! and *outC*!. This schema actually combines four distinct output operations with the pre-condition of each as specified in the predicate part of the schema. For instance, predicate $\phi = dispense \Rightarrow cup! = \psi(current) \wedge outC! = [\![\ ]\!]$ specifies that if the pre-condition $\phi = dispense$ is *true*, then the post-condition will be that the value returned by $\psi(current)$ will be placed on output variable *cup*! while output variable *outC*! will be an empty bag of coins.

Note that the post-execution value of an output variable is not primed; being an output variable already connotes that it will only be assigned a value at the end of the execution. Other output operations in the schema can be read in similar manner.

$$
\begin{array}{|l}
\multicolumn{1}{l}{BVMOutput} \\
\hline
\Xi BVMState \\
cup! : CUP \\
outC! : [\![COIN]\!] \\
\hline
\phi = dispense \Rightarrow cup! = \psi(current) \land outC! = [\![\,]\!] \\
\phi = return \Rightarrow cup! = null \land outC! = \kappa(credit - price) \\
\phi = reject \Rightarrow cup! = null \land outC! = [\![badC]\!] \\
\phi = cancel \land (escrow \neq [\![\,]\!] \lor badC \neq null) \Rightarrow cup! = null \land outC! = escrow \cup [\![badC]\!] \\
\phi \neq dispense \land \phi \neq return \land \phi \neq reject \land \\
\neg\ (\phi = cancel \land (escrow \neq [\![\,]\!] \lor badC \neq null)) \Rightarrow cup! = null \land outC! = [\![\,]\!] \\
\hline
\end{array}
$$

Figure3.12Output operations of BVM

3.2.4.6 Z schema calculus

Schema calculus involves the building of schema expressions for hierarchical construction of complex schemas using logical connectives to combine schemas with type-compatible signatures. Two signatures are said to be type-compatible if each variable common to the two has the same type in both of them [Spi92].Hence, two type-compatible signatures can be combined into a larger signature containing the union of the sets of variables in the two. Using this as a premise, when two schemas, A and B, are combined to form a bigger schema C, the declaration part (signature) of C contains all the variables declared in A and B while the combination of the constraints in its predicate part will depend on the logical connective involved.

Logical connectives such as $\wedge, \vee, \Rightarrow$or$\Leftrightarrow$ may be used to combine two type-compatible schemas; it is, however, preferable to call them *schema connectives* in this context as they have more complicated semantics than in conventional logic. In each case, the resulting schema is one whose signature is a merge of the signatures of the two arguments while the predicate part is the result of joining the predicate parts of the two arguments with the chosen schema connective. The unary operator $\neg$ may also be used to express the negation of a schema; in this case, schemas S and $\neg$S have identical signatures but the properties (constraints in the predicate part of the schema) of the latter are the exact negations of the properties of the former.

Next, we will present a series of examples of the use of schema calculus in the specification of our BVM. Firstly, the five schemas presented previously in Figure 3.9 (a-e) are different cases of internal state transition behavior of the BVM and only one of the cases can be executed at any time. Hence, we can combine them into one big schema using the schema$\vee$ connective as follows:

$$BVMInternalTransition \triangleq charge2cancelTrans_{int} \ \vee \ cancel2idleTrans_{int} \ \vee \\ reject2chargeTrans_{int} \ \vee \ reject2dispenseTrans_{int} \ \vee \ cancel2idleTrans_{int}$$

As described previously, an internal transition may be accompanied by an output operation, which will occur just before the transition to the next state. An output event may occur at the end of a state just before a transition to the next state; we have specified the possible cases of output operations in schema *BVMOutput* (see Figure3.12). We can use the ∧ connective to combine the output and internal transition operations as follows:

$$BVMInternalTransitionEvent \triangleq BVMOutput \land BVMInternalTransition$$

Similarly, we combine the six cases of external state transitions presented in Figure 3.10 into one big schema as follows:

$$BVMExternalTransitionEvent \triangleq idle2chargeTrans_{ext} \lor charge2chargeTrans_{ext} \lor charge2dispenseTrans_{ext} \lor charge2returnTrans_{ext} \lor charge2returnTrans_{ext} \lor charge2cancelTrans_{ext}$$

No output operation accompanies an external state transition, hence we will not combine the $BVMExternalTransitionEvent$ with *BVMOutput*.

Using the same techniques, we combine the cases of confluent transition operations presented in Figure 3.11 under one schema as follows:

$$BVMConfluentTransitionEvent \triangleq BVMOutput \land (charge2dispenseTrans_{conf} \lor charge2returnTrans_{conf} \lor charge2cancel1Trans_{conf} \lor charge2cancel2Trans_{conf})$$

The combined confluent transition schema also contains the output schema because there is possibility of an output operation accompanying a confluent transition.

Finally, only one out of the three kinds of state transitions can occur at time; hence, we combine the all transition operations under one bigger schema as:

$$BVMStateTransitionEvent \triangleq BVMInternalTransitionEvent \lor BVMExternalTransitionEvent \lor BVMConfluentTransitionEvent$$

3.2.5 Object-Z

Object-Z [Smi92, Smi12] is an Object-Oriented extension of Z. It adopts the concept of class from Object Orientation (OO) to add structure, modularity and clarity to Z specifications. The fundamental difference between Z and Object-Z is the presence of *class schema* in the latter. Object-Z introduces the concept of "class schema", which is defined on top of Z’s notion of schema; an Object-Z's class schema encloses a single state schema and all the operation schemas that manipulate and/or use its declared variables. Hence, the basic building block for system specification in Object-Z is the class schema. In addition to the encapsulation property, the class schema exhibits other OO properties like inheritance and polymorphism. Another interesting component of Object-Z, which is not present in Z, is its integration with temporal logic to specify some history invariants of systems.

$ClassName[Parameter_1, \ldots, Parameter_n]$
$[Visibility\ list]$
$[Inherited\ class_1$
$\vdots$
$Inherited\ class_n]$
$[Local\ definition_1$
$\vdots$
$Local\ definition_n]$
$[State]$
$[Initial\ state]$
$[Operation_1$
$\vdots$
$Operation_n]$
$[History\ invariant_1$
$\vdots$
$History\ invariant_n]$

Figure 3.13Syntactic structure of class schema

Figure 3.13 presents the syntactic structure of the class schema in the form of a general template for specification, showing its possible elements and the orders in which they may appear. A class schema has a unique name as an identifier to differentiate it from other classes in the specification. In addition to the class name, the header may specify some generic parameters. Since the class schema encapsulates its contents, the *visibility list* specifies the interface through which the elements of an object of the class may be accessed i.e., a list of variables and operations that can be visible outside the class in similitude to public attributes and methods in OO.

An *Inherited Class* designator provides a reference to an existing Class schema whose definition is imported for reuse in the current class in similitude to the concept of inheritance in OO.

A *Local Definition* may be a local type or constant definition (usually specified in an axiomatic definition) or a reference to another class.

A class schema may have a maximum of one state schema, represented as *State* in the template, which defines its state space through the declaration of state variables and invariants (if any). The state schema in Object-Z is same as in Z except that it does not have a schema name; i.e., a state schema in Object-Z assumes the name of the class schema that encapsulates it.

This state schema may be followed by a specification of the *initial state* schema, simply referred to as *init*, which specifies a set of predicates that must be satisfied by the state variables (declared in the state schema) of every new object of the class before it undergoes any change of state.

$$\begin{array}{|l}
\hline OperationName \\
[\Delta(Delta\ list)] \\
[declName_1 : declType_1 \\
\vdots \\
declName_n : declType_n] \\
[inputVariable_1? : varType_1 \\
\vdots \\
inputVariable_n? : varType_n] \\
[outputVariable_1! : varType_1 \\
\vdots \\
outputVariable_n! : varType_n] \\
\hline [precondition_1 \\
\vdots \\
precondition_n] \\
[postcondition_1 \\
\vdots \\
postcondition_n] \\
\hline
\end{array}$$

Figure 3.14Syntactic structure of Object-Z operation

The init schema may be followed by operation schemas, simply referred to as operations in Object-Z, which use and/or manipulate the other elements of the class schema. Unlike Z operation schemas, which explicitly declare the state schemas they operate on, an Object-Z operation inherently has full access to all variables declared in the only state schema encapsulated in the same class schema with it. This is premised on the understanding that in OO, an operation has implicit access to all attributes declared in its class. An Object-Z operation, however, declares an optional *Delta list*, preceded by the Δ symbol, which indicates the state variables that will be modified when the operation is executed (see the first line in Figure 3.14). An empty or absent delta list implies that the operation does not cause any change of state.

Finally, a class schema in Object-Z (Figure 3.13) may contain an optional set of Temporal Logic-based history invariants, which may contain liveness properties that must be satisfied by the operations of the class schema. The history invariants are specified below a dividing line that separates them from other features of the class schema. Conceptually, the dividing line serves a similar purpose as the dividing line of any Z schema: the same way predicates below the dividing line of a schema constrains the declarations above the line, history invariants in class schema specify constraints on the set of derivable histories from the state and operations of a class schema [Smi92]. At this point, we will defer further discussions on Temporal Logic and related requirement properties until Section 3.2.6.

As an example of system specification with Object-Z, we present, Figure 3.15, the Object-Z specification of the BVM. Intuitively, this is a variant of the Z-specification of the BVM, which we discussed in Section 3.2.4.

The *BVM* class schema starts with *local definitions* consisting of the free type definitions *COIN*, *CUP* and *CODE* as well as the axiomatic definition, all of which are as described previously in the Z specification. This is followed by the state schema, which is same as that in the Z specification except that it has no name. Following the state schema is the *Init* schema that specifies the system's initial state. The series of *operations* in the class schema are variants of the respective operation schemas presented in the Z specification. As we have pointed out earlier, there are mainly two differences between each of the operations and its corresponding operation schema in Z:

- Unlike the explicit inclusion of the state schema in the latter, the former is considered to have implicit access to all the declarations in the state schema of its class.
- The former declares a delta list, which indicates the variables (declared in the state schema) that will be modified by the operation; thus, the "after-execution" values of only the variables mentioned in the delta list are specified while others are considered to be unchanged. The operations are followed by a series of schema compositions, using schema calculus, which separately combine the internal state transition operations, external state transition operations and confluent state transition operations as specified previously in the Z specification of the system.

BVM

$COIN ::= 1cent \mid 2cent \mid 5cent \mid 10cent \mid 20cent \mid 50cent \mid 1euro \mid 2euro$

$CUP ::= cocoa \mid coffee \mid orange \mid apple$

$CODE ::= 1 \mid 2 \mid 3 \mid 4$

$v : COIN \longrightarrow \{1, 2, 5, 10, 20, 50, 100, 200\};\ \psi : \{1, 2, 3, 4\} \longrightarrow CUP$

$\sigma : [\![COIN]\!] \longrightarrow \mathbb{N};\ \kappa : \mathbb{N} \nrightarrow \mathbb{P}[\![COIN]\!]$

$v : 1cent \mapsto 1 \wedge v : 2cent \mapsto 2 \wedge v : 5cent \mapsto 5 \wedge v : 10cent \mapsto 10 \wedge$

$v : 20cent \mapsto 20 \wedge v : 50cent \mapsto 50 \wedge v : 1euro \mapsto 100 \wedge v : 2euro \mapsto 200$

$\rho : 1 \mapsto 100 \wedge \rho : 2 \mapsto 80 \wedge \rho : 3 \mapsto 120 \wedge \rho : 4 \mapsto 130 \wedge \rho : 5 \mapsto 0$

$\psi : 1 \mapsto cocoa \wedge \psi : 2 \mapsto coffee \wedge \psi : 3 \mapsto orange \wedge \psi : 4 \mapsto apple$

$\forall\, C : [\![COIN]\!] \bullet \sigma(C) = \sum_{i=1}^{i=|C|} v(C_i);$

$\forall (n, C) \in \kappa \bullet \sigma(C) = n$

$vault, escrow : [\![COIN]\!];\ badC : \mathbb{P}\, COIN;\ credit, price, current : \mathbb{P}\, \mathbb{N}$

$\phi : \{idle, charge, dispense, cancel, return, reject\}$

$v(badC) \notin \{10, 20, 50, 100, 200\} \wedge$

$price \in \{0, 80, 100, 120, 130\} \wedge current \in \{0, 1, 2, 3, 4, 5\}$

$\phi = idle \Rightarrow credit = current = price = 0 \wedge badC = null \wedge escrow = [\![]\!]$

$\phi = charge \Rightarrow current \in \{1, 2, 3, 4\} \wedge price > 0 \wedge credit < price \wedge badC = null$

$\phi = dispense \Rightarrow current \in \{1, 2, 3, 4\} \wedge price > 0 \wedge credit = price \wedge badC = null$

$\wedge\ escrow = [\![]\!]$

$\phi = return \Rightarrow current \in \{1, 2, 3, 4\} \wedge price > 0 \wedge credit > price \wedge badC = null$

$\wedge\ escrow = [\![]\!]$

$\phi = cancel \Rightarrow current = 5 \wedge credit < price$

$\phi = reject \Rightarrow current \in \{1, 2, 3, 4\} \wedge badC \neq null$

$\textsc{Init}$

$\phi = idle$

$charge2cancelTrans_{int}$

$\Delta(current, \phi)$

$\phi = charge \land current' = 5 \land \phi' = cancel$

$cancel2idleTrans_{int}$

$\Delta(escrow, credit, price, badC, curent, \phi)$

$\phi = cancel \land$
$escrow' = \llbracket\rrbracket \land credit' = 0 \land price' = 0 \land badC' = null \land current' = 0 \land \phi' = idle$

$reject2chargeTrans_{int}$

$\Delta(badC, \phi)$

$\phi = reject \land badC' = null \land \phi' = charge$

$return2dispenseTrans_{int}$

$\Delta(vault, \phi)$

$\phi = return \land vault' = vault \setminus \kappa(credit - price) \land \phi' = dispense$

$dispense2idleTrans_{int}$

$\Delta(escrow, credit, price, current, \phi)$

$\phi = dispense \land escrow' = \llbracket\rrbracket \land credit' = 0 \land price' = 0 \land current' = 0 \land \phi' = idle$

$idle2chargeTrans_{ext}$

$\Delta(credit, price, current, \phi)$
$code? : CODE$

$\phi = idle \land code? \in \{1, 2, 3, 4\} \land$
$credit' = 0 \land price' = \rho(code?) \land current' = code? \land \phi' = charge$

$charge2chargeTrans_{ext}$

$\Delta(escrow, credit, \phi)$
$inC? : COIN$

$\phi = charge \land v(inC?) \in \{10, 20, 50, 100, 200\} \land credit + v(inC?) < price \land$
$escrow' = escrow \cup \llbracket inC? \rrbracket \land credit' = credit + v(inC?) \land \phi' = charge$

$\mathit{charge2dispenseTrans}_{ext}$

$$\Delta(\mathit{vault}, \mathit{escrow}, \mathit{credit}, \phi)$$
$$\mathit{inC?} : \mathit{COIN}$$

$$\phi = \mathit{charge} \land v(\mathit{inC?}) \in \{10, 20, 50, 100, 200\} \land \mathit{credit} + v(\mathit{inC?}) = \mathit{price} \land$$
$$\mathit{vault}' = \mathit{vault} \cup \mathit{escrow} \cup [\![\mathit{inC?}]\!] \land \mathit{escrow}' = [\![]\!] \land \mathit{credit}' = \mathit{credit} + v(\mathit{inC?}) \land$$
$$\phi' = \mathit{dispense}$$

$\mathit{charge2returnTrans}_{ext}$

$$\Delta(\mathit{vault}, \mathit{escrow}, \mathit{credit}, \phi)$$
$$\mathit{inC?} : \mathit{COIN}$$

$$\phi = \mathit{charge} \land v(\mathit{inC?}) \in \{10, 20, 50, 100, 200\} \land \mathit{credit} + v(\mathit{inC?}) > \mathit{price} \land$$
$$\mathit{vault}' = \mathit{vault} \cup \mathit{escrow} \cup [\![\mathit{inC?}]\!] \land \mathit{escrow}' = [\![]\!] \land \mathit{credit}' = \mathit{credit} + v(\mathit{inC?}) \land$$
$$\phi' = \mathit{return}$$

$\mathit{charge2rejectTrans}_{ext}$

$$\Delta(\mathit{badC}, \phi)$$
$$\mathit{inC?} : \mathit{COIN}$$

$$\phi = \mathit{charge} \land v(\mathit{inC?}) \notin \{10, 20, 50, 100, 200\} \land \mathit{badC}' = \mathit{inC?} \land \phi' = \mathit{reject}$$

$\mathit{charge2cancelTrans}_{ext}$

$$\Delta(\mathit{current}, \phi)$$
$$\mathit{code?} : \mathit{CODE}$$

$$\phi = \mathit{charge} \land \mathit{code?} = 5 \land \mathit{current}' = \mathit{code?} \land \phi' = \mathit{cancel}$$

$\mathit{charge2dispenseTrans}_{conf}$

$$\Delta(\mathit{vault}, \mathit{escrow}, \mathit{credit}, \phi)$$
$$\mathit{inC?} : \mathit{COIN}$$

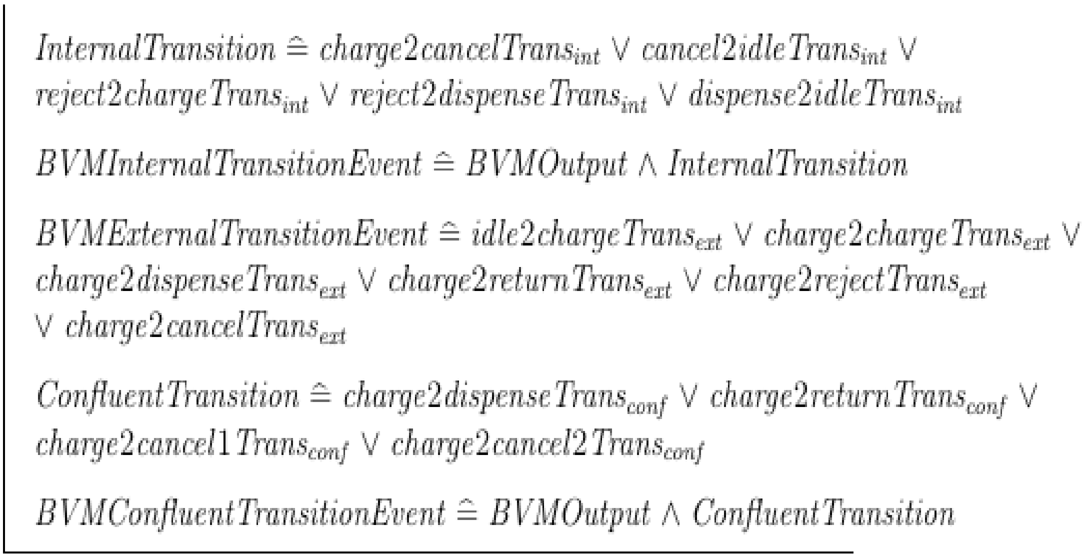
$$\phi = \mathit{charge} \land v(\mathit{inC?}) \in \{10, 20, 50, 100, 200\} \land \mathit{credit} + v(\mathit{inC?}) = \mathit{price} \land$$
$$\mathit{vault}' = \mathit{vault} \cup \mathit{escrow} \cup [\![\mathit{inC?}]\!] \land \mathit{escrow}' = [\![]\!] \land \mathit{credit}' = \mathit{credit} + v(\mathit{inC?})$$
$$\phi' = \mathit{dispense}$$

$$\begin{array}{|l}
\textit{charge2returnTrans}_{conf} \\
\hline
\Delta(vault, escrow, credit, \phi) \\
inC? : COIN \\
\hline
\phi = charge \land v(inC?) \in \{10, 20, 50, 100, 200\} \land credit + v(inC?) > price \land \\
vault' = vault \cup escrow \cup [\![inC?]\!] \land escrow' = [\![]\!] \land credit' = credit + v(inC?) \land \\
\phi' = return \\
\hline
\end{array}$$

$$\begin{array}{|l}
\textit{charge2cancel1Trans}_{conf} \\
\hline
\Delta(escrow, credit, current, \phi) \\
inC? : COIN \\
\hline
\phi = charge \land v(inC?) \in \{10, 20, 50, 100, 200\} \land credit + v(inC?) < price \land \\
escrow' = escrow \cup [\![inC?]\!] \land credit' = credit + v(inC?) \land current' = 5 \land \\
\phi' = cancel \\
\hline
\end{array}$$

$$\begin{array}{|l}
\textit{charge2cancel2Trans}_{conf} \\
\hline
\Delta(badC, current, \phi) \\
inC? : COIN \\
\hline
\phi = charge \land v(inC?) \notin \{10, 20, 50, 100, 200\} badC' = inC? \land current' = 5 \land \\
\phi' = cancel \\
\hline
\end{array}$$

$$\begin{array}{|l}
\textit{BVMOutput} \\
\hline
\Delta() \\
cup! : CUP \\
outC! : [\![COIN]\!] \\
\hline
\phi = dispense \Rightarrow cup! = \psi(current) \land outC! = [\![]\!] \\
\phi = return \Rightarrow cup! = null \land outC! = \kappa(credit - price) \\
\phi = reject \Rightarrow cup! = null \land outC! = [\![badC]\!] \\
\phi = cancel \land (escrow \neq [\![]\!] \lor badC \neq null) \Rightarrow cup! = null \land outC! = escrow \cup [\![badC]\!] \\
\phi \neq dispense \land \phi \neq return \land \phi \neq reject \land \\
\neg\,(\phi = cancel \land (escrow \neq [\![]\!] \lor badC \neq null)) \Rightarrow cup! = null \land outC! = [\![]\!] \\
\hline
\end{array}$$



$$InternalTransition \mathrel{\widehat=} charge2cancelTrans_{int} \lor cancel2idleTrans_{int} \lor$$
$$reject2chargeTrans_{int} \lor reject2dispenseTrans_{int} \lor dispense2idleTrans_{int}$$

$$BVMInternalTransitionEvent \mathrel{\widehat=} BVMOutput \land InternalTransition$$

$$BVMExternalTransitionEvent \mathrel{\widehat=} idle2chargeTrans_{ext} \lor charge2chargeTrans_{ext} \lor$$
$$charge2dispenseTrans_{ext} \lor charge2returnTrans_{ext} \lor charge2rejectTrans_{ext}$$
$$\lor\ charge2cancelTrans_{ext}$$

$$ConfluentTransition \mathrel{\widehat=} charge2dispenseTrans_{conf} \lor charge2returnTrans_{conf} \lor$$
$$charge2cancel1Trans_{conf} \lor charge2cancel2Trans_{conf}$$

$$BVMConfluentTransitionEvent \mathrel{\widehat=} BVMOutput \land ConfluentTransition$$

Figure 3.15 Object-Z specification of the BVM

3.2.6 Temporal Logic

Temporal Logic (TL) is a general term used to describe the logical frameworks for representing and reasoning about time and temporal information; it is used in Computer Science as a formalism for the specification and verification of properties of the executions of computer programs and systems [Lam83, Lam94, GG15]. According to Lamport [Lam94], TL is particularly designed for reasoning about algorithms by reasoning about the sequences of states produced, due to changes in the values of one or more variables, when the algorithm is executed. He considered that the execution of an algorithm could be described by the resulting sequence of states; hence, the semantics of the algorithm can be obtained from a collection of all its possible executions (sequences of states). The specification, and reasoning about, such collections of possible executions is what he described as the "province" of TL.

In the context of a systems engineering, the sequence of states visited during execution, in fact, describes the behavior of a system. Hence, we can as well claim that TL can be used for the specification of, and reasoning with, the behavior of an ideal system. This behavior of the ideal system can serve as the metamodel that specifies the required behavioral properties of the real system. Therefore, with the help of verification techniques such as model checking [BKL08, CGP99], we can verify whether or not a given model of the real system satisfies the required properties.

Temporal properties are classified into three broad categories [Lam77]: safety property, liveness property and fairness property. A safety property states that an undesirable event should not happen. i.e., the system should never be in a specified state. A liveness property specifies that an event must occur. i.e., the system must eventually be in a particular state.

As formalisms, TLs extend predicate logic with operators for describing temporal or time-dependent concepts to specify predicates on the sequences of states representing the evolutions of system's states over time [Smi92]. It must be noted, however, that though TLs allow for the specification of relative orders of events or states, they do not refer to the precise or exact timing of such events/states [BLK08]; that is, the concept of time in TL is logical rather than physical.

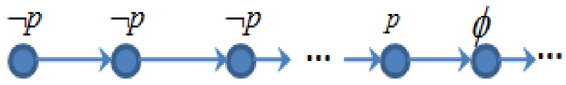
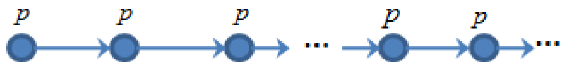
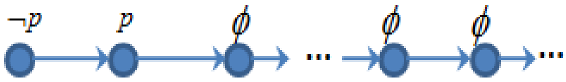
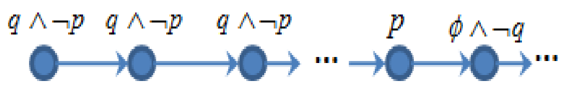
Based on whether the underlying nature of time is linear or branching, TL is classified into two categories [BLK08]: Linear Temporal Logic and Computation Tree Logic.

3.2.6.1 Linear Temporal Logic

The Linear Temporal Logic (LTL) [Pnu77] is based on a linear, description of time; it describes the behavior of a system as an infinite sequence of states. The qualitative notion of time in LTL is path-based, and it is considered to be linear because at every time instant, the system has only one possible successor state; hence it can be said that every time instant has a unique future [BLK08]. Therefore, LTL describes the semantics of a system's behavior as an infinite sequence of states.

Table 3.3 Temporal operators in LTL

p, and *q* and are specific properties usually defined by predicates on system's state variables.ϕ is an arbitrary property, which may include the specific property of interest, or not. In each case of the illustrations, we take the first state in the sequence of states as "*now*" or the state of interest while subsequence states are successors of the previous.

Temporal operators	Symbols	Meanings	Illustrations	
Eventually	◊	At some point(s) in the future	$\Diamond p$	 $\neg p$ $\neg p$ $\neg p$ ... p ϕ ...
Always	□	Now and forever in the future	$\Box p$	
Next	○	Next state/event	$\circ p$	
Until	U	From now until a specified state or event occurs in the future	$q\text{U}p$	 $q \wedge \neg p$ $q \wedge \neg p$ $q \wedge \neg p$ ... p $\phi \wedge \neg q$ ...

Properties are specified in LTL as formulae consisting of atomic propositions, logical connectives (e.g., ¬, ∧,∨, ⇒) and temporal operators. A proposition is an assertion about the values of the system's variables. Temporal operators include *eventually*, *always*, *next*, and *until*; they are described in Table 3.3.

As shown in Table 3.3above, the temporal operators can be described as follows:

Eventually (◊) operator: The *eventually* operator, ◊, is used to specify a property that must be satisfied at some time in the future of a given moment, which serves as the starting point for the search. As the example in Table 3.3 illustrates, starting from the first state in the sequence, $\Diamond p$ specifies that property p must hold at least once in the future. i.e., there exists a state in the future in which p holds; once one of such states is found, it is immaterial whether p holds subsequently or not.

Always (□) operator: The *always* operator, □, is used to specify an invariant property; a property that must hold at the moment of observation and continuously in the future. The example, $\Box p$, in Table 3.3 specifies that property p must hold now and forever.

Next (○) operator: The *next* operator,○, specifies a property that must hold in the successor state of the state of interest. It does not matter whether it continues to hold afterwards or not. We illustrate with an example, in Table 3.3, where $\circ p$ specifies that property p does not hold in the state at which observation starts but must hold in the successor state.

Until (∪) operator: The *until* operator, ∪, operator takes two properties as arguments and specifies that the property on the lhs (left-hand side) must *always* hold *until* the time when the property on the rhs (right-hand side) will hold. In Table 3.3, $q \cup p$ illustrates that property q will continue to hold, while p does not, until the moment when property p eventually holds. From this moment onward, q must not hold any more.

The temporal operators described above may be combined to specify complex properties. For instance, $\Box\Diamond p$ may be read as "property p should always eventually hold" or "property p should hold infinitely often". i.e., at any moment, it should be the case that p will eventually hold in the future. In other words, it states that p should hold "continually". Similarly, the combination $\Diamond\Box p$ specifies that "property p will eventually hold forever". i.e., there is a time in the future from which p should hold "continuously".

The statement that a property p is holds in a state s is written mathematically as $s \vDash p$. The semantics of this in LTL is that all computations originating from state s must satisfy p. This is, in fact, the basis for the path-based description in LTL which postulates that every state of a system has a unique successor during execution. This, however, amounts to a restriction of a system's behavior to a single path of execution, which may not necessarily be true in all cases.

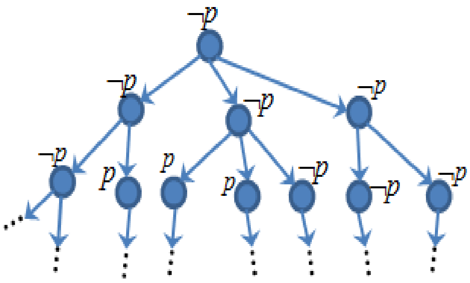
This Computation Tree Logic, an extension of LTL which is presented in the next sub subsection, has addressed this challenge.

3.2.6.2 Computation Tree Logic

The Computation Tree Logic (CTL) [CES86], also known as *branching temporal logic* [BLK08], extends the LTL with a *branching* notion of time which postulates that every moment of time, may have more than one possible future. i.e., a state of a system may have many possible successor states; this is, in fact, inherent in the behavior of a discrete event system. In this case, every possible future of a given moment marks the beginning of a path. Therefore, CTL describes a system's behavior as an infinite tree of states; starting from the root of the tree, there are possibly multiple paths to be followed from every inner node visited.

In addition to the logical and temporal operators used in LTL, CTL supports the use of the *existential path quantifier*∃ (resp. *universal path quantifier*∀) for the specification of properties that must be satisfied by *some* (resp. *all*) computations starting in a state of interest. We describe the application of existential path quantifiers in CTL with illustrations in Table 3.4 and universal path quantifier in Table 3.5. We invite the reader to consult some of the numerous textbooks on TL for further details and applications of the formalism.

Table 3.4 Existential path quantifier/ "some" branching operator (∃) in CTL
p and q are specific propositional predicates on the system variables. ϕ is an arbitrary propositional predicate, which may include p or q unless otherwise stated.
In each example, the root of the tree presented is considered the state of interest.

Examples	Meanings	Sample Computation Trees
$\exists \Diamond p$	This requires that $\Diamond p$ holds in *some* paths of executions starting from the state of interest. We present a sample computation tree showing three paths of execution in which the property holds; traversing the tree through the left branch from the root shows a path while two more paths can be found through the middle branch	

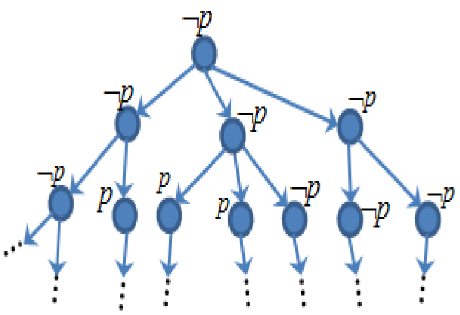
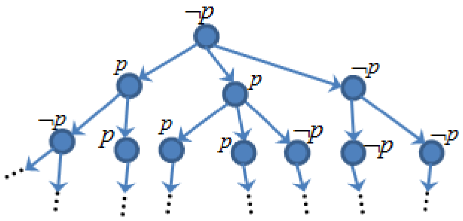
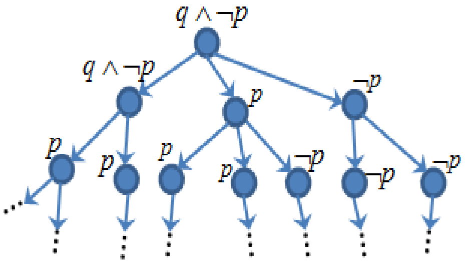
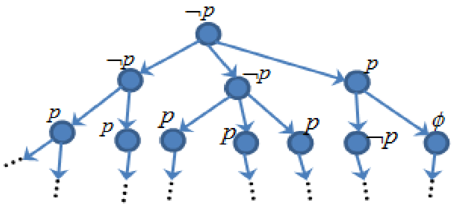
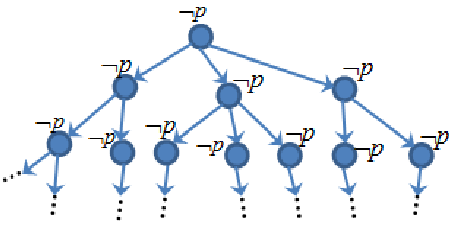
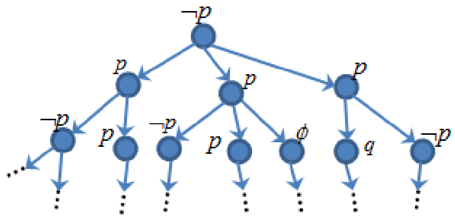
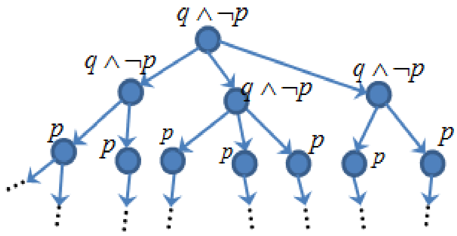
$\exists \Box \neg p$	This specifies that $\Box \neg p$ must hold in *some* paths of computation starting from the state of interest. In the sample computation tree, starting from the root node, four paths of execution satisfy the requirement; one each through the left and middle branches and two through the right branch.	 $\neg p$ $\neg p$ $\neg p$ $\neg p$ $\neg p$ p p p $\neg p$ $\neg p$ $\neg p$
$\exists \circ p$	A specification that $\circ p$ should hold in *some* paths of execution starting from the state of interest. We present a sample computation tree in which the property holds in the execution paths that follow the left and middle branches from the root node.	 $\neg p$ p p $\neg p$ $\neg p$ p p p $\neg p$ $\neg p$ $\neg p$
$\exists q \mathsf{U} p$	This requires that the property $q \mathsf{U} p$ holds in *some* paths of execution starting from the state of interest. Taking the root node as the state of interest in the computation tree in the next cell, the requirement will be satisfied when traversing through the left branch of the tree.	 $q \wedge \neg p$ $q \wedge \neg p$ p $\neg p$ p p p p $\neg p$ $\neg p$ $\neg p$

Table 3.5 Universal path quantifier/ "all" branching operator (∀) in CTL

p and q are specific propositional predicates on the system variables. ϕ is an arbitrary propositional predicate, which may include p or q unless otherwise stated.
In each example, the root of the tree presented is considered the state of interest

Examples	Meanings	Sample Computation Trees
$\forall \Diamond p$	This requires that $\Diamond p$ holds in *all* paths of executions starting from the state of interest. As illustrated in the sample computation tree, irrespective of the path of traversing the tree, we eventually arrive at a node (state) where p holds.	 $\neg p$ $\neg p$ $\neg p$ p p p p p p $\neg p$ ϕ

$\forall\Box\neg p$	This specifies that $\Box\neg p$ must hold in *all* paths of computation starting from the state of interest. In the sample computation tree, starting from the root node, all paths traversing the tree satisfy the requirement.	 ¬p ¬p ¬p ¬p ¬p ¬p ¬p ¬p ¬p ¬p ¬p
$\forall\circ p$	A specification that $\circ p$ should hold in *all* paths of execution starting from the state of interest. The sample computation tree presented illustrates that the property must hold in all branches starting from the node (state) of interest.	
$\forall q \cup p$	The property $q \cup p$ must hold in *all* paths of execution starting from the state of interest. We illustrate this with the computation tree in the next cell; irrespective of the path followed in traversing the tree from the root node, the property should hold.	 $q \wedge \neg p$ $q \wedge \neg p$ $q \wedge \neg p$ $q \wedge \neg p$ p p p p p p p

3.2.6.3 *Property patterns in TL*

There exist tools, such as NuSMV [CCG+99, CCG+00], NuSMV2 [CCG+02], TSMV [MS04], SPIN [Hol97],TLAPS[CDL+10, CDL+12], and TaLiRo [FP08, ALF+11, FSU+12]to automate the rigorous verification of temporal properties of systems such that the user does not necessarily need to have the knowledge of the internal mechanism of the tools. The user needs, however, to be able to specify, correctly, the properties to be verified in the specification formalism supported by the chosen verification tool; one of such formalisms is the TL, which we have just introduced. It is a common knowledge that dealing with such formalism is usually non-trivial; it takes some level of expertise in handling the idioms of logic and discrete mathematics to correctly read and/or write complex requirement properties. Lack of this expertise has been widely acknowledged by FM researchers as one of the main inhibitors to the wide adoption of formal verification tools, and as a consequent, challenging the translation of research outputs in FM into practice.

In an effort to proffer a solution to this problem, Dwyer, Avrunin and Corbett [DAC98, DAC99] hypothesized that the experience base of experts in specification formalisms could be captured in parameterized patterns in formalism-independent formats to allow for systematic mapping to equivalent representations in some known specification formalism. They argued that this could be an easy way to transfer the experiences of experts in the domain to emerging practitioners and potential users.

Dwyer et al. were inspired by the successes that had been recorded with the use of design patterns to provide guidance on the best ways to language features to solve recurring problems by documenting tested solutions to such problems in patterns that can be easily reused to solve similar problems. With this they envisioned the success of a pattern-based approach to formal specification of properties of finite state systems for verification. The output of their research was the recognition of some commonly occurring requirement property patterns from a collection of over five hundred property specifications they collected about thirty-five sources comprising academia and industry. Based on their findings, Dwyer et al. proposed parameterized templates for the recognized property patterns in five property specification formalisms: LTL, CTL, QRE (Quantified Regular Expression) [DC94], GIL (Graphical Interval Logic) [Mel88]and INCA queries [CA95]. Some other researchers have later reproduced the templates in Action CTL (ACTL) [Fer94] and μ-calculus [Koz83] while the patterns are gaining popularity among FM practitioners.

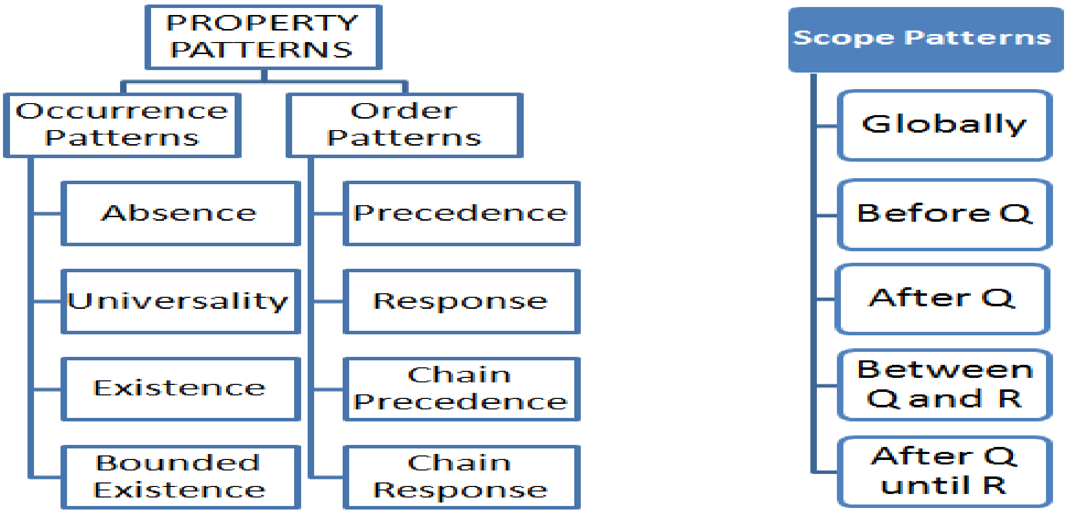


Figure3.16 Temporal property specification patterns [DAC98, DAC98]
Q and R are parameters representing some temporal locations in the trace of a system.

Figure3.16 presents the patterns of temporal property specification (on the left) due to Dwyer et al. and the possible scopes (on the right) of such patterns in execution of a system. The satisfaction or otherwise of the property specified with any of the pattern is checked within the specified scope. "Globally" scope, as the name implies, specifies that a property should hold throughout the execution. "Before Q" (resp. "After Q") scope specifies that a given property must hold before (resp. after) the occurrence of a specified state/event Q. "Between Q and R" implies that a given property must hold after the occurrence of state/event Q and before R where it is certain that R will eventually occur. "After Q until R" has a similar implication with "Between Q and R" except that, in the former, it is not certain whether R will occur or not.

The property patterns (see left of Figure3.16) are classified into two categories: *occurrence* and *order* to describe properties on the occurrences or non-occurrence of states/events and relative order of states/events respectively within the segment of execution defined by the associated scopes.Table 3.6 presents brief descriptions of the intents of the different patterns.

Table 3.6 Intents of the temporal property patterns of Dwyer et al

Pattern category	Pattern	Description
Occurrence patterns	Absence	It specifies states or events that must never occur within the specified scope
	Universality	It specifies states or events that must continuously occur within the specified scope
	Existence	It specifies states or events that must eventually occur (i.e., at least once) within the specified scope
	Bounded existence	It specifies the maximum possible number of occurrences of certain states or events within the specified scope
Order patterns	Precedence	It specifies a cause and effect relationship between two states or events such that the occurrence of one must always have been preceded by the occurrence of the other within the specified scope
	Response	It specifies a stimulus and response relationship between two states or events such that the occurrence of one must always eventually be followed by the occurrence of the other within the specified scope
	Chain precedence	It specifies a variant of the precedence pattern with m-cause to n-effect where $m, n \in \mathbb{N}$; e.g., 1-cause to 2-effects, 2-cause to 1-effect.
	Chain response	It specifies a variant of the response pattern with m-stimulus to n-response where $m, n \in \mathbb{N}$; e.g., 1-stimulust to 2-response, 2-stimulus to 1-response.

Dwyer et al. have provided the templates, based on seven property specification formalisms, for the property patterns on the project's website[9]; for the purpose of our discussion in this thesis, we only the equivalent LTL and CTL formulae or templates for the occurrence and order patterns in Table 3.7 and Table 3.8respectively.

For each template pattern, variables p, q, r, s and t are parameters to be replaced with user-defined predicates on the system variables when instantiated. Each formula can be matched to a domain problem by combining the syntax of the property pattern (see first column of each table) with an appropriate scope pattern (in second column). For instance, the LTL formula for the absence property "*p is false globally*" is $\square(\neg p)$and the CTL formula for existence property "*p becomes true between q and r*" is $\forall\square(q\wedge\neg r\Rightarrow \forall[\neg r\,\mathsf{w}\,(p\wedge\neg r)])$.

Table 3.7 LTL and CTL templates for occurrence property patterns

Occurrence Property Patterns			
Patterns	**Scopes**	**LTL Specifications**	**CTL Specifications**
Absence Syntax: *p*is false	Globally	$\square(\neg p)$	$\forall\square(\neg p)$
	Before *r*	$\Diamond r \Rightarrow (\neg\, p \,\mathsf{U}\, r)$	$\forall[(\neg p\vee \forall\square(\neg r))\ \mathsf{w}\ r]$[10]
	After *q*	$\square(q\Rightarrow\square(\neg p))$	$\forall\square(q \Rightarrow \forall\square(\neg p))$
	Between *q*and *r*	$\square((q\wedge\neg r\wedge\Diamond r) \Rightarrow(\neg p\,\mathsf{U}\, r))$	$\forall\square(q\wedge\neg r\Rightarrow \forall[(\neg p\vee \forall\square(\neg r))\ \mathsf{w}\, r])$
	After *q* until *r*	$\square((q\wedge\neg r) \Rightarrow(\neg p\,\mathsf{w}\ r))$	$\forall\square(q\wedge\neg r\Rightarrow \forall[\neg p\,\mathsf{w}\, r])$
Existence Syntax: *p* becomes true	Globally	$\Diamond p$	$\forall\Diamond p$
	Before *r*	$\neg r\,\mathsf{w}\,(p\wedge\neg r)$	$\forall[\neg r\,\mathsf{w}\,(p\wedge\neg r)]$
	After *q*	$\square(\neg q) \vee\Diamond(q\wedge\Diamond p))$	$\forall[\neg q\,\mathsf{w}\,(q\wedge \forall\square(p))]$

[9] http://patterns.projects.cs.ksu.edu/documentation/patterns.shtml, last accessed August 23, 2016

[10] w is the weak until operator which may be related to until, U, operator using any of the following equivalences: $p\,\mathsf{W}q = (\square p) \vee (p\mathsf{U}q)$ or $p\,\mathsf{W}q = \Diamond(\neg p) \Rightarrow (p\ \mathsf{U}q)$ or $p\,\mathsf{W}q = p\ \mathsf{U}\ (q \vee\square p)$

	Between *q* and *r*	$\Box(q\wedge\neg r\Rightarrow(\neg r\,\mathcal{W}\,(p\wedge\neg r)))$	$\forall\Box(q\wedge\neg r\Rightarrow\forall[\neg r\,\mathcal{W}\,(p\wedge\neg r)])$
	After *q* until *r*	$\Box(q\wedge\neg r\Rightarrow(\neg r\,\mathsf{U}\,(p\wedge\neg r)))$	$\forall\Box(q\wedge\neg r\Rightarrow\forall[\neg r\,\mathsf{U}\,(p\wedge\neg r)])$
Bounded Existence Syntax: *p*-states occur at most *n* times[11]	Globally	$(\neg p\,\mathcal{W}\,(p\,\mathcal{W}\,(\neg p\,\mathcal{W}\,(p\,\mathcal{W}\Box\neg p))))$	$\neg\exists\Diamond(\neg p\wedge\exists\circ(p\wedge\exists\Diamond(\neg p\wedge\exists\circ(p\wedge\exists\Diamond(\neg p\wedge\exists\circ(p)))))$
	Before *r*	$\Diamond r\Rightarrow((\neg p\wedge\neg r)\,\mathsf{U}\,(r\vee((p\wedge\neg r)\,\mathsf{U}(r\vee((\neg p\wedge\neg r)\,\mathsf{U}\,(r\vee((p\wedge\neg r)\,\mathsf{U}(r\vee(\neg p\,\mathsf{U}\,r)))))))))$	$\neg\exists[\neg r\,\mathsf{U}\,(\neg p\wedge\neg r\wedge\exists\circ(p\wedge \mathrm{E}[\neg r\,\mathsf{U}\,(\neg p\wedge\neg r\wedge\exists\circ(p\wedge\exists[\neg r\,\mathsf{U}\,(\neg p\wedge\neg r\wedge\exists\circ(p\wedge\neg r))]))]))]$
	After *q*	$\Diamond q\Rightarrow(\neg q\,\mathsf{U}\,(q\wedge(\neg p\,\mathcal{W}\,(p\,\mathcal{W}\,(\neg p\,\mathcal{W}\,(p\,\mathcal{W}\,\Box\neg p)))))$	$\neg\exists[\neg q\,\mathsf{U}\,(q\wedge\exists\Diamond(\neg p\wedge\exists\circ(p\wedge\exists\Diamond(\neg p\wedge\exists\circ(p\wedge\exists\Diamond(\neg p\wedge\exists\circ(p))))))]$
	Between *q* and *r*	$\Box((q\wedge\Diamond r)\Rightarrow((\neg p\wedge\neg r)\,\mathsf{U}\,(r\vee((p\wedge\neg p\mathsf{U}(r\vee((\neg p\wedge\neg r)\,\mathsf{U}(r\vee((p\wedge\neg r)\,\mathsf{U}(r\vee(\neg p\mathsf{U}r)))))))))))$	$\forall\Box(q\Rightarrow\neg\exists[\neg r\,\mathsf{U}\,(\neg p\wedge\neg r\wedge \mathrm{E}\circ(p\wedge\exists[\neg r\,\mathsf{U}\,(\neg p\wedge\neg r\wedge\exists\circ(p\wedge\exists[\neg r\,\mathsf{U}\,(\neg p\wedge\neg r\wedge\exists\circ(p\wedge\neg r\wedge\exists\Diamond(r)))]))]))])$
	After *q* until *r*	$\Box(q\Rightarrow((\neg p\wedge\neg r)\,\mathsf{U}\,(r\vee((p\wedge\neg r)\mathsf{U}(r\vee((\neg p\wedge\neg r)\,\mathsf{U}\,(r\vee((pf\neg r)\,\mathsf{U}(r\vee(\neg p\,\mathcal{W}r)\,\varpi\Box p)))))))))$	$\forall\Box(q\Rightarrow\neg\exists[\neg r\,\mathsf{U}\,(\neg p\wedge\neg r\wedge\exists\circ(pf\exists[\neg r\,\mathsf{U}\,(\neg p\wedge\neg r\wedge\exists\circ(p\wedge\exists[\neg r\,\mathsf{U}\,(\neg p\wedge\neg r\wedge\exists\circ(p\wedge\neg r))]))]))])$
Universality Syntax: *p*is true	Globally	$\Box p$	$\forall\Box(p)$
	Before *r*	$\Diamond r\Rightarrow(p\mathsf{U}r)$	$\forall[(p\vee\forall\Box(\neg r))\,\mathcal{W}r]$
	After *q*	$\Box(q\Rightarrow\Box(p))$	$\forall\Box(q\Rightarrow\forall\Box(p))$
	Between *q* and *r*	$\Box((q\wedge\neg r\wedge\Diamond r)\Rightarrow(p\mathsf{U}r))$	$\forall\Box(q\wedge\neg r\Rightarrow\forall[(p\vee\forall\Box(\neg r))\,\mathcal{W}r])$
	After *q* until *r*	$\Box(q\wedge\neg r\Rightarrow(p\,\mathcal{W}r))$	$\forall\Box(q\wedge\neg r\Rightarrow\forall[p\,\mathcal{W}r])$

[11] n = 2 for the formulae presented in this table; variation on the mapping is required to specify other bounds

Table 3.8 LTL and CTL templates for order property patterns

Order Property Patterns			
Patterns	**Scopes**	**LTL Specifications**	**CTL Specifications**
Precedence Syntax: *s* precedes *p*	Globally	$\neg p \mathcal{W} s$	$\forall[\neg p \mathcal{W} s]$
	Before *r*	$\Diamond r \Rightarrow (\neg p \mathsf{U} (s \vee r))$	$\forall[(\neg p \vee \forall\Box(\neg r))\ \mathcal{W}\ (s \vee r)]$
	After *q*	$\Box\neg q \vee \Diamond(q f\ (\neg p \mathcal{W} s))$	$\forall[\neg q \mathcal{W} (q \wedge \forall[\neg p \mathcal{W} s])]$
	Between *q* and *r*	$\Box((q \wedge \neg r \wedge \Diamond r) \Rightarrow (\neg p \mathsf{U} (s \vee r)))$	$\forall\Box(q \wedge \neg r \Rightarrow \forall[(\neg p \vee \forall\Box(\neg r))\ \mathcal{W}\ (s \vee r)])$
	After *q* until *r*	$\Box(q \wedge \neg r \Rightarrow (\neg p \mathcal{W} (s \vee r)))$	$\forall\Box(q \wedge \neg r \Rightarrow \forall[\neg p \mathcal{W} (s \vee r)])$
Response Syntax: *s* responds to *p*	Globally	$\Box(p \Rightarrow \Diamond s)$	$\forall\Box(p \Rightarrow \forall\Diamond(s))$
	Before *r*	$\Diamond r \Rightarrow (p \Rightarrow (\neg r \mathsf{U} (s \wedge \neg r))) \mathsf{U} r$	$\forall[((p \Rightarrow \forall[\neg r \mathsf{U} (s \wedge \neg r)]) \vee \forall\Box(\neg r)) \mathcal{W} r]$
	After *q*	$\Box\ (q \Rightarrow \Box\ (p \Rightarrow \Diamond s)\ )$	$\forall[\neg q \mathcal{W} (q \wedge \forall\Box(p \Rightarrow \forall\Diamond(s))]$
	Between *q* and *r*	$\Box((q \wedge \neg r \wedge \Diamond r) \Rightarrow (p \Rightarrow (\neg r \mathsf{U} (s \wedge \neg r))) \mathsf{U} r)$	$\forall\Box(q \wedge \neg r \Rightarrow \forall[((p \Rightarrow \forall[\neg r \mathsf{U} (s \wedge \neg r)]) \vee \forall\Box(\neg r))\ \mathcal{W} r])$
	After *q* until *r*	$\Box(q \wedge \neg r \Rightarrow ((p \Rightarrow (\neg r \mathsf{U} (s \wedge \neg r))) \mathcal{W} r)$	$\forall\Box(q \wedge \neg r \Rightarrow \forall[(p \Rightarrow \forall[\neg r \mathsf{U} (s \wedge \neg r)]) \mathcal{W} r])$
Precedence chain Syntax: *p* precedes *s*, *t*	Globally	$(\Diamond(s \wedge \circ\Diamond \mathrm{T})) \Rightarrow ((\neg s) \mathsf{U} p))$	$\neg\exists[\neg p \mathsf{U} (s \wedge \neg p \wedge \exists\circ(\exists\Diamond(t)))]$
	Before *r*	$\Diamond r \Rightarrow ((\neg(s \wedge (\neg r) \wedge \circ(\neg r \mathsf{U} (t \wedge \neg r)))) \mathsf{U} (r \vee p))$	$\neg\exists[(\neg p \wedge \neg r) \mathsf{U} (s \wedge \neg p \wedge \neg r \wedge \exists\circ(\exists[\neg r \mathsf{U} (t \wedge \neg r)]))]$
	After *q*	$(\Box\neg q) \vee ((\neg q) \mathsf{U} (q \wedge ((\Diamond(s \wedge \circ\Diamond t)) \Rightarrow ((\neg s) \mathsf{U} p)))$	$\neg\exists[\neg q \mathsf{U} (q \wedge \exists[\neg p \mathsf{U} (s \wedge \neg p \wedge \exists\circ(\exists\Diamond(t)))])]$
	Between *q* and *r*	$\Box((q \wedge \Diamond r) \Rightarrow ((\neg(s \wedge (\neg r) \wedge \circ(\neg r \mathsf{U} (t \wedge \neg r)))) \mathsf{U} (r \vee p)))$	$\forall\Box(q \Rightarrow \neg\exists[(\neg p \wedge \neg r) \mathsf{U} (s \wedge \neg p \wedge \neg r \wedge \exists\circ(\exists[\neg r \mathsf{U} (t \wedge \neg r \wedge \exists\Diamond(r))]))])$
	After *q*	$\Box(q \Rightarrow (\neg(s \wedge (\neg r) \wedge \circ(\neg r \mathsf{U} (t \wedge$	$\forall\Box(q \Rightarrow \neg\exists[(\neg p \wedge \neg r) \mathsf{U}(s \wedge \neg p \wedge$

	until *r*	$\neg r))) \mathsf{U}(r\vee p) \vee\Box(\neg(s\wedge\circ\Diamond t))))$	$\neg r\wedge \exists\circ(\exists[\neg r\mathsf{U} (t\wedge \neg r)]))])$
Precedence chain Syntax: *s*, *t* precedes *p*	Globally	$\Diamond p\Rightarrow (\neg p\mathsf{U}(s\wedge \neg p\wedge\circ(\neg p\mathsf{U}t)))$	$\neg\exists[\neg s\mathsf{U}p] \wedge \neg\exists[\neg p\mathsf{U} (s\wedge \neg p\wedge \exists\circ(\exists[\neg t\mathsf{U} (p\wedge \neg t)]))]$
	Before *r*	$\Diamond r\Rightarrow (\neg p\mathsf{U} (r\vee (s\wedge \neg p\wedge\circ(\neg p\mathsf{U}t))))$	$\neg\exists[(\neg s\wedge\neg r) \mathsf{U} (p\wedge \neg r)] \wedge\neg\exists[(\neg p\wedge \neg r) \mathsf{U} (s\wedge \neg p\wedge \neg r\wedge \exists\circ(\exists[(\neg t\wedge \neg r) \mathsf{U} (p\wedge\neg t\wedge \neg r)]))]$
	After *q*	$(\Box\neg q) \vee (\neg q\mathsf{U}(q\wedge\Diamond p\Rightarrow (\neg p\mathsf{U} (s\wedge \neg p\wedge\circ(\neg p\mathsf{U}t))))$	$\neg\exists[\neg q\mathsf{U}(q\wedge \exists[\neg s\mathsf{U}p] \wedge \exists[\neg p\mathsf{U}(s\wedge \neg p\wedge \exists\circ(\exists[\neg t\mathsf{U} (p\wedge \neg t)]))])]$
	Between *q* and *r*	$\Box((q\wedge\Diamond r) \Rightarrow (\neg p\mathsf{U} (r\vee(s\wedge\neg p\wedge\circ(\neg p\mathsf{U}t)))))$	$\forall\Box(q\Rightarrow \neg\exists[(\neg s\wedge \neg r)\mathsf{U}(p\wedge\neg r\wedge \exists\Diamond(r))] \wedge\neg\exists[(\neg p\wedge\neg r)\mathsf{U}(s\wedge\neg p\wedge \neg r\wedge \exists\circ(\exists[(\neg t\wedge\neg r)\mathsf{U}(p\wedge\neg t\wedge \neg r\wedge \exists\Diamond(r))]))])$
	After *q* until *r*	$\Box(q\Rightarrow (\Diamond p\Rightarrow (\neg p\mathsf{U} (r\vee (s\wedge \neg p\wedge\circ(\neg p\mathsf{U}t))))))$	$\forall\Box(q\Rightarrow \neg\exists[(\neg s\wedge \neg r)\mathsf{U}(p\wedge \neg r)] \wedge \neg\exists[(\neg p\wedge \neg r) \mathsf{U} (s\wedge \neg p\wedge \neg r\wedge \exists\circ(\exists[(\neg t\wedge\neg r) \mathsf{U} (p\wedge\neg t\wedge \neg r)]))])$
Response chain Syntax: *s*, *t* respond to *p*	Globally	$\Box(p\Rightarrow\Diamond(s\wedge\circ\Diamond t))$	$\forall\Box(p\Rightarrow \forall\Diamond(s\wedge \forall\circ(\forall\Diamond(t))))$
	Before *r*	$\Diamond r\Rightarrow (p\Rightarrow (\neg r\mathsf{U}(s\wedge \neg r\wedge\circ(\neg r\mathsf{U}t)))) \mathsf{U}r$	$\neg\exists[\neg r\mathsf{U} (p\wedge \neg r\wedge (\exists[\neg s\mathsf{U} r] \vee\exists[\neg r\mathsf{U} (s\wedge \neg r\wedge \exists\circ(\exists[\neg t\mathsf{U}r]))]))]$
	After *q*	$\Box(q\Rightarrow\Box(p\Rightarrow (s\wedge\circ\Diamond t)))$	$\neg\exists[\neg q\mathsf{U} (q\wedge \exists\Diamond(p\wedge(\exists\Box(\neg s) \vee \exists\Diamond(s\wedge \exists\circ(\exists\Box(\neg t))))))]$
	Between *q* and *r*	$\Box((q\wedge\Diamond r) \Rightarrow (p\Rightarrow (\neg r\mathsf{U} (s\wedge \neg r\wedge\circ(\neg r\mathsf{U}t)))) \mathsf{U}r)$	$\forall\Box(q\Rightarrow \neg\exists[\neg r\mathsf{U} (p\wedge \neg r\wedge (\exists[\neg s\mathsf{U}r] \vee \exists[\neg r\mathsf{U} (s\wedge\neg r\wedge \exists\circ(\exists[\neg t\mathsf{U}r]))]))])$
	After *q* until *r*	$\Box(q\Rightarrow (p\Rightarrow (\neg r\mathsf{U} (s\wedge \neg r\wedge\circ(\neg r\mathsf{U}t)))) \mathsf{U} (r\vee\Box(p\Rightarrow (s\wedge\circ\Diamond t))))$	$\forall\Box(q \Rightarrow \neg\exists[\neg r\mathsf{U} (p\wedge\neg r\wedge(\exists[\neg s\mathsf{U}r] \vee \exists\Box(\neg s\wedge\neg r) \vee \exists[\neg r\mathsf{U} (s\wedge\neg r\wedge \exists\circ(\exists[\neg t\mathsf{U}r] \vee \exists\Box(\neg t\wedge \neg r)))]))])$
Response chain Syntax:	Globally	$\Box(s\wedge\circ\Diamond t\Rightarrow\circ(\Diamond(t\wedge\Diamond p)))$	$\neg\exists\Diamond(s\wedge \exists\circ(\exists\Diamond(t\wedge \exists\Box(\neg p))))$
	Before *r*	$\Diamond r\Rightarrow (s\wedge\circ(\neg r\mathsf{U}t) \Rightarrow\circ(\neg r\mathsf{U} (t\wedge\Diamond p))) \mathsf{U}r$	$\neg\exists[\neg r\mathsf{U} (s\wedge\neg r\wedge \exists\circ(\exists[\neg r\mathsf{U}(t\wedge \neg r\wedge \exists[\neg p\mathsf{U}r])]))]$

p responds to s, t	After q	$\Box(q\Rightarrow\Box(s\wedge\circ\Diamond t\Rightarrow\circ(\neg t\,\mathsf{U}(t\wedge\Diamond p))))$	$\neg\exists[\neg q\,\mathsf{U}(q\wedge\exists\Diamond(s\wedge\exists\circ(\exists\Diamond(t\wedge\exists\Box(\neg p)))))]$
	Between q and r	$\Box((q\wedge\Diamond r)\Rightarrow(s\wedge\circ(\neg r\,\mathsf{U}t)\Rightarrow\circ(\neg r\,\mathsf{U}(t\wedge\Diamond p)))\,\mathsf{U}r)$	$\forall\Box(q\Rightarrow\neg\exists[\neg r\,\mathsf{U}(s\wedge\neg r\wedge\exists\circ(\exists[\neg r\,\mathsf{U}(t\wedge\neg r\wedge\exists[\neg p\,\mathsf{U}r])]))])$
	After q until r	$\Box(q\Rightarrow(s\wedge\circ(\neg r\,\mathsf{U}t)\Rightarrow\circ(\neg r\,\mathsf{U}(t\wedge\Diamond p)))\,\mathsf{U}(r\vee\Box(s\wedge\circ(\neg r\,\mathsf{U}t)\Rightarrow\circ(\neg r\,\mathsf{U}(t\wedge\Diamond p)))))$	$\forall\Box(q\Rightarrow\neg\exists[\neg r\,\mathsf{U}(s\wedge\neg r\wedge\exists\circ(\exists[\neg r\,\mathsf{U}(t\wedge\neg r\wedge(\exists[\neg p\,\mathsf{U}r]\vee\exists\Box(\neg p\wedge\neg r)))]))])$

3.2.6.4 Specification of the BVM's design requirements based on the TL property patterns

We present in this subs-subsection the specification of the required properties of the BVM, which were stated in the synopsis of our running example in Section 3.2.1.We will reuse the system variables declared in the Z specification (see Section 3.2.4) to specify the requirement properties. The variables are *vault*, *escrow*, *credit*, *badC*, *price*, *current* and ϕ.

The required properties were stated in natural language as follows:

I. BVM must not dispense unless enough coins are inserted to pay for the selected drink

We can rephrase this property to match the "precedence chain" property pattern as:

The selection of a drink, and acquisition of sufficient coins always precede the dispense of selected drink.

This statement matches with the property pattern "*s, t, precede p globally"* where:

s = "a drink is selected", t = "sufficient coins have been acquired" and p = "selected drink is dispensed". From the Z and DEVS specifications of BVM, we know that $current \in \{1,2,3,4\}$ when a drink has been ordered. $credit \geq price$ when sufficient coins have been cumulated for a transaction and $\phi = dispense$ must be true for BVM to deliver a drink to the user. For $\phi = dispense$ to be true, predicate $(current \in \{1,2,3,4\} \wedge price > 0 \wedge credit = price)$ must hold. The parameterized LTL template of the pattern is given in Table 3.8 is:

$$\Diamond p\Rightarrow(\neg p\,\mathsf{U}\,(s\wedge\neg p\wedge\circ(\neg p\,\mathsf{U}t)))$$

By substituting $current \in \{1,2,3,4\}$ for s, $credit \geq price$ for t and $current \in \{1,2,3,4\} \wedge price > 0 \wedge credit = price$ for p, the LTL specification of the property is:

$$\Diamond(current \in \{1,2,3,4\} \wedge price > 0 \wedge credit = price) \Rightarrow(\neg(current \in \{1,2,3,4\} \wedge price > 0 \wedge credit = price)\,\mathsf{U}\,(current \in \{1,2,3,4\} \wedge \neg(current \in \{1,2,3,4\} \wedge price > 0 \wedge$$

$$credit = price) \wedge \circ(\neg(current \in \{1, 2, 3, 4\} \wedge price > 0 \wedge credit = price) \mathsf{U} credit \geq price)))$$

II. BVM should always refund the balance whenever excess coins are inserted, i.e., when the amount inserted is greater than the price of the selected drink.

We rephrase this requirement to match the response pattern as:

Refund balance responds to excess payments always

This statement matches with the pattern "*s responds to p globally"* where s = "refund balance occurs" and p = "excess coins have been inserted".

We know that excess coins have been inserted when $credit > price$ and refund of balance of transaction has occurred when *credit* reduces to the value of *price*, i.e., $credit = price$. The LTL and CTL templates for the pattern are $\square(p \Rightarrow \lozenge s)$ and $\forall\square(p \Rightarrow \forall\lozenge(s))$ respectively. By substituting $credit > price$ for p and $credit = price$ for s, the LTL specification for this property is:

$$\square(credit > price \Rightarrow \lozenge\, credit = price).$$

And the CTL specification will be:

$$\forall\square(credit > price \Rightarrow \forall\lozenge(credit = price))).$$

III. Once the payment for a drink is complete, the transaction cannot be canceled any longer

We can rephrase this property to match with the absence property pattern as follows:

Transaction is canceled is not allowed after sufficient coins have been acquired for the transaction.

This matches with the occurrence pattern *p is false after q* with p = "transaction is canceled" and q = "sufficient coins have been acquired for the transaction".

Using the system variables, we know that transaction is canceled when *current* = 5 and sufficient coins have been acquired when $credit \geq price$. The LTL and CTL templates for the pattern are $\square(q \Rightarrow \square(\neg p))$ and $\forall\square(q \Rightarrow \forall\square\ (\neg p))$ respectively. Substituting $current\ =\ 5$ for p and $credit \geq price$ for q, we have the LTL specification of the property as:

$$\square(credit \geq price \Rightarrow \square(\neg(current\ =\ 5)))$$

And the CTL specification is:

$$\forall\square(credit \geq price\ \Rightarrow \forall\square\ (\neg(current\ =\ 5)))$$

3.3 MODEL-DRIVEN ENGINEERING

Model-Driven Engineering (MDE) [Ken02, AK03, Fav04, Béz04, Béz06, Sch06, FR07] is a modern Software Engineering approach that promotes the creation of abstract models of software systems and systematically transform them to concrete implementations. MDE approaches consider every artifact of software development as a model in similitude to the way everything is considered an object in Object-Oriented approaches.

The overall goal of MDE is to bridge the wide conceptual gaps between problem and implementation domains using technologies that support automated transformations of problem-level abstractions to software implementations to avoid the accidental complexities that often characterize the manual direct coding of problems [FR07]. Therefore, the vision of the MDE initiative is an ideal world of software engineering in which high-level models can be used, at different phases, to communicate and understand problems, validate design assumptions, verify the satisfaction of design requirements, document software architectures and ultimately, to drive the successive automated synthesis of software artifacts until executable program codes are obtained.

Empirical evidences from surveys conducted by different researchers over the last decade [WW06, MD08, BC10, HWR+11, Sel12, WHR14] suggest that though the adoption of MDE for software engineering processes is not without some challenges, the success stories far outweigh the failures. Some of the potential benefits identified are reduced development time and cost, improved software quality, increased productivity of the development team, and portability of models between solution platforms, improved communication among stakeholders and within development teams.

3.3.1 Model-Driven Architecture

Model-Driven Architecture (MDA™) [KWB03, Mel04, Tru06] is the Object Management Group (OMG)'s framework and standard for the realization of the MDE initiatives. MDA proposes a three-layered architectural framework with standards and technologies to model software from a given conceptual viewpoint with each layer modeling specific concerns of the viewpoint. Recall that we have previously described a viewpoint as a mechanism comprising a domain, language, specifications and methodology to capture and process certain software and systems engineering concern(s) about a system, the information associated with such concern(s) and their relationships [FKN+92, KW07].

Though the author considers that related concerns of many viewpoints may be mapped to the abstract description of MDA's architectural layers, the prominent viewpoint in the literature of MDA is the "abstraction level" of software systems, a modeling technique for focusing on some specific concerns about a system while suppressing all irrelevant detail [Tru06]. In MDA, this viewpoint is usually used to capture three concerns:

a. System's context and requirements
b. System's structure and operational capabilities
c. Details of execution platforms.

These concerns are addressed in separate layers (abstraction layers) of the three-layered MDA framework. They are *computation-independent layer* (CIL) which is concerned with a system's context and requirements without structural or processing details, *platform-independent layer* (PIL) which is concerned with system's operational capabilities and *platform-specific layer* (PSL) which is concerned with specific execution platforms in addition to a system's operational capabilities.

An execution platform or simply *platform* can be described as the specification of an execution environment for a set of models [Mel04]. It is a collection of subsystems or frameworks and technologies that provide a coherent set of functionality through interfaces and usage patterns that clients can use without having any knowledge of their implementation details [Tru06].Examples of platforms are middleware solutions like CORBA, J2EE, Microsoft .NET, operating systems, databases, programming languages like Java, C++, C#, Python, etc. Therefore, platform-independence as applied in the context of MDA is a measure of the degree of the separation of a viewpoint's concern from the features of a platform. Based on this viewpoint, MDA proposes three categories of models:

a. Computational-Independent Model (CIM) is the model of a software system at the CIL. It is also referred to as domain model as it encourages the use of domain vocabularies, which the practitioners are accustomed, for system specifications [Est07].
b. Platform-Independent Model (PIM) is the model at the PIL. A PIM describes system features that are not likely to change from one platform to another [FR07]. i.e., it exhibits as much degree of platform independence as to allow its use with one or more platforms [Tru06, Est07].
c. Platform-Specific Model (PSM) is the model at the PSL. Essentially, a PSM is a combination of a PIM with the necessary details of a platform, usually referred to as Platform Description Model (PDM).

Through this viewpoint, MDA approaches enable modelers to separate essential business concerns from the details of implementation platforms; thereby enhancing efficient solution designs, increased productivity and reduced development time, portability, interoperability and reusability. The bedrocks of MDA are the OMG's standards like the Meta-Object Facility (MOF) [OMG04], which is a subset of the Unified Modeling Language (UML) for specifying the abstract syntax of modeling languages, the UML and model transformation standards and technologies.

3.3.2 Other MDE initiatives

In addition to MDA, there are other industrial standards for the implementation of the MDE initiative such the Microsoft Software Factories [GS03, GS04, War07] and Model-Integrated Computing (MIC) [SK97, Spr04]. While MDA promotes the use of MOF-based languages as a general-purpose modeling language to define models, other implementations of MDE are inclined towards the specification of some domain-specific languages (DSL) [KT08, KT08] to efficiently model the concepts of some considerably narrow domains based on the vocabularies of such domains. Proponents of domain-specific modeling (DSM) argue that DSLs are more efficient than the standard UML in that they allow for further abstraction away from technologies and work at higher levels. Moreover, the possibility of modeling with original domain vocabulary that non-experienced users are already accustomed to offer an added advantage.

(Meta) modeling and transformations between models are the most significant concepts in the MDE paradigm; we present next some backgrounds on these concepts.

3.3.3 (Meta)Modeling

3.3.3.1 Modeling

Several definitions and description of model have been proposed in the literature of software and systems engineering, each highlighting some significant properties and uses of models. The author does not intend to add to the plethora of definitions; rather excerpts of a few descriptions will be considered to explain the context and use of models in this thesis.

According to Seidewitz [Sei03], *a model is a set of statements about some system under study (SUS)*. A model is considered to be a valid representation of an SUS if all the statements in the model are true for the SUS. A model is created to be analyzed as a way of reasoning about the SUS.

Zeigler and colleagues [ZPK00] provide two definitions of a model: 1) as a set of instructions, rules, equations, or constraints for generating input and output behaviors of a system, and 2) as any physical, mathematical, or logical representation of a system, entity, phenomenon, or process. In a similar description by Selic [Sel03], a model may be developed as a foundation to implement a physical system; it may also be derived from an existing system or a system in development to study its behavior. He noted that the ultimate goal of engineering models is to reduce risk by facilitating better understanding of both a complex problem and its potential solutions before undertaking the expenses and effort of a full implementation.

France and Rumpe [FR07] describe a model as *an abstraction of some aspect of a system to serve some particular purposes* where the system being described may or may not exist at the time the model is created. The purpose of a model could be, inter alia, to present a human understandable description of a system or to present information in a form that can be mechanically analyzed.

In a paper of Bézivin and Gerbé [BG01], *a model is a simplification of a system built with an intended goal in mind. The model should be able to answer questions in place of the actual system*. Bézivin [Béz05] describes the act of modeling as the cost-effective use of something that is simpler, safer or cheaper than reality in place of reality for some cognitive purpose. This way, model presents an abstraction of reality as it cannot represent all aspects of reality, thereby paving the way for dealing with the world in a simplified manner, avoiding the complexity, danger and irreversibility of reality.

According to Brown [Bro04], a model provides abstractions of a physical system that allow engineers to reason about that system by ignoring extraneous details while focusing on the relevant ones. He noted that models are essential to understand complex real world systems in all forms of engineering. He identified some possible uses of models, which include predicting system qualities, reasoning about specific properties when aspects of the system are changed, and communicating key system characteristics to its various stakeholders.

In a similar account, Kühne [Küh06] defines a model as an abstraction of a (real or language based) system allowing predictions or inferences to be made.

Lastly, according to Truyen [Tru06], a model is a formal specification of the function, structure and behavior of a system within a given context, and from a specific point of view (or reference point). In this case, a "formal specification" is one that is written in a language which is based on a well-defined syntax and that has a precise semantic meaning associated with each of its constructs.

There are some salient features (non-exhaustive) of a model that can be derived from these definitions:

It is a simplified representation of some aspects of a system, i.e., it abstracts away from details that are not relevant for the purpose of the model.

It serves to document and communicate a system with relevant stakeholders and development teams. Hence, a model should adequately capture the concerns of the relevant stakeholders. This also brings forward the need for the model to be written in a language whose syntax is comprehensible to all relevant stakeholders.

It can serve as a design to prescribe the blueprint for a to-be system. Thus, the language in which the model is written should be expressive enough to capture the stakeholders' concerns.

It has some precise meaning, and hence amenable to manipulation by machine

It is created for a purpose defined by its context; in fact, a correct interpretation of a model is done with respect to its purpose [KW07]

It provides answers to some ingenious questions (through some forms of analysis) which will ordinarily be difficult to answer by intuition

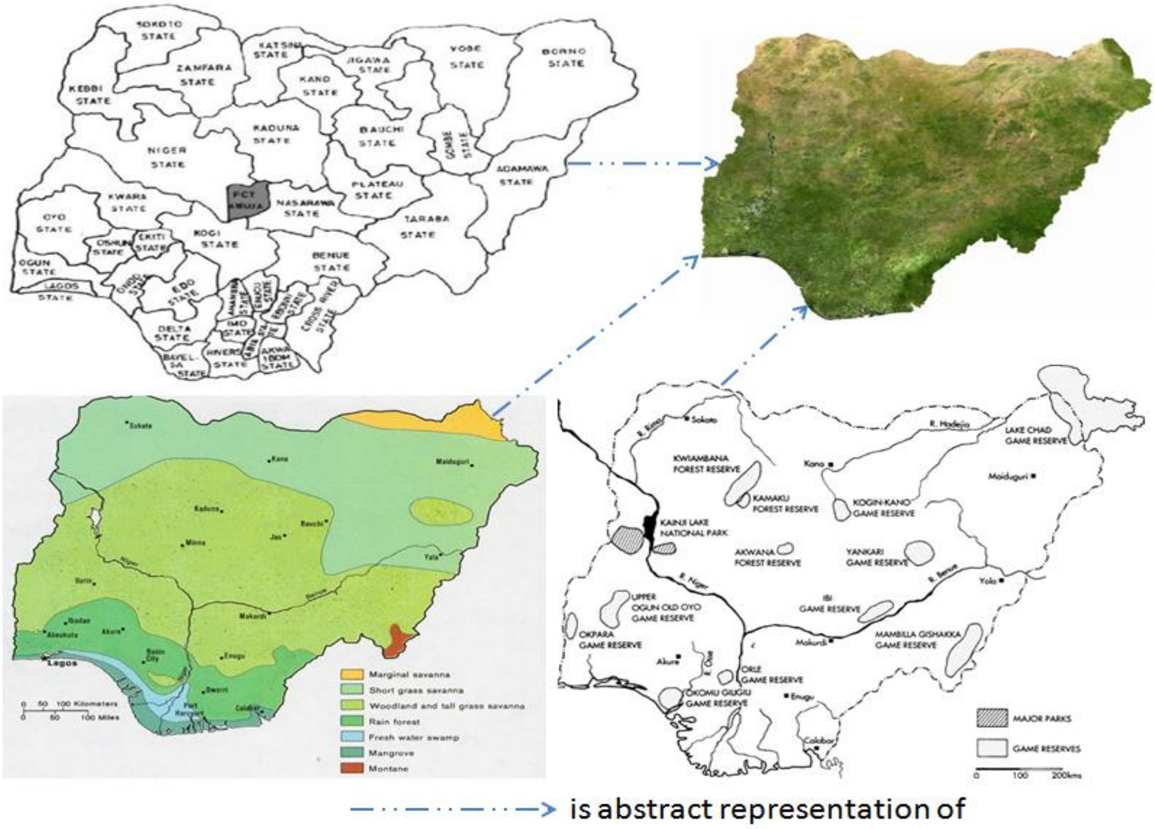


Figure 3.17Model as an abstract representation of a system for a purpose
Each of the three maps is a model representing the entity in the north-east of the figure for specific purposes. The model in the north-west represents the division of the country into 36 administrative units and a central administrative capital; the model in the south-west represents the vegetation in different parts of the country while the model in the south-east represents the locations of major rivers, lakes parks and games reserves.

A complex system may be represented by multiple models serving to answer questions about different aspects of the system. For instance, the north-east region of Figure 3.17 shows an image of the landmass of the country Nigeria. The map on the north-west is a model of the country, which presents the distribution of the landmass into thirty-six independent administrative states and a central administrative capital and abstracts away from all other details about the country.

This model may be useful in providing answers to some complex social-political questions about the country. The map on the south-east region of the figure is a model of the country that presents the major rivers, lakes, parks and games reserves in the country while excluding all other details about the country. This model may be useful to study aspects of the country such as tourist activities, surface water-related activities, etc. Finally, the map in the south-west region of Figure 3.17 is a model that presents the diversity of vegetation of the country. This model may be useful for some agricultural and ecological studies. Several models of like these can be created to study other aspects of the country, e.g., population distribution, locations of mineral resources, industries, institutions, etc. and answering some intricate questions may require the combined analysis of two or more of the models. These different models explain the concept of abstraction in modeling, which is guided by the objective of the model being constructed.

3.3.3.2 Metamodeling

We have described a model in the previous section. Models are written in some modeling languages and the language elements (modeling concepts) used in expressing models are described in meta-models. In other words, *a meta-model is a model of a modeling language* [BJV04, Fav05a]. One of the core tasks in building modeling tools is the modeling of the structure and well-formedness rules of the languages in which the models are expressed; such models are called meta-models [Fav04]. A meta-model defines the kinds of elements that can possibly be contained in a class of models (written in a language) and the valid ways they are related to one another [BJV04]. In other words, a meta-model defines the syntax rules of a language; a model written in that language is considered valid only if it respects these syntax rules. Therefore relationship between a model *M* and the meta-model *MM* that defines the syntax rules of the language in which *M* is written is referred to as *conformance relation* [Fav05a]. In effect, when we say that *M* conforms to *MM,* it implies that *M* respects the syntax rules and constraints specified in *MM.*

Meta-models play a central role in MDE; so they must be precisely defined to ensure the entire MDE process yields the desired result. In addition to being a prerequisite for performing automated model transformations [Fav04], a precise meta-model is instrumental to ascertaining the validity of a model with respect to the domain of the system it represents.

Being a model itself (i.e., a model of a modeling language), a meta-model also must conform to a *meta-meta-model* which models the syntax rules of the language in which the meta-model is written. Intuitively, it can be assumed that this hierarchical model relation, referred to as the "metaization" [GA09] will likely extend infinitely, there is, however, a general understanding that in practice, the relation is always "reflexive" at the meta-meta-model level. i.e., the meta-meta-model specifies the language in which it is written, and hence becomes the meta-model of its own self. Therefore, "meta-meta-model conforms to itself".

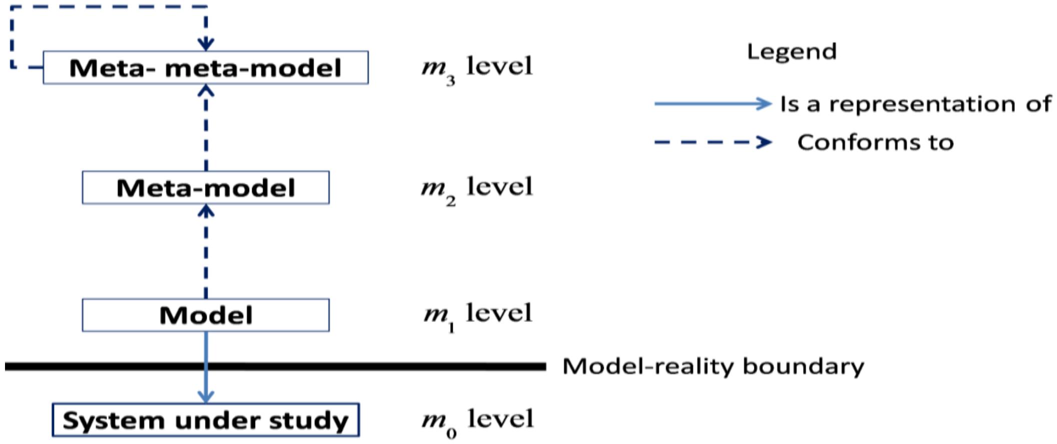


Figure 3.18Metaization viewpoint

Figure 3.18 presents this metaization phenomenon in a four-level architectural framework originally defined by the OMG and sometimes referred to as MDA four-level modeling stack [KBA02]. The *model-reality boundary* line divides the architecture into two spaces: the reality space and the modeling space below and above the boundary line respectively. The two spaces are linked by the "is a representation of" relation from M_1 level to M_0 level. The actual (meta-) modeling activities take place in the modeling space while the M_0 level, in the reality space, contains the system under study. Kühne [Küh06] has provided a formal explanation of the subscripts (0-3) associated with the four levels of the architecture; he related each metaization level to the number of time modeling activities could be "hierarchically" associated with it. In effect, no modeling activity occurs at M_0 level since it only contains the system under study (SUS) to be modeled. M_1 level has one associated modeling activity - modeling the SUS. M_2 models the language with which to perform the task at M_1. Similarly, M_3 models the language that allows the task at M_2 to be performed.

To put this in the perspective of general software and systems engineering, it would be fair to say that this metaization levels define another viewpoint in the MDE paradigm similar to the abstraction level viewpoint discussed previously in the context of MDA. Recall that the abstraction level viewpoint discussed under MDA captures three engineering concerns: system's context and requirements, system's structure and operational capabilities, and details of execution platforms; in a similar standpoint, this metaization viewpoint captures a set of three somewhat more abstract and generic metaization concerns in the (meta)modeling space:

MC 1. Formal specification of the system under study (SUS)

MC 2. Formal specification of the domain of SUS

MC 3. Formal specification of the language rules/infrastructure for defining system domains

Such that metaization concerns MC 1, MC 2 and MC 3 are adequately captured and managed at the M_1, M_2 and M_3 levels respectively.

This metaization viewpoint is inherent in any MDE process. In fact, it provides the infrastructure to precisely define the automated synthesis of some kinds of models from others, the hall mark of MDE paradigm. For instance, it is technically embedded within, and provides support for every layer of MDA. This claim can easily be verified by considering that each of CIM, PIM and PSM is a model of a system, and thus must conform to a meta-model somewhere. i.e., each of the three models reside in the M_1 level of metaization viewpoint and would require other levels to ensure the automated transformations between it and its counterparts in other MDA layers.

Lastly, it is important to also note that this metaization viewpoint is not peculiar to the model ware technological space (TS); rather it is inherent in other TSs. Kurtev and colleagues [KBA02] describe a TS as " *a working context with a set of associated concepts, body of knowledge, tools, required skills, possibilities and possibly a given user community with shared know-how"*. Examples of TS include modelware, grammarware, dataware etc. While modelware, also known as MDA TS refers to model-based development artifacts, grammarware comprise grammars and grammar-dependent software used in the sense of all established grammar formalisms and grammar notations including context-free grammars, class dictionaries, and XML schemas as well as some forms of tree and graph grammars [KLV05]. Dataware refers to the elements used in relational data modeling such as database and database schema.

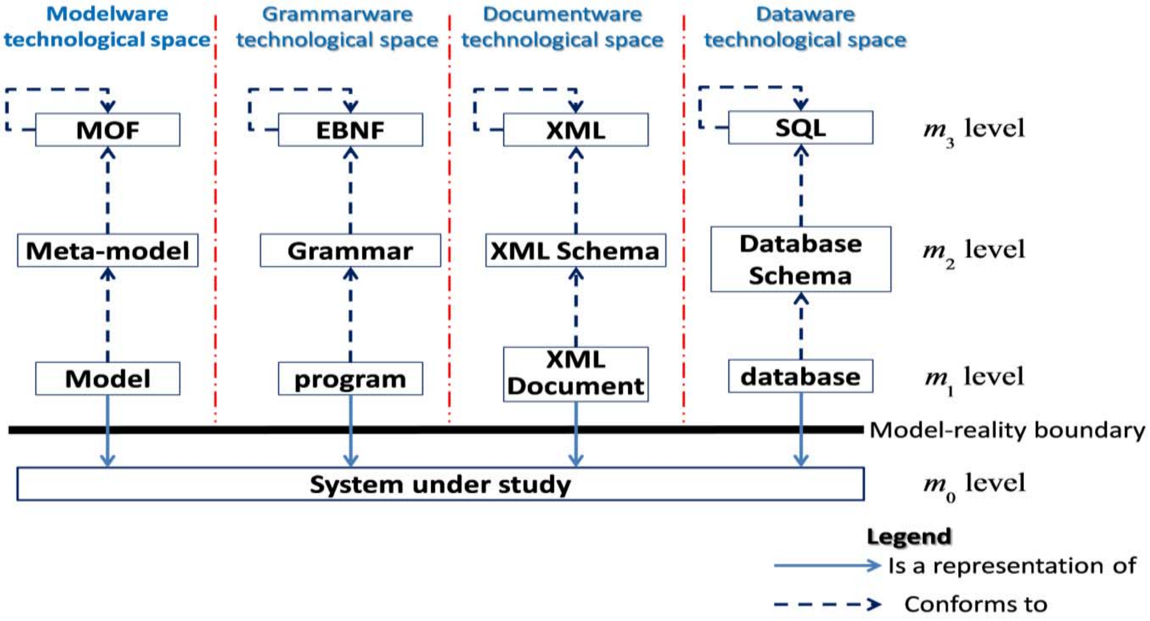


Figure 3.19 Metaization viewpoints of technological spaces

Figure 3.19 presents the metaization viewpoints of four TSs to show the corresponding elements of each TS at the different levels of metaization. Though the documentware is considered to be subsumed by the grammarware, we can, at least introduce it as a special case of grammarware because it will particularly be used in a later chapter of this thesis. The figure is self-explanatory and it shows a common example of meta-meta-model technology in each TS. One of the goals of MDE is to build bridges between these TSs [MH05, WK05, FN05, KBA02]; such bridges are usually defined formally at the M_2 level of metaization by mapping corresponding elements of language specifications.

3.3.3.3 Ecore metamodeling language

There are several languages, based on different technologies, for defining metamodels. Examples include the Ecore, MetaEdit [SLT+91], GME [LMB+01, Dav03], AToM^3 [DV02], KM3 [JB06], etc. We present an overview of the Ecore in this sub-subsection as we will be using it in subsequent chapters to describe metamodels.

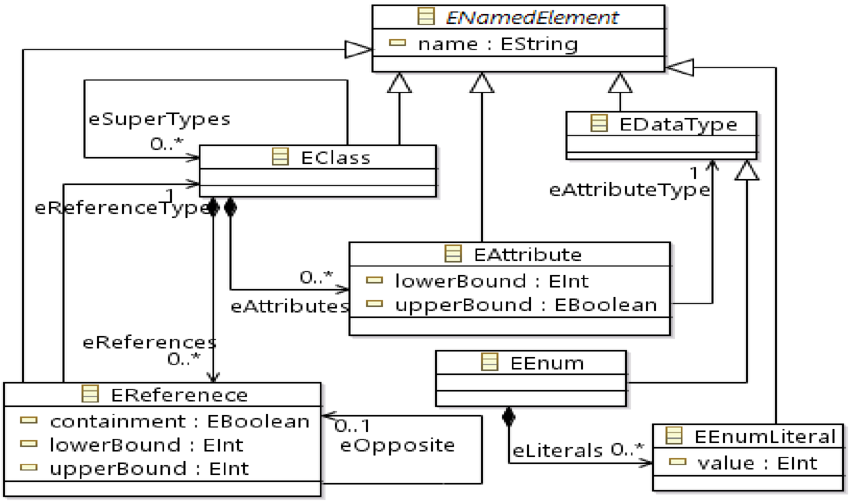


Figure 3.20 A simplified Ecore kernel

The Ecore is a metamodeling language, based on OMG's Essential Meta-Object Facility (EMOF)[12], which uses a subset of the UML class concepts to describe metamodels. Ecore is the underlying metamodeling language for the Eclipse Modeling Framework (EMF) [SBM+08], an

[12] http://www.omg.org/spec/MOF/2.5/

open-source modeling project and code synthesis facility for developing MDE-based tools within the Eclipse IDE[13].

Figure 3.20 shows an excerpt from the Ecore kernel, which is just sufficient to introduce the essential elements that will be used in this thesis; the reader may want to consult some of the textbooks (e.g., [KWB03]) for a detailed description of the language.

ENamedElement is the base class that describes any entity or relationship in a metamodel that has a unique name. Therefore, every metamodel element that could be identified by a name inherits a name attribute from this class. *EClass* describes a class, which models an independent entity. A class may have some attributes (*eAttributes*) and/or references (*eReferences*) describing its structural features and it may *inherit* the properties of some other classes by referring to them as *superTypes*. The *superTypes* relation is transitive; i.e., given three classes A, B and C, B is a *superType* of A and C is a *superType*of B implies that C is a superType of A. Every attribute has a type that is defined by a data type or an Enumeration. The minimum and maximum number of possible occurrences of an eAttribute or eReference in a class is defined by *lowerBound* and *upperBound* respectively; this is known as the *cardinality* of the attribute or reference.

3.3.3.4 Metamodel composition techniques

In MDE, the syntax of a modeling language is defined by a meta-model. In essence, it defines the concepts described in a language and the relationships between them. In this sub-subsection, we give brief descriptions of three techniques, proposed by Emerson and Sztipanovits [ES06], for integrating meta-models to define new languages: metamodel merge, metamodel interfacing and class refinement. These techniques are illustrated in Figure 3.21.

Metamodel merge is used to integrate independent metamodels that share some common abstractions of real world entities - a phenomenon referred to as concept collision. The common concepts are used as the seam(s) to merge the separate metamodels into a unified whole. It is similar to the package merge mechanism [ZDD06] that recursively takes the union of model elements (in different packages) matched by name and meta-type. Meta-model merge is, however, different in two ways: 1) it occurs at class level instead of package level, and 2) common concepts do not necessarily have to match by name in metamodel merge. Once matching classes are identified, the two classes cease to exist but merge into a new class in the integrated metamodel; the new class encompasses all attributes and associations of the source classes.

[13]https://projects.eclipse.org/projects/modeling.emf

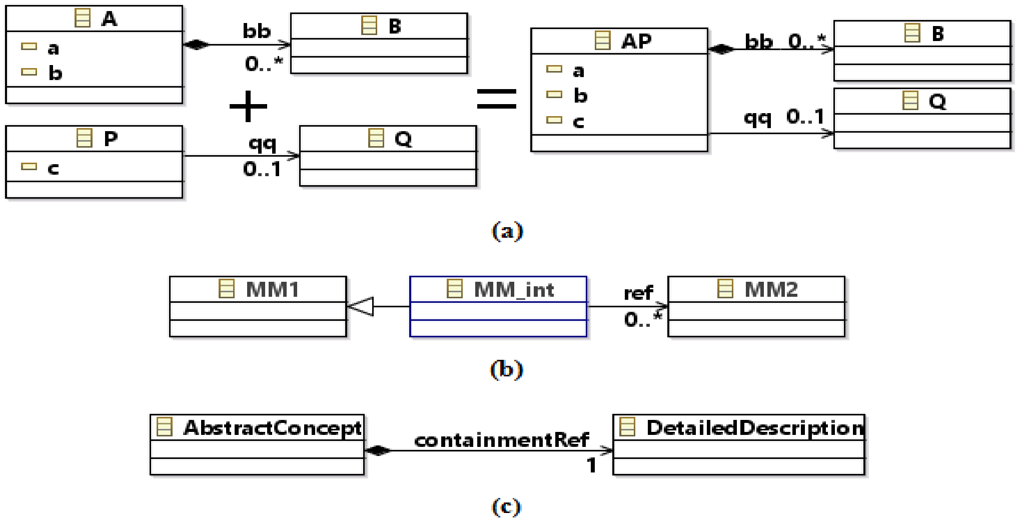


Figure 3.21 Metamodel composition techniques

An illustration of the application of metamodel merge is provided in Figure 3.21(a). Considering that classes A and P in separate metamodels have been identified to match, then they can be merged into class AP as described in the figure.

Metamodel interfacing is employed to combine two metamodels describing distinct but related domains in order to explore the relationships between them. Its implementation requires the introduction of new classes and relations (that do not necessarily belong to either of the two source metamodels) which serve as the interface between the distinct meta-models through associations. The technique is described in Figure 3.21(b) with MM1 and MM2 representing classes in separate meta-models and MM_int representing the interface class, which is introduced to establish relationships between them.

Class refinement is the technique used to establish relationships between closely related (or in fact, same concepts) expressed at different levels of refinement in two independent metamodels. Specifically, a hierarchical containment relationship is created between the two meta-models fragments (as described in Figure 3.21(c)) with the more abstract fragment as the container(s) of the more detailed descriptions provided by the other.

3.3.4 Model Transformation

A model transformation is described in [KWB03] as an automatic generation of a target model from a source model according to a transformation definition. A transformation definition is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language and a transformation rule specifies how one or more constructs in the source language can be transformed into one or more constructs in the target language.

In order to capture the special cases of model transformation where there may be multiple source and/or target models, we extend the definition of [KWB03] to describe a model transformation, in the context of this thesis, as *an automated synthesis of one or more target model(s) from one or more source model(s) according to some transformation definitions*.

The practice of model transformation in the field of computing is older than MDE. In fact, we can arguably say that it is as old as computing itself but it had been done under different nomenclatures. For instance, the compilation of program C/C++ codes to byte codes and conversion of assembly language to machine codes consist of some chains of model transformation since all these artifacts involved are also models but at low-levels of abstraction. The advent of MDE has however stimulated the development of theories and technologies to perform similar activities at some higher levels of abstraction. Moreover, model transformation in MDE gives room for user-defined transformations between models in different domain-specific languages.

Model transformation is one of the cornerstones of MDE; [SK03] considers it *the heart and soul* of MDE. It is instrumental to the widely reported benefits of MDE. For instance, the concept of separation of concerns between problem and solution domains and the consequent support for high-level description of problems rely on model transformations to, automatically, synthesize the solution artifacts. Similarly, the portability of models between multiple platforms and tools for model reuse, integration of tools and development environment and the building of bridges between technical spaces (e.g., model ware, grammar ware, document ware, data ware) all rely on some kinds of model transformations to be realized.

There are several categories of model transformation techniques and technologies in the literature; this section will discuss only a few classes along on three dimensions: relative levels of abstraction, technical spaces and languages of the source and target models(s). The reader is invited to consult [CH03, CH06, MG06, SK03, MGV+06] for extensive classifications of model transformation approaches and tools based on diverse criteria for guidance on when and how to use each variant.

Based on the relative levels of abstraction of its source and target models, a model transformation can be classified as *horizontal* or *vertical* [MG06]. A horizontal model transformation is one that involves models at the same level of abstraction. For instance, a transformation within MDA's PIM (or PSM) layer is an horizontal transformation. There may be cases where models written in different languages are required (by different tools) for some specific analysis at the same level of abstraction, it may be sufficient in such cases to create one of the models and use it as the source from which to synthesize the others. A horizontal transformation may also occur between models written in the same language usually for the purpose of model or code refactoring [ZLG05, Men06, FCS+03]. Conversely, a vertical model transformation is one, which its input and output models are in different levels of abstraction. Typical examples are PIM-PSM transformation in MDA, code synthesis and refinement.

Considering the technical spaces of the source and target models of a transformation, we can have *model-to-model transformation* (M2M), *mode-to-text transformation* (M2T), and *reverse engineering* [CH03, MG06].An M2M, as the name implies, is one, which both source and target models belong to the model ware technical space. An M2M may be *horizontal* or *vertical*. The *transformation definition* of an M2M needs to have knowledge of the meta-models of both source and target languages; the *transformation rules* are based on either *type or pattern matching* to generate target models from source models. MOF-based transformation tools (e.g., ATL [JAB+06]) map instances of meta-classes (i.e., classes, attributes and references) in the source meta-model(s) to those in the target meta-model(s), graph transformation-based tools (e.g. GReAT [BNB+07], AToM^3 [LV02]) recognize instances of some specified structural patterns in the typed graph representing the source metamodel and write the corresponding target patterns in the target models. An M2T is a transformation from the model ware technical space to grammar ware (e.g., program codes) or document technical spaces (e.g., XML documents). Unlike M2T that requires source and target meta-models, most M2T tools (e.g., XPand [Kla08].

Acceleo [MJL+06]) require only the source meta-model(s); the target documents are generated based on specified templates that define the structure of the documents [Cle01]. Thus, instances of types defined in the source meta-model are mapped to corresponding textual artifacts in the target templates. M2T tools are used extensively for code synthesis in MDE. *Reverse engineering* is the reverse of M2T; it is the extraction of high-level models from low-level textual artifacts like program codes.

Lastly, model transformations can be classified, based on whether the languages of the source and target models are the same or not, into *endogenous* and *exogenous* transformations [MG06]. The former involves source and target models written in the same language while the latter translates between source and target models written in different languages. M2T (i.e., code synthesis) and reverse engineering are typical examples of exogenous transformation; an M2M may be endogenous or exogenous. Model optimization and model refinement are examples of endogenous transformation.

3.4 Megamodeling

This section provides some backgrounds on the MDE concept of megamodeling towards the presentation of SimStudio II megamodel (architecture) in the next Chapter.

3.4.1 Definition

Megamodeling is the activity of abstract positioning of models of various kinds with respect to one another to model the linguistic architecture of software systems in terms of the involved languages, technologies, concepts and artifacts [LZ13]. It is the simplification of an MDE process with the goal of providing an overall view of the concepts [Fav04]. In [Fav05b], Favre described megamodeling as modeling in the large and the basic modeling as modeling-in-the-small; while the latter is *the activity that considers the details of models, metamodels, etc*; the former *considers the global relationships between these artifacts, without considering their content*. He describes a *megamodel* as a model that represents the complex structure of models, metamodels and other artifacts that make up an MDE process such as interpreters, transformation models, transformation engines, etc. without going into the details of the different artifacts. It should allow for reasoning about a complex software engineering process without going into the details of the technological spaces involved [FN05].

Bézivin and colleagues [BJV04, BJR+05] gave a similar account of megamodeling in independent works. They described megamodeling (or modeling-in-the-large) as the activity of establishing and using global relationships and metadata on the basic macroscopic entities of an MDE process such as models and metamodels while ignoring their internal details. Hence, a megamodel is considered to be a model of which the elements represent or refer to models, metamodels, metametamodels, services, tools, etc. and the relationships between them. A survey of several overlapping definitions of megamodeling, including the ones cited here, is provided in [HSG12].

3.4.2 Applications/uses of megamodels

This sub-section highlights some of the uses of megamodels by different practitioners extracted from a compilation of uses provided by Hebig and colleagues [HSG12].

- To define software architectures with design decisions in relations between heterogeneous models [PBR09].
- To model an MDE process and reason about the relations that can exist in the context of MDE through exemplary patterns [FN05, Fav05b]
- To model software evolution through model transformations [FN05]
- To present a global view of models and facilitate traceability between models and their elements [BFB07]
- To capture and analyze modeler's intention about how different views of a system are related to one another [SME09]

3.4.3 Formal description of megamodeling concepts

The relations between the elements of a megamodel have been defined by Favre [Fav04] from a mathematical standpoint. He opined that megamodeling is a rudimentary theory for reasoning about MDE processes and hence must be precisely defined to clarify the valid relationships between models, languages, metamodels, transformations, and systems under study. Favre argued that the use of English and informal diagrams to describe an MDE process will most likely fail to support reasoning about the process when its complexity grows beyond intuition. He proposed that a better approach is to describe basic megamodeling concepts with simple regular structures based on set theory and language theory such that the structures can be incrementally combined to describe a complex MDE architecture.

Based on his argument, Favre described a megamodel as a labeled directed graph with systems under study, models, languages, metamodels and transformations as nodes and edges with labels $\mu, \in, \chi$ and δcorresponding to relations *representationOf*, *elementOf*, *conformsTo*, and *decomposedIn* respectively. In [FN05], Favre and NGuyen extended the graph description to include additional edges and patterns that describe model transformations and evolutions.

Figure 3.22Megamodel elementOf relation

3.4.3.1 ElementOf relation

From formal language theory [Har78, HU79], a language is an infinite set of models that can be described in the language. For instance, the UML is the set of all possible UML diagrams; C++ is the set of all C++ programs. Hence a model, *m*, is an element of the language, *L*, in which it is created. i.e., $m \in L$. This relation is modeled graphically as the directed edge between nodes *m* and *L* in Figure 3.22above.

3.4.3.2 RepresentationOf relation

One of the prominent properties of a model, *m*, as provided in the definitions given previously is that it is a "representation of" a system under study, *sus*. This relation is modeled in a graph, as illustrated on the left of Figure 3.23, as a directed edge from node *m* to node *sus*.

Figure 3.23Megamodel representationOf relation

Recall that a language, *L*, is an infinite set of models; a model, *mm*, of *L* is required in order to formally reason about the latter [Fav04]. In MDE, *mm* is referred to as the "metamodel" of *L*. for instance, the language of even numbers, $L = \{2, 4, 6, 8, \dots\}$ can be represented (modelled) as $L = \{n \in \mathbb{N} | n \, mod \, 2 = 0\}$ in order to formally reason about its elements. Since *mm* is a "model" of *L*, it follows that the *representationOf* relation also holds between the duo as illustrated in the graph on the right of Figure 3.23.

3.4.3.3 ConformsTo relation

The *conformsTo* relation can be derived formally in two ways. Firstly, it can be derived mathematically from language and set theory based on model-language-metamodel relations. Recall the example of the language of even numbers presented previously under the *representationOf* relation; every valid element of the language, *L*, must satisfy (i.e., conform to) the constraints specified in the model (i.e., metamodel) of the language. In other words, given a model *m* written in a language *L* whose metamodel is *mm*, *m* "conformsTo" *mm*.

Another approach, proposed by Favre [Fav04], to derive the conformance relation is through graph transformation. To begin with, a superposition of the three graphs in Figure 3.22 and Figure 3.23 will yield the graph on the left of Figure 3.24; from this left graph, we can deduce a pattern graph comprising nodes *m*, *L*, and *mm* that forms the left hand side (LHS) of the graph transformation rule at the middle of Figure 3.24.

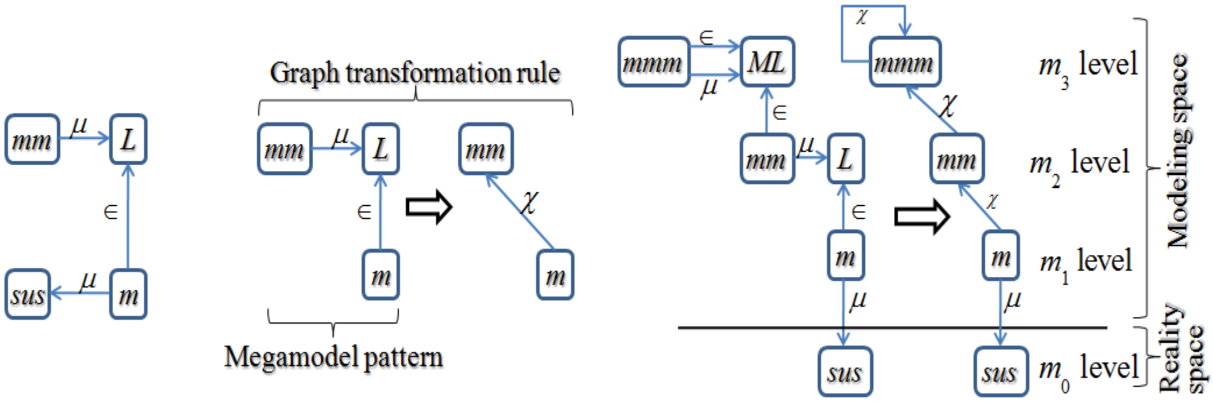


Figure 3.24Megamodel conformsTo relation

The transformation rule translates the source pattern to the graph on the right hand side (RHS), which comprises nodes *m* and *mm* linked by the *conformsTo* edge. We apply this rule on the right of Figure 3.24 to obtain the MDE metaization levels presented previously in Section 0. The LHS of the rule consists of model *m* written in language *L* and representing the system under study *sus*. The metamodel *mm* of *L* is written in metalanguage *ML* whose metamodel (i.e., metametamodel wrt. *m*) is *mmm*. Applying the rule to the LHS produces the cascaded *conformsTo* relations on the RHS.

3.4.3.4 DecomposedIn relation

This describes the relation between a composite model or MDE artifact and its components. It simply implies that the whole decomposes into the various components hierarchically.

Mathematically, this relation can be described as a tree structure having a composite model as the root, composite components as inner nodes and the smallest units as the leaves of the tree. We illustrate this relation with an example in Figure 3.25. Imagine we have a *car* model that has five component models: *engine*, *ignition system*, *suspension and steering system*, *transmission system* and *braking system* coupled to make a whole; each component can be linked to the composite model with the *decomposedIn* (δ) relation in a megamodel as shown in the diagram.

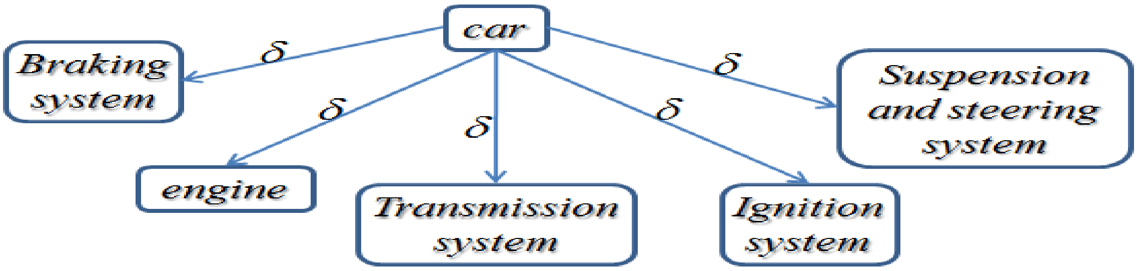


Figure 3.25Megamodel decomposedIn relation

3.4.3.5 Transformation relations

The graph structure Figure 3.26 illustrates a megamodel pattern for describing the participants in a model transformation process and their relationships to one another. The graph pattern is built from node and edge types already discussed: *model*, *language*, and *metamodel* nodes and $\mu, \in, \chi$ edges with an introduction of two seemingly new node types and four new edge (relation) types. The newly introduced edges (relations) are *dd*, *rr*, *ss*, and *tt* corresponding to *domain*, *range*, *source* and *target* respectively. Nodes *transformation instance*, $T_{instance}$, and *transformation specification*, T_{spec} are apparently new in this discussion; we will see, however, in further discussion that they are some forms of *system under study* and *model* respectively.

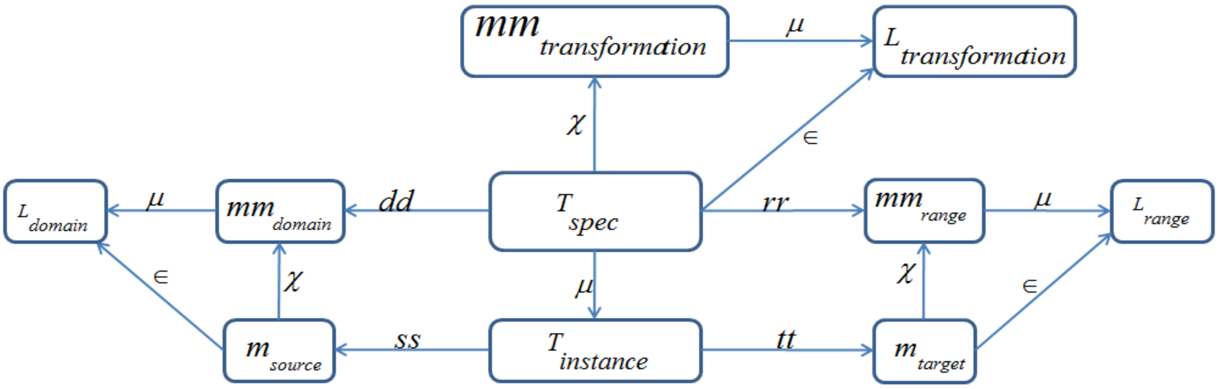


Figure 3.26Megamodel pattern for model transformation

Starting from the bottom of the graph, node m_{source} is the source (or input) model that is transformed to thetarget (or output) model m_{target} by the operation $T_{instance}$. Hence the relations of $T_{instance}$ with m_{source} and m_{target} are *source* (*ss*) and resp. *target* (*tt*). Technically, $T_{instance}$ can be considered as an in-memory system that takes in a model as input and produces another model as output; hence this system is modeled (represented by) the T_{spec}. Mathematically, T_{spec} is a function that maps two sets (in this case languages) described by the *domain metamodel*, mm_{domain}, and *range metamodel,* mm_{range}. Hence the relations of T_{spec} with mm_{domain} and mm_{range} are *dd* (doamain) and *rr* (range) respectively. Therefore m_{source} and m_{target} conform tomm_{domain}, and mm_{range}, respectively. Also being a model,T_{spec} is written is a transformation language $L_{tranform\ ation}$ and conforms to a metamodel $mm_{tranformation}$. This graph structure can be replicated within megamodels to describe systematic transformations and evolutions of models in MDE-based architectures.

3.5 ELEMENTS OF A LANGUAGE SPECIFICATION

In this section, we introduce the main elements in the specification of a computational modeling language and their purposes.

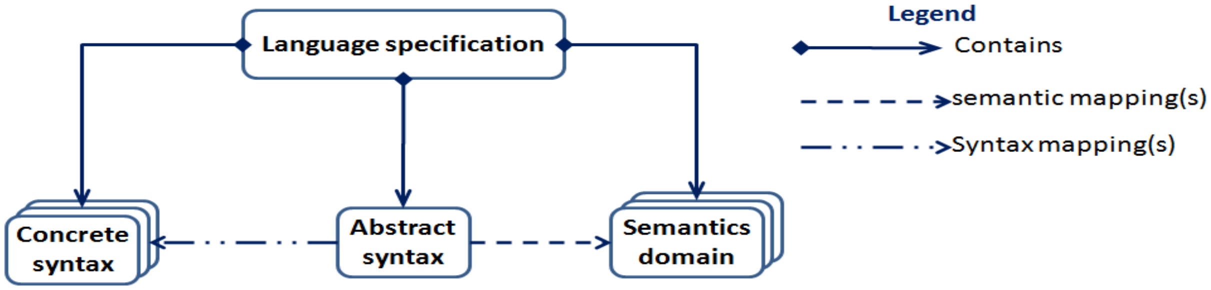


Figure 3.27 Elements of a language specification

Essentially, as Figure 3.27 depicts, a modeling language specification can consist of the abstract syntax, a family of concrete syntaxes and syntax mapping(s) and a family of semantics domains and semantic mapping(s) [Kle08, HR00].

Mathematically, the specification of a language $\mathcal{L}$can be described as:

$$\mathcal{L} = \langle A, \{C_i\}, \{S_j\}, \{m_{AC_i}\}\{m_{AS_i}\}\rangle_{i,j\in\mathbb{N}}$$

Such that $\forall C_i, m_{AC_i}: A \longrightarrow C_i$ and$\forall S_j, m_{AS_j}: A \longrightarrow S_j$.

A is the abstract syntax of $\mathcal{L}$.

$\{C_i\}$ is a family of concrete syntax specifications for $\mathcal{L}$. i.e., every C_i is aconcrete syntax specification.

$\{S_j\}$ is a family of semantics domains for $\mathcal{L}$. i.e., every S_j is a semantics domain for the language.

$\forall C_i$, $m_{AC_i}: A \longrightarrow C_i$ is a syntactic mapping which assigns concrete notations for expressing the elements of A.

$\forall S_j$, $m_{AS_j}: A \longrightarrow S_j$ is a semantics mapping which assigns meanings to the elements of A in the domain defined by S_j.

Though most language specifications define one each of concrete syntax and semantics domain, it is possible to have multiple of each. A language may have multiple concrete syntaxes, each used by different users or in different contexts or viewpoints. Similarly, a language's semantics may be defined in multiple domains for different computational purposes. In particular, we will demonstrate the use, and need for multiple semantics domains in subsequent chapters of this document.

The rest of this section elaborates a bit on the main components of a language specification; comprehensive discussions on these components have been presented in the literature (e.g., [Kle08, HR00, Sel09, HR04]).

3.5.1 Abstract Syntax

The abstract syntax of a language defines its vocabulary by describing the various concepts expressed in the language and the relationships between them. "*Abstract syntax defines the set of language concepts and the composition rules that represent the 'algebra' for combining these concepts into valid or so-called 'well-formed' models*"[Sel09].

Though abstract syntax describes the valid expressions of a language, it does not provide any information about how the entities and relationships are rendered physically; rather it serves as the bridge between those concrete representations and their semantics [Kle08, Sel09].

An Abstract syntax is often described by a metamodel in MDE-based projects. We have presented metamodeling previously in Section3.3.3. A typical technological support for defining abstract syntax for MDE-based languages is provided by the Ecore, which is the central metamodeling language in the Eclipse Modeling Framework (EMF) [SMB+08].

3.5.2 Concrete Syntax and Syntax Mapping

The concrete syntax describes the set of notations used to render the entities and relationships described in the abstract syntax. It offers the actual human-readable notation used to present and view models [Sel09]. Depending on the language type, concrete syntax notations may consist of texts, graphics or a mixture of both. i.e., it may be contain elements like words, sentences, boxes, diagrams, icons, etc.

The syntax mapping is a relation that assigns to every element of the abstract syntax, corresponding elements of the concrete syntax so that the latter can be used to express the concept described by the former.

The Graphical Modeling Framework (GMF), which ships with the Eclipse Modeling Project (EMP) [Gron09] provides a typical technological platform to define a concrete syntax (particularly graphical notations), specify the syntax mapping between it and an EMF-based abstract syntax and automate the generation of the runtime codes to power the use of the language editor. Similarly, Xtext [BCE+08] provides the technological support for the specification and implementation of *textual* concrete syntaxes based on Extended Backus Naur Form (EBNF) grammar [Wir96].

3.5.3 Semantics, Semantics Domain and Semantics Mapping

The semantics of a language is the precise and detailed meanings of its concrete modeling constructs. i.e., it defines the meanings of the concepts captured in the abstract syntax. It must provide the meaning of each expression of the language in some well-defined and well-understood domain [HR04]. In other words, the semantics domain is the context from which meanings are assigned to the concepts of a language.

Technically speaking, the semantics domain provides a conceptual model of how the computations that are being modeled in a language occur at runtime; such models of computation may be some kinds of algorithmic models, event-driven models, flow-based models, logic programming models, etc. [Sel09]. It must be noted, however, that though an explicit definition or selection of a semantics domain is vital to give meanings to the concrete expressions/notations of a language, the semantics domain itself is normally independent of the notation [HR04].

Therefore, a sound language specification must provide a relation that unambiguously associates the syntactic elements to their corresponding meanings in the semantics domain. This relation is called the semantics mapping. That is, the semantics mapping formalizes the relationships between the concepts of the language being designed and those of the semantics domain.

3.5.4 Semantics Description Methods

Formal semantics, in computer science, is primarily concerned with the rigorous specification of the meanings of grammatically correct programs, behaviors of some hardware, etc. to provide the basis for implementation or analysis and verification [NN92]. This subsection presents overviews of the main methods for formalizing the explanations of semantics of programs in different contexts. These methods are generally classified into three categories [NN92, SK95]: 1) *Operational semantics*, which describes how the effects of the computations in a program are produced when it is executed on a machine. 2) *Denotational semantics*, which uses mathematical objects to describe the effects of executing the constructs in a program, without the details of how such effects are produced. 3) A*xiomatic semantics*, which describes specific (not necessarily covering all aspects) properties - expressed in logical propositions - of the effects of the execution of the constructs in the program. We will present a brief overview of *Translational semantics*, a special method used in practice to derive rigorous semantics for high level modeling languages by connecting them with these low-level semantics description methods. It is important to note that we only provide informal descriptions of the different methods, an interested reader is invited to consult some of the numerous texts in the literature, such as [Hoa69, NN92, SK95, Sch96, FPB+09, RŞ12, Sto77, Plo04, Weg72, Hen90, Gor12], which have been dedicated to formal discussions of different aspects of the subject.

3.5.4.1 Operational semantics

Operational semantics describes the meanings of program constructs in terms of how the effects of the computations associated to them are produced when the program is executed on an abstract machine; i.e., "*how the program execution is done on an abstract machine*" [NN92, SK95]. According to Roşu and Ştefănescu, the operational semantics of a "programming" language defines a formal executable model for the language usually in terms of transition relations between program configurations, which provides a formal basis for the language's understanding and the design and implementation of software with it [RŞ12].

Applying these two descriptions in [NN92, SK95, RŞ12] to the domain of systems engineering, the operational semantics of a model specification language for DES may be described as a formal specification of the procedure to be followed by an abstract machine to generate the behavior traces of a system specification in terms of state transitions during execution. It is usually defined for an abstract machine in order to be generic enough to allow different implementers of the language in their choice platforms.

A typical example of operational semantics definition that is referred to in this thesis is the simulation protocols of DEVS atomic and coupled model specifications [ZPK00], which defines the language's operational semantics in the form of abstract algorithms that must be executed to generate the behaviors of system models described in DEVS.

3.5.4.2 Denotational semantics

Denotational semantics, also known as mathematical semantics [SK95], describes the meanings of syntactical constructs in a program in terms of mathematical objects such as numbers, truth values, functions, tupples, etc which represent the *effects* of executing such constructs without paying attention to *how* the effects are produced [NN92, SK95].

In contrast to operational semantics which describes how to generate the effects of executing a program, denotational semantics uses mathematical objects to precisely *express the effects* themselves while abstracting away from the how? For instance, two program functions implementing different algorithms to compute the factorial of a given number can have different operational semantics but the same denotational semantics since they both produce the same effects - the factorial of the input value. Conversely, if we execute one of the two functions with different inputs, they can have the same operational semantics but different denotational semantics; it is so because the "*how to generate the effects*" remain unchanged with different inputs but the "*effects generated"* are different.

According to Slonneger and Kurtz [SK95], the fundamental principle underlying denotational semantics is the notion that computer programs and the objects they manipulate are symbolic realizations of abstract mathematical objects; hence, each statement of the language is said to *denote* a mathematical object. Thus, we can associate an appropriate mathematical object with each statement of the language so that the former is a *denotation* of the latter. The denotations of complex program statements are built from the composition of the denotations of its sub-statements and the denotation of a complete program derived from the composition of the denotations of its individual statements.

3.5.4.3 Axiomatic semantics

While operational (resp. denotational) semantics describes how to realize the computations of a program (resp. the effects of executing the program), axiomatic semantics expresses specific properties of the effects of executing certain constructs of the program in terms of assertions for logical reasoning [NN92]. In essence, it allows one to prove whether, or not the desired effects will be produced when the program is executed.

Based on methods of logical deductions from predicate logics, axiomatic semantics describes the meaning of a computer program using *assertions* about relationships that must remain the same (i.e., invariants) for all executions of the program. An assertion, in this context, is a logical formula (statement), on system variables and constants, constructed using predicate calculus; an assertion becomes *true* or *false* when the variables involved take specific values during program execution.

The primary goal of axiomatic semantics is to provide axioms and proof rules that capture the intended meaning of each command in a programming language to define a proof system, typically based on Hoare's triples [Hoa69]as a basis for logical reasoning with, and formal verification of programs [SK95, RŞ12].

Hoare [Hoa69] has proposed a prominent proof system to provide axiomatic semantics in the format "$Pre\ \{S\}\ Post$" or "$true\ \{S\}\ Post$" where$\{S\}$ is a sequence of program statements and Pre (resp. $Post$) is a sequence of assertions specifying the pre-conditions (resp. post conditions) for the execution of $\{S\}$. While Predescribes the relationships between the system variables based on their values just before the execution of $\{S\}$is initiated, $Post$ is a description of the relationships between the variables based on the result of the execution of $\{S\}$. According to Hoare, $Pre\ \{S\}\ Post$ may be interpreted as "If the assertion Pre is true before initiation of a program $\{S\}$, then the assertion $Post$ will be true on its completion (if $\{S\}$ runs to completion)." The alternative format"$true\ \{S\}\ Post$" is applicable where no preconditions are imposed before the initiation of the execution of $\{S\}$.A proof system of this form can be used to prove the partial correctness (or otherwise) of$\{S\}$relative to the specification,the correctness is said to be complete if we can also prove that $\{S\}$ will eventually run to completion.

3.5.4.4 Translational semantics

Translational semantics, also known as *semantic anchoring* [CSA+05], is a method commonly used (by courtesy of model transformation) to formalize the semantics of DSLs. It involves the mapping of a DSL's abstract syntax onto the abstract syntax, of an existing formal language, L, with a formalized and well-understood semantics so that the semantics of *DSL* can be inferred from that of L [SK95, CSA+05, BGM+11]. Essentially, the idea of translational semantics was borne out of the quest for a way to complement high-level languages with formally defined semantics [BGM+11].

According to Slonneger and Kurtz in [SK95], translational semantics is based on the premise that the semantics of a language can be preserved when it is translated into another form, called the target language. Therefore, if the target language can be defined by a small number of primitive constructs that are closely related to actual or hypothetical machine architecture, then it can provide a basis to define the semantics of the source language.

In their paper on semantic anchoring [CSA+05], Chen et al. proposed a two-step translational semantics strategy to define the behavioral semantics of DSLs as follows:

i. *Define a set of minimal modeling languages $\{L_i\}$ for the basic behavioral abstractions and develop the precise specifications for all components of $L_i = \langle A_i, C_i, S_i, M_{Si}, M_{Ci}\rangle$. We use the term "semantic unit" to describe these basic modeling languages.*

ii. *Define the behavioral semantics of an arbitrary $L = \langle A, C, S, M_S, M_C\rangle$modeling language transformationally by specifying the $M_A : A \rightarrow A_i$ mapping. The $M_S : A \rightarrow S$ semantic mapping of L is defined by the $M_S = M_{Si} \circ M_A$ composition, which indicates that the semantics of L is anchored to the semantics domain$S_i$of L_i.*

This strategy succinctly describes the approach commonly used when a high-level modeling interface is provided to alleviate the complexity of dealing with a highly mathematical formalism such as described previously in Chapter 2 of this document (see Section 2.2). In fact, one can as well claim that it is applicable to describe the basis for the pair wise integration of MDSE methodologies discussed in Section 2.3. For instance, when a DEVS-based language is transformed to Z for formal analysis, we can say that, in this context, the semantics of the former is anchored to the semantics domain of the latter for logical analysis. In [SK95], Slonneger and Kurtz argued that even compilers of general purpose languages *perform a translation of high level language into a low level such that executing this target program on a computer provides the semantics of the program in a high level language*.

While translational semantics offers the great advantage of allowing a DSL to take benefit of the existing tools of the language into which it is translated, it also has a few challenges to meet. Notable among the challenges, as pointed out by the authors of [BGM+11] is that "*since the semantics definition is not defined in the metamodel of the DSML, it is very challenging to correctly map the constructs of the DSML into the constructs of the target language. The underlying cause for this is that the mappings are not at the same level of abstraction and the target language may not have a simple mapping from the constructs in the source language."* This corroborates our argument in Section 2.3 that a *surjective* mapping of the source language to the target language is not always guaranteed. Another important challenge of translational semantics is that an expert of the source language may find it difficult to comprehend the traces generated by execution in the target language; this can be handled, however, by either translating the traces to the domain of the source language or visualizing/animating it with some specially designed tools.

3.6 CONCLUSION

In this chapter, we introduced the theories and techniques upon which the contributions of this thesis are based. In Section 3.2, we introduced DEVS, a system-theoretic formalism for simulation of DES, Z notations and its object-oriented variant, Object-Z, both of which are used for the specification of state-based systems for formal analysis, and TL for specifying temporal properties to be verified about the behavior of DESs. We also presented an overview of a pattern-based approach, proposed by Dwyer et al. in the late 1990s to alleviate the complexity of property specification with TL. We used the beverage vending system as a running example to illustrate the use of each of the formalisms to model different aspects of a system. This running example, though considerably simple, typifies the herculean task of specifying a system with disparate formalisms for different analysis methodologies towards the complementary study and analysis of different aspects of the system. We have presented, in the previous chapter, a literature review of research efforts to alleviate this task; the contributions of this thesis to the same course are presented in subsequent chapters.

We presented, in Section 3.3, an overview of MDE techniques and terminologies. Essentially, we discussed MDE's global objectives; MDA, which is a proposal by the OMG™ for industrial implementation of the MDE initiatives; and the fundamental tasks in an MDE process: modeling, metamodeling and model transformations between and (or within) technological spaces and languages. We also presented an overview of the Ecore metamodeling technology and the techniques proposed by Emerson and Sztipanovits for the composition of metamodels. Sequel to MDE, we presented an introduction to the concept of megamodeling, in Section 3.4, for formal description of the relationships between the different concepts and artifacts in an MDE process.MDE and megamodeling techniques will be used extensively in subsequent chapters for the formulation and presentation of the thesis' contributions. We expect that this chapter has provided the necessary for the reader to follow all MDE-based presentations in this thesis.

Finally, in Section3.5, we introduced the essential elements in the specification of a modeling language. We described the abstract syntax, which involves the definition of a language's vocabulary; the concrete syntax and syntax mapping to provide the concrete notations for expressing the concepts and relations in the abstract syntax; and the semantics domains and semantics mapping to, unambiguously, define the meanings of the elements of the language's vocabulary. We also presented overviews of different theoretical methods for describing the semantics of a language for different purposes. We will use the lessons learned in this section, in combination with MDE techniques, in subsequent chapters, for the specification of a high-level system specification language, which is at the heart of the contribution of this thesis.

4 SIMSTUDIO II: AN INTEGRATIVE FRAMEWORK FOR MODEL-DRIVEN SYSTEMS ENGINEERING

4.1 INTRODUCTION

This chapter builds on our preliminary results, reported in [AT15a, AT15b, AT16], to launch the presentation of the contributions of the thesis and provide answers to research questions RQ1 and RQ4 and abstract answers to RQ2 and RQ3. We explore, with the SimStudio II framework, the integration of three MDSE theories and methodologies - simulation, formal methods and enactment - with the goal of harnessing the synergy of the diverse theories, tools and experiences for complementary, rather than competitive, studies and investigations of systems' static and dynamics properties.

In the course of our research work towards this thesis, we realized from our preliminary findings - reported in [AT15a, AT15b, AT16] - that having a unified formalism at the front end is paramount to the integration of the disparate methodologies in the proposed framework. We then defined the *High Level Language for Systems Specification (HiLLS)*to play this role in the SimStudio II framework. Essentially, HiLLS abstract syntax is built from DES and software engineering concepts and mapped onto DEVS for simulation, a DEVS-based framework for enactment, and Z and TL for formal analysis.

In the overall, the work presented in this thesis took its initial momentum from a decade-long research agenda outlined in [Tra08], named SimStudio, and which has nurtured a previous doctoral thesis. In essence, with the SimStudio II, we have widened the horizons of SimStudio's legacy to cover more MDSE methodologies and clear the way for the incremental realizations of the long-term goals of the project.

In the sequel, we begin the rest of the chapter with the overviews of the SimStudio project initiative and the previous thesis that had stemmed from it in Section4.2. This is followed by a description of the methodology and functional requirements of SimStudio II in Section 4.3. Section 04.4 presents the framework's megamodel, a formal description of its architecture showing the positions of the various artifacts relative to one another from MDE perspective. In Section 4.5, we propose a process model to describe a workflow to provide a guide to using the framework - to achieve the different goals - based on who performs what activity and when. Finally, in Section 4.6, the chapter ends with concluding remarks and an outline for elaborate presentations of the various elements of the framework.

4.2 EVOLUTION OF THE SIMSTUDIO PROJECT

4.2.1 The SimStudio Manifesto

The SimStudio framework was first envisioned by Traoré [Tra08] in his reaction to the emergent and far-reaching quests for an operational M&S framework to match theoretical advancements with computing and technological infrastructures towards advancing simulation-based engineering science [GCG+05,PIT+05, GS05, NSF06]. There were concerns that though M&S had gained considerable adoption for solving problems in most science and engineering domains, there was the need to deal with some pressing issues relating to simulation models and the performance of simulators for the practice of M&S to continue to meet the requirements of growing system complexities. Those concerns include the verification and validation of the credibility of simulation models, model reusability and portability between existing simulation environments, space-time complexities of simulation protocols, tools interoperability, … Traoré noted that the grand challenges raised the need to treat simulation models as algebraic entities which can be manipulated symbolically as well as be easily translatable into operational objects. He then grouped the various concerns to into four research axes that must be explored and concretized by an operational framework for M&S:

Axis 1. *An algebraic axis,* also known as *specification axis,* to study how to formally specify a model and the context(s) in which it is used: efforts should be made to answer the question "what are the objects of the domain and what relationship do they entertain?"

Axis 2. *A logical analysis axis* to explore the underlying logical semantics to make simulation models amenable to formal analysis and pave the way for logical reasoning about the structural and behavioral properties of models. He argued that in addition to the classic approach of post-simulation analysis of traces, there was also the need to consider ante-simulation exploration of properties. He noted that this could also help in verifying some algebraic properties such as the true applicability of the model to the context of the simulation in the first place.

Axis 3. *An executive axis* to tackle, both at algorithmic and technological levels, the automated synthesis of executable simulation codes from models and automatic generation of output trajectories. He argued that bringing transparency in the specification-to-code process in the reverse process can help to maintain a clear separation of modeling activities from simulation activities in M&S.

Axis 4. *An application axis* to deal with how to support the scale crossing of application codes; he identified the issues to explore here to be the definition of generic simulation-based problem solving schemes and the integration of software components in real-time environments.

The SimStudio project was envisioned to concretize the theoretical research efforts identified in the proposed research axes with the aim of building a next generation M&S framework that will serve both as a virtual lab to study and experiment with advanced M&S concepts and as a collaborative and community-focused platform for the mutualization of M&S resources. To set the research work in motion, an extensible web-based plug-in architecture was proposed in [Tra08], which advocates continuous installation of plug-ins (by the user community) for modeling, formal analysis, simulation and visualization of simulation traces with model transformation interfaces to integrate the heterogeneous plug-in modules.

The SimStudio manifesto is indeed an encompassing outline of a long-term research agenda; a doctoral research work, by Touraille [Tou12], has been expended in studying some aspects of the proposed framework to provide answers to some of the questions raised. We present an overview of the doctoral thesis in the next subsection.

4.2.2 A Previous Thesis on SimStudio

In his doctoral thesis, Touraille [Tou12] proposed a framework, described in Figure 4.1below, to concretize some of the research axes and the plug-in based architecture proposed in [Tra08]; he was particularly interested in applying his findings to address the problem of lack of tools for interoperability and issues on performance of simulators within the DEVS M&S paradigm.

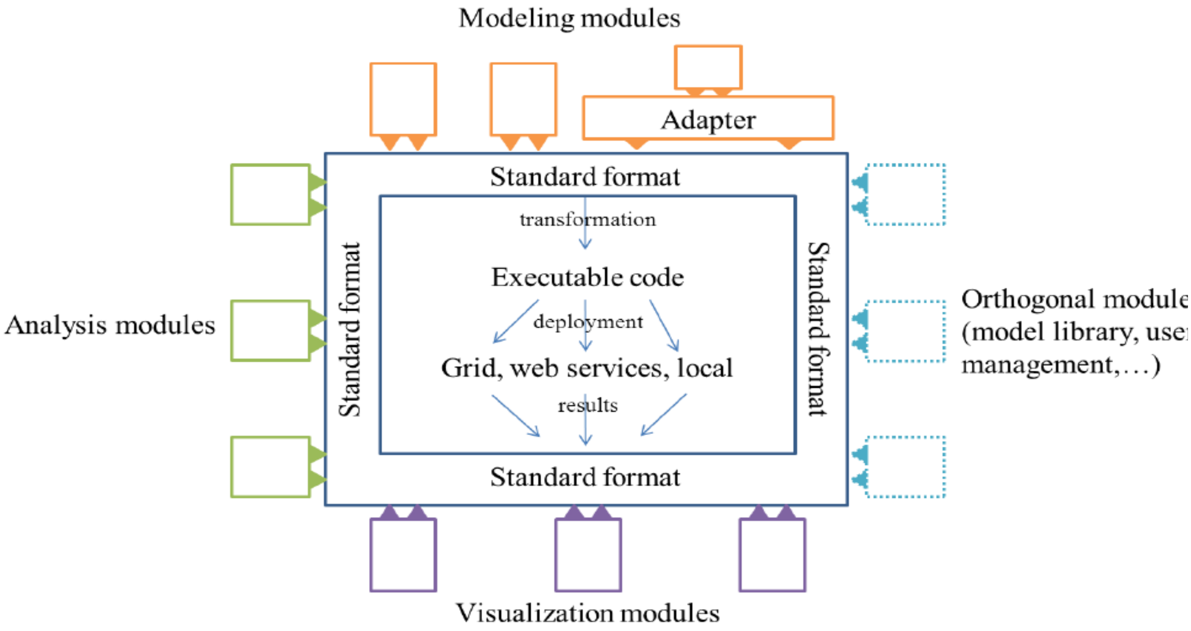


Figure 4.1 SimStudio architecture(excerpted from [Tou12, TTH11])

Touraille recognized that the problem of tool interoperability in M&S, which is a major hindrance to collaborative research among scientists in the same domain, could be attributed to the non-existence of a consensual standard for expressing DEVS models, which had led to the proliferation of models in independent formats that are tightly tied to particular environments. He then took the challenge and proposed an XML markup for DEVS to define an interchange format between disparate DEVS-based tools in his plug in-based SimStudio architecture, which is described in Figure 4.1above.

- **Standard format:** To deal with the issue of interoperability among DEVS tools, Touraille proposed the DEVS Markup Language (DML) [TTH09], an XML markup language for DEVS, at the core of the framework's architecture, to serve as a "standard format" (see Figure 4.1) for model representation. DML is considered to capture essential DEVS concepts in a platform-independent manner to be suitable to provide the required liaison to glue heterogeneous DEVS-based M&S tools together in the framework.
- **Modeling modules:** These comprise new and legacy model editors for DEVS. While new editors may be built to store models directly in the "standard format", legacy editors can be integrated into the framework through adapters that encapsulate model transformation specifications targeting DML (standard format). Model modules for non-DEVS simulation formalisms may also be accommodated in the framework through suitable adapters; this claim is premised on the work of Vangheluwe [Van00], which suggests that DEVS is expressible enough to serve as a common denominator for DES simulation formalisms.
- **Simulation modules**: These are platform-specific software implementations of the DEVS simulation protocols and algorithms. The executable simulation codes (see the inner rectangle of Figure 4.1) are obtained through model transformations with the DML standard format as source and selected simulation modules as target. Depending on the capability of the module, a simulation process may be deployed on a local system or transparently over distributed infrastructures.
- **Visualization modules:** These are modules for presenting simulation traces in various forms: listings, diagrams, animations, etc.
- **Analysis modules**: These include tools to check the structural validity (i.e., wellformedness property) of models with respect to the language's syntax and post-simulation analysis of simulation results, for instance by computing statistics. These, however, do not include the capacity for rigorous ante-simulation logical analysis with formal methods as campaigned in [Tra08].
- **Managing modules**: These modules offer orthogonal services such as model repositories, collaborative tools, workspace customization, etc.

Another global contribution of Touraille's thesis to the concretization of the SimStudio manifesto is in the area of enhancing the performance of simulation processes. He proposed the DEVS-Meta Simulator (DEVS-MS) [TTH10], a parameterized DEVS simulator that uses the C++ Template MetaProgramming technique [AG04] to generate executable codes of specialized simulators for input DEVS models at compilation time. The DEVS-MS was meant to enhance the performance of a DEVS simulation process by generating only artifacts that are necessary for simulating the input model while filtering out time-consuming processes that are contained in the generic simulators, but which are not necessarily required for the input model in particular. We did not explore the area of simulator performance in this thesis, hence no further details will be provided on the topic. An interested reader may want to consult [Tou12, TTH10] for detailed discussions on DEV-MS and its comparison with generic simulators.

We recall from Section 4.2.1 that the SimStudio manifesto, as envisioned in [Tra08], prescribed research axes for both simulation and logical analysis using FM. The concretization of the manifesto in [Tou12] is however focused on simulation-based approach to MDSE. Thus, we can classify Touraille's integration approach into the category of methodology-specific integration approaches discussed in Chapter 2 (Section2.2).

The general contribution of this thesis to the project, under the name SimStudio II, widens the horizons of the current solutions by studying both the simulation and FM axes as well as an additional axis for enactment. In other words, we explore the interoperability between, not only simulation tools, but also between development tools for simulation, formal analysis and enactment as well as collaborations among their respective practitioners. i.e., we explore intra- and inter-disciplinary collaborations of the three dimensions of MDSE. This has become important with the increasing need for the diverse methodologies to be used in synergy, rather than competitively, for the development of complex systems. The next section presents the overview of our methodology and the functional requirements of SimStudio II before we elaborate on its various components in subsequent sections and chapters.

4.3 THE SIMSTUDIO II APPROACH

4.3.1 MDSE Methodology Integration Approach in SimStudio II

Recall that we discussed the state of the practice of approaches to methodology integration in Chapter 2 and classified them into two broad categories: methodology-specific and pair wise integration approaches. The former approach involves finding a way to express the concepts of a domain in a format that is generic enough not to be tied to any of the existing tools, and yet portable enough to be compatible with as many tools as possible. The latter approach relies on the correspondences between the underlying formalisms of two disparate analysis methodologies to define transformation rules between them. We also discussed the strengths and weaknesses of each category. The former approach offers the benefit of allowing the systematic synthesis of multiple and disparate sets of solution- and/or platform-specific artifacts from a platform-independent system model; it is, however, limited by the fact that the unifying formalisms are often targeted at specific MDSE methodologies, thereby making it difficult to put analysis engines for other methodologies behind them. In contrast, the latter approach allows for inter-disciplinary transformation of models between disparate MDSE methodologies. The mapping functions from the source to target formalisms are, however, usually not surjective; hence, the need to always resort to manual update of all models during the several iterations of computational analysis activities.

After taking a keen look at the methodology-specific integration approaches, we recognized that the definition of a generic representation format for any group of tools is premised on the fact that they all represent the same sets of concepts but in different ways. For instance, all DEVS simulation tools represent the same set of concepts defined in [ZPK00] but in different formats and probably at different levels of refinements. By applying the same reasoning to the integration of simulation, formal analysis and enactment methodologies for DES, we can attempt to define a generic representation of models for the three methodologies since they describe sets of concepts that are only slightly different from one another though in different forms and at different levels of abstraction. For instance, a DES description is typically characterized by concepts like input, output, components/subsystems, states, state transitions, etc. and any formalism for DES has a way of describing most (if not all) of these concepts irrespective of the analysis methodology behind it. The integration approach we developed for SimStudio II stemmed from this reasoning. While this may appear rather ambitiously motivated, we will show in Chapter 5 that such formalisms are, in fact, realizable though not trivial. In this chapter, we focus on the macro-description of the methodology a presentation of the framework's architecture.

Figure 4.2 describes the proposed approach to integrate three disparate MDSE methodologies - simulation, formal analysis and enactment - behind a unified formalism in SimStudio II. The framework is described within the dashed box. Each plane of the 3D system contains the formalisms and technologies of one of the three analysis methods. i.e., the XY, YZ and XZ planes contains the theories and techniques for simulation, formal analysis and enactment respectively.

We mentioned in Section 4.1 that the *High Level Language for Systems Specification* (HiLLS) is the unified formalism we defined for methodology integration in SimStudio II. Imagine that, in Figure 4.2, the DES concepts for modeling systems with HiLLS, denoted by the $\boldsymbol{H_1}$,$\boldsymbol{H_2}$, **…**, $\boldsymbol{H_n}$ polygons, occupy the three-dimensional (3D) space enclosed by the XY, YZ and XZ planes while the modeling concepts for each of the three methodologies reside in its two-dimensional (2D) plane. i.e., modeling concepts for simulation (resp. enactment), denoted by the $\boldsymbol{s_1}$,$\boldsymbol{s_2}$, **…**, $\boldsymbol{s_n}$ circles (resp. $\boldsymbol{enact_1}$,$\boldsymbol{enact_2}$, **…**, $\boldsymbol{enact_n}$ rectangles) in the XY (resp. XZ) plane. The formal analysis methodologies reside in the XY plane, which is divided into two parts by a horizontal dashed line such that the concepts for system specification, denoted by $\boldsymbol{ss_1}$,$\boldsymbol{ss_2}$, **…**, $\boldsymbol{ss_n}$ rectangles reside below the line while the concepts for formal specification of required properties, denoted by the $\boldsymbol{ps_1}$,$\boldsymbol{ps_2}$, **…**, $\boldsymbol{ps_n}$ rectangles reside above the dividing line.

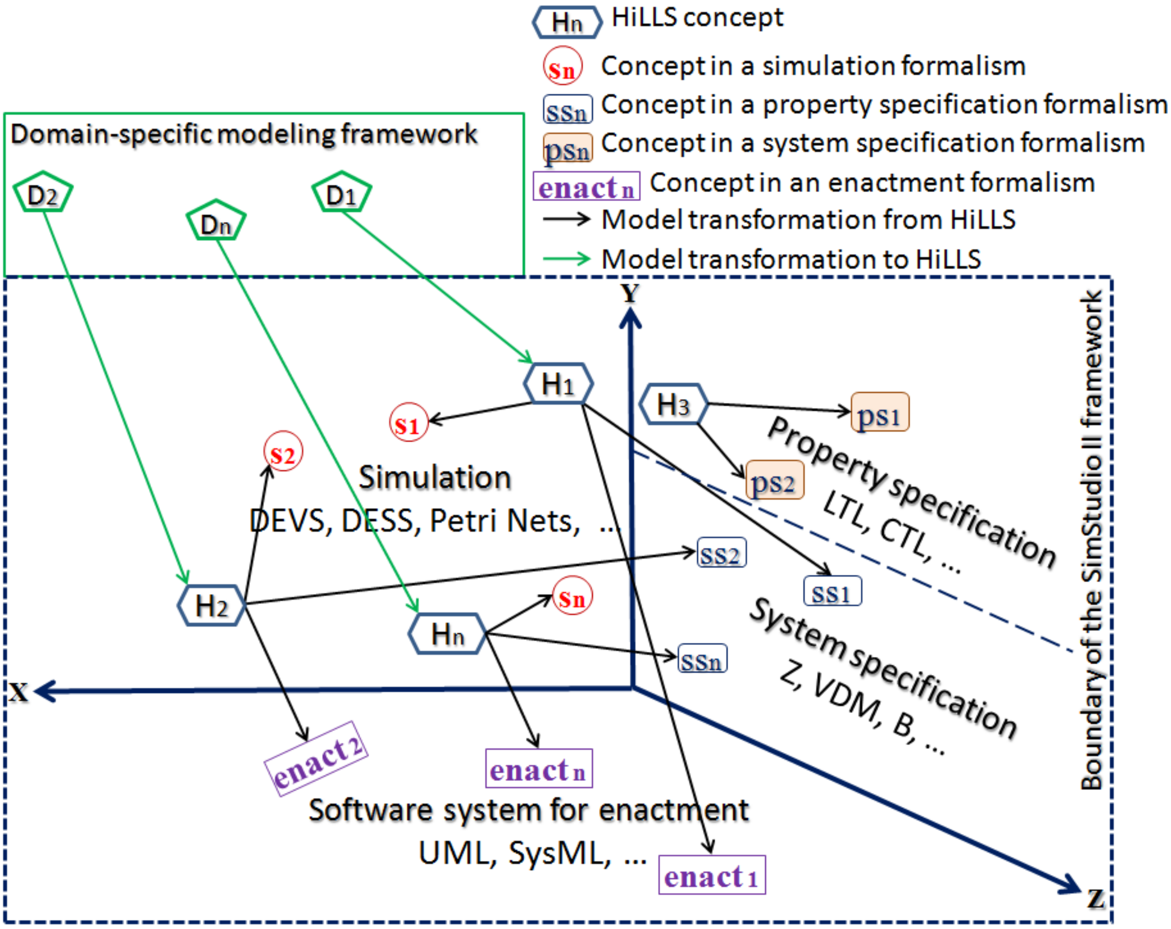

Hn HiLLS concept
Sn Concept in a simulation formalism
SSn Concept in a property specification formalism
pSn Concept in a system specification formalism
enactn Concept in an enactment formalism
Model transformation from HiLLS
Model transformation to HiLLS
Domain-specific modeling framework
D2
Dn
D1
Y
H1
H3
ps1
S1
S2
Simulation
DEVS, DESS, Petri Nets, ...
Property specification
LTL, CTL, ...
ps2
ss2
ss1
H2
Hn
Sn
SSn
System specification
Z, VDM, B, ...
Boundary of the SimStudio II framework
X
enact2
enactn
Software system for enactment
UML, SysML, ...
enact1
Z

Figure 4.2 SimStudio II methodology

By convention, if we were to use the three methodologies in isolation, the modeling activity for each of them resides within the 2D plane containing it. The approach in SimStudio II takes a departure from this conventional practice by moving the task of creating and updating models to the 3D space enclosed by the planes. Through model transformations from HiLLS to the chosen formalisms in the different planes, a given system model M_{HiLLS} in the 3D space has projections M_S on the XY plane, M_F on the YZ plane and M_E on the XZ plane for simulation, formal analysis and enactment respectively. In other words, M_{HiLLS} is the unified or shared model that establishes the links between the MDSE processes in the different planes. We will give a detailed presentation, in Chapter 5, of how the HiLLS' syntax has been built from a disciplined integration of concepts described by some considerably universal DES formalism in the three planes.

As a contribution to the state of the art, we claim that the integration approach described in Figure 4.1 has the potential to combine the strengths of both method-specific and pair-wise integration approaches; it presents a unified formalism at the front-end to describe one independent model, which permits the systematic derivations of the artifacts needed by three disparate MDSE processes.

In addition to direct modeling with universal DES concepts within the 3D space, the proposed solution also envisions the mapping of DES-compatible domain-specific models (DSMs) to models within the 3D space so that the DSM framework can take benefit of the computational analysis infrastructure provided by the SimStudio II. For instance, given a DES-compatible domain-specific language with concepts $\boldsymbol{D_1}$, $\boldsymbol{D_2}$, **…**, $\boldsymbol{D_n}$ illustrated in the upper part of Figure 4.1, the domain concept-to-SimStudio II mapping can be realized with the green transformation arrows targeting the ***H*** concepts in the diagram.

Interestingly, there is a philosophical similarity between the approach adopted in [Tou12] and that proposed in this thesis to promote interoperability between tools, and collaboration between practitioners: both approaches are based on some unified formalisms for model representation. While the former uses the DML "standard format" to capture DEVS concepts in a platform-independent format, the latter uses "HiLLS" to express DES in a methodology-independent form. The main functional difference is in the area of the target user communities; while the scope of the former is restricted to the M&S community, the latter targets a wider community with the aim of exploiting the synergy of diverse expertise and capacities to perform a task that cannot be individually performed by any of them. It should be noted, however, that the work of this thesis does not override the intra-disciplinary collaboration established by the previous thesis; on the contrary, the present thesis may be regarded as an "overlay" over the previous work and other similar efforts for formal analysis and enactment to serve as a layer of inter-disciplinary cooperation between them.

4.3.2 Functional Requirements of SimStudio II

The UML use case diagram in Figure 4.3 presents the various MDSE activities that may be performed, and by whom, with the proposed SimStudio II framework. It should allow simulation, formal analysis or enactment experts to create and edit system models with HiLLS and validate the models against the language's syntactic constraints.

HiLLS should allow a *formal analysis expert* to model system's required properties in addition to the system itself. From the HiLLS model of both the system and its requirements, it should be possible to, systematically, generate the necessary artifacts to run a formal analysis of the model for a rigorous logical investigation of the specified properties.

Similarly, a *simulation expert* should be able to model experimental frames in HiLLS, couple them to system models and systematically generate the low-level artifacts required to run simulation processes.

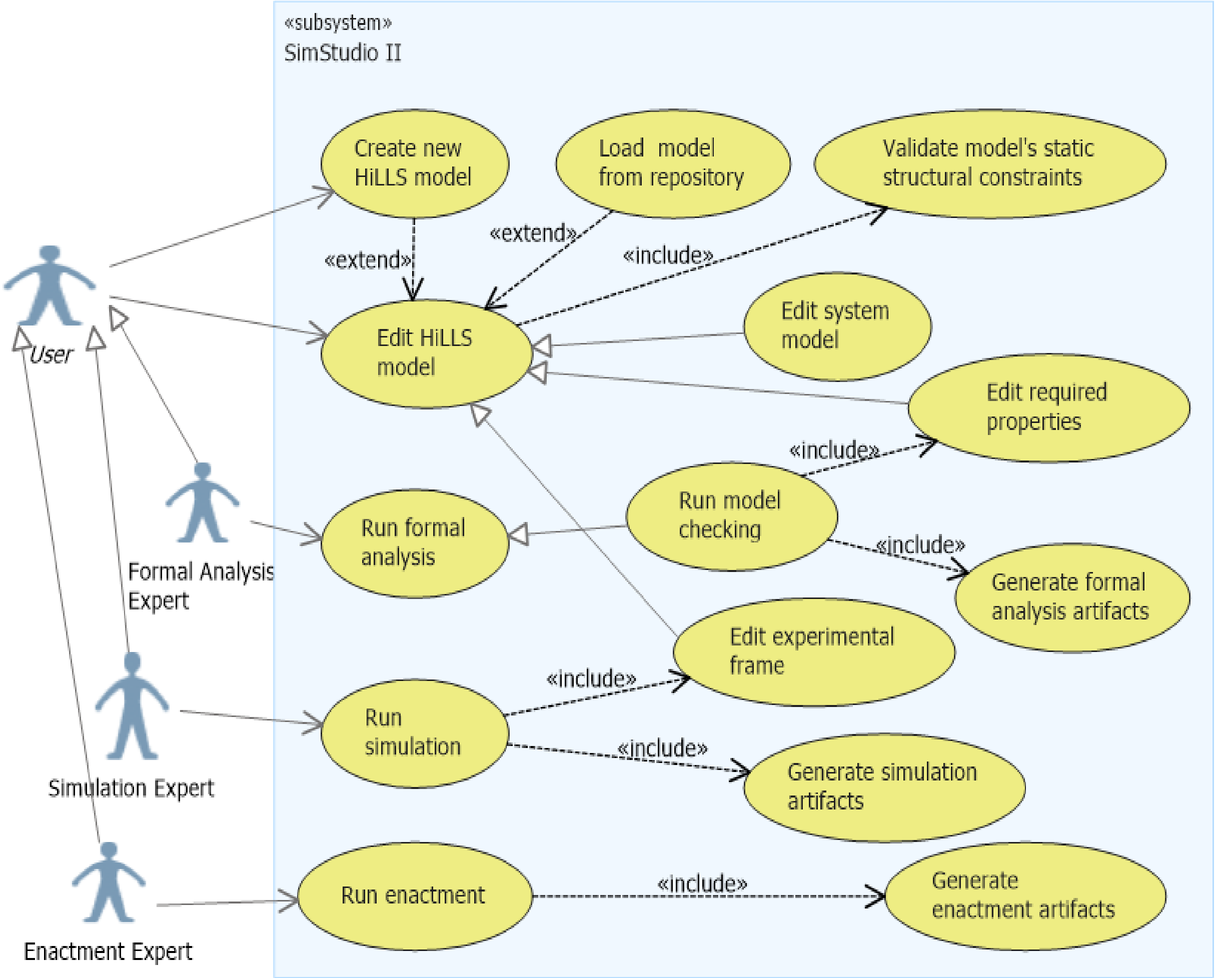


Figure 4.3Functional requirements of SimStudio II

Finally, an *enactment expert* should be able to generate, from a given HiLLS model of a system, the software prototype of the system under study.

In all cases, the same system model can be used for any of the three MDSE processes.

In the next section, we will present the architecture of SimStudio II to discuss the formal relationships between the various artifacts in the framework.

4.4 SIMSTUDIO II ARCHITECTURE

The integration methodology in SimStudio II relies heavily on MDE techniques to hide the complexities of dealing with the different formalisms by federating them behind a unified high-level language. Figure 4.4 presents the framework's architecture, showing its various elements and the intricate relationships between them. Considering their intricacies, we adopt the megamodeling relations presented earlier in Section3.4 to, unambiguously, position each of the various artifacts with respect to the others.

The architecture is a complex labeled and directed typed graph. Each node of the graph is a software engineering (SE) artifact, whose type belong to the set {*system*, *model*, *metamodel*, *metametamodel*, *language*}, and which itself belongs to one of the different technological spaces (TSs) - model ware, grammar ware and document ware. Each directed edge of the graph has a label from the set $\{\mu, \in, \chi, \delta, dd, rr, s, t\}$, which corresponds to a megamodeling relation. For clarity, the SE artifacts represented in the framework are grouped according to their respective TSs; and within each group, they are further categorized according to the metaization levels - m_0, m_1, m_2 and m_3 - presented previously. For each TS group, every m_1 artifact is a model (μ relation) of the system under study (SUS) at the m_0 level, which is outside the framework, except for transformation models that specify some in-memory instances of model transformation processes.

Conceptually, the entire architecture is a disciplined cascade of two levels of MDA-induced tree structures with HiLLS models at the topmost root and executable codes for the different MDSE processes as leaves. Let us discuss the different artifacts under their respective TSs for ease of understanding.

4.4.1 Model ware Artifacts in SimStudio II

This group (see top of Figure 4.4) comprises the SE artifacts involved in the development of generic high-level models of systems, their behavioral requirements and experimental frames. At the m_1 level, a *HiLLS model* may be a composition of (δ relation) a *system model*, a *requirement model* and an *experimental frame* of the SUS. *HiLLS model* is authored with ($\in$ relation) the language *HiLLS* and conforms to (χ relation) the *HiLLS metamodel*, at m_2, which itself conforms to the *MOF* at the m_3 level. The *HiLLS metamodel*is a model of (μ relation) *HiLLS*; it describes the language's syntax rules, which must be respected while using the language to describe models.

From MDA viewpoint (recall from Section 3.3.1), a *HiLLS model* is a "Methodology-Independent Model" (MIM) in that the *system model* it describes is not directly tied to any of the intended analysis methodologies; rather, it is expressive enough to capture the information required for the systematic synthesis of their required artifacts.

4.4.2 Document ware Artifacts in SimStudio II

This group, located on the right side of Figure 4.4, consists of the methodology-specific models of the SUS based on DEVS, Z, TL and a DEVS-based enactment framework (see Chapter 5).

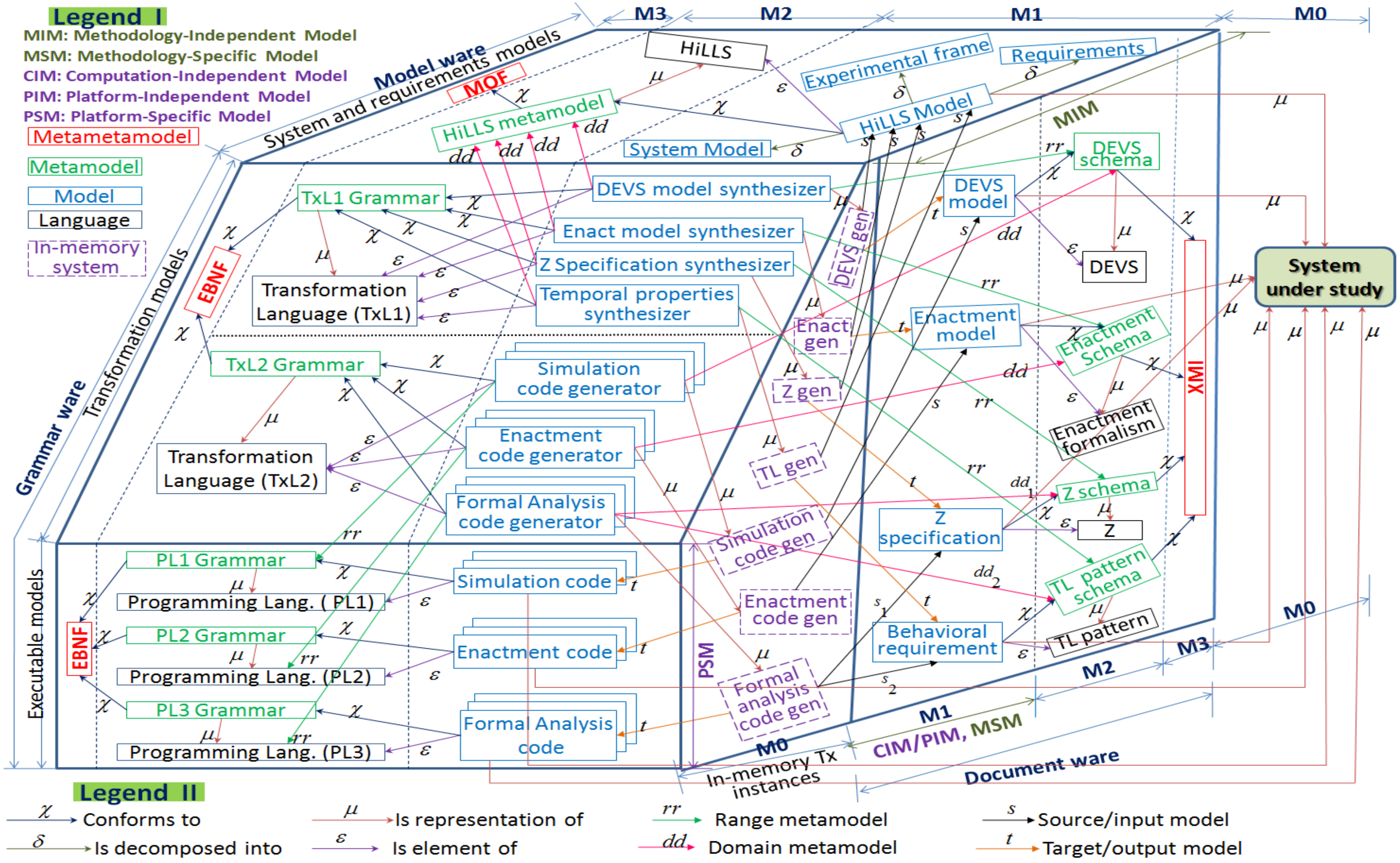


Figure 4.4Megamodel of the SimStudio II framework

In the m_0 level, we have the *DEVS model*, *Enactment model*, *Z specification*, *Behavioral requirements* which conform to the *DEVS schema*, *Enactment schema*, *Z schema* (this is different from the "Z schema" discussed previously in Section 3.2.4) and *TL pattern schema*(see Section 3.2.6.3) respectively. Each of the "schemas" is a "metamodel" of corresponding formalism expressed in, and which conforms to, XMI[14] (XML Metadata Interchange).

XMI is an interchange format for metadata that is defined in terms of the Meta Object Facility (MOF) standard [OMG15]. It is a widely used XML interchange format that defines an XML-based representation of objects in terms of elements and attributes, a standard mechanism to link objects within the same file or across files, a validation scheme for XMI documents using XML schemas, and Object identity, which allows objects to be referenced from other objects in terms of IDs and UUIDs. According to the OMG™, every instance of the MOF is required to have a corresponding XMI document to perform some XML validation on the data serialized in the XML document. This requirement is implemented in the Eclipse-based EMF [SMB+08] and its associated projects in the Eclipse Modeling Projects (EMP) [Gro09] where an every model or metamodel specified has a corresponding XMI document generated for it. In fact, EMF-based models are permanently stored in XMI.

This group serves as the middleware between the model ware elements and the *executable models* in the grammar ware group. From MDA viewpoint, the models (m_1 elements) in this group can be considered to be Methodology-Specific Models (MSM) when examined relative to the models in the model ware group; i.e., they are systematically derived for specific MDSE methodologies from the HiLLS model, which is considerably generic rather than being tied to any of the methodologies. In contrast, when examined, from MDA viewpoint also, relative to the executable models in the grammar ware group, each of*DEVS model*, *Enactment model* and *Z specification* plays the role of a Platform-Independent Model (PIM) while *Behavioral requirements* plays the role of a Computational-Independent Model (CIM) since it does not represent any computation activity. This is why we said, in the beginning of this section, that the architecture of the proposed framework is a cascade of two MDA processes. *DEVS model*, *Enactment model* and *Z specification* are PIMs because system data they contain are represented in XMI format, which is not directly tied to any programming implementation platform.

4.4.3 Grammar ware Artifacts in SimStudio II

The grammar ware elements in the architecture are classified into two: *Executable models* and *transformation models*.

[14] http://www.omg.org/spec/XMI/; last accessed 3rd September, 2016

4.4.3.1 Executable models

The m_1 elements in this group are constitute another set of models of the SUS in the form of platform-specific executable program codes, based on some programming languages (PL) such as Java, C++, C#, Python, etc., to realize the corresponding MDSE methodologies. As you will see in the next sub subsection, they should be synthesized from the m_1 elements in the document ware group by model transformation processes.

A model in the document ware group may be used to drive the synthesis of executable models in multiple implementation platforms depending on the available tools. For instance, from *DEVS model* in the document ware, we should be able to generate executable models based on any available DEVS-based simulation framework irrespective of the programming language involved.

4.4.3.2 Transformation models

This group comprises the various artifacts that work together to effect the model transformation processes, which engineer the transmissions of system data captured in the *HiLLS model*(in model ware) through the m_1 elements of the document ware to the executable models in the grammar ware.

Each m_1 element in this group is a specification of a model transformation process written in a transformation language ($\in$ relations) and which conforms to (χ relations) a metamodel (of the transformation language), which is itself conformed to the Extended Backus Naur Form (EBNF) [Wir96].

In addition to the $\in$ and χ relations, a transformation model has three more relations with other artifacts in the framework: *dd*, *rr* and μ. Recall from our previous discussion in Section 3.4.3.5 that a model transformation model is, theoretically, a mathematical function with *domain* and *range* sets, which correspond to the languages used in writing the source and target models respectively. In the MDE context, a language is modeled by a metamodel. Relations *dd* (resp. *rr*) specify the model of the domain (resp. range) of a model transformation model. i.e., *dd* and *rr* refer to the metamodels to which the *source* (input) and *target* (output) models of a transformation process must conform. For instance, in Figure 4.4, the *dd* and *rr* of the model transformation model*DEVS synthesizer* are *HiLLS metamodel* and *DEVS schema* respectively. Note that model transformation model may have more than one *dd* and/or *rr* relations. An example of this in Figure 4.4 is the *Formal Analysis code generator*, which has two *dd* relations: $dd_1 = Zschema$ and $dd_2 = TL\ Pattern\ schema$.

In contrast to all other m_1 elements in the framework, which are representations of the SUS, the transformation models are representations of *model transformation processes*. i.e., the systems they represent are the actual in-memory runtime processes that generate the target models from the source models. Hence, the μ relations of the model transformation models in Figure 4.4 point to the in-memory transformation instances, which are sandwiched between the grammar ware and the document ware groups. Thus, we can describe these processes as the being in the m_0 layer to complete the metaization layer of the model transformation artifacts.

A *model transformation process* may have one or more of each of *s* and *t* relations, which point to the *source/input model (s)* and *target/output model(s)* respectively. For instance, in Figure 4.4, the model transformation process, *DEVS gen*, which is described by the *DEVS model synthesizer*, has an *s* relation with *HiLLS model* and a *t* relation with *DEVS model*. This implies that the process generates *DEVS model* from a *HiLLS model*. Similarly, the model transformation process *Simulation code gen*, described by the model transformation model *Simulation code generator*, takes a *DEVS model* as input (*s* relation) and produces a *Simulation code* as output (*t* relation).

4.5 SIMSTUDIO II PROCESS MODEL

In this section, we present a process model, which describes the workflow that may be followed by the different users described in Figure 4.3 to achieve their respective goals given an implementation of the framework architecture presented in the previous section. i.e., a model that guides the prospective users on how to use an implementation of the framework to realize the functional requirements presented earlier in Section 4.3.2.

The workflow is described by the UML activity diagram in Figure 4.5. Intuitively, an MDSE process should start with a model editing activity; the model being edited may be a newly created one or an existing model loaded from a repository. The case is not different with the proposed SimStudio II; we recall that the modeling language of the framework is the HiLLS. Hence, the user may start by creating a new HiLLS model or by loading an existing HiLLS model from the repository (this thesis does not dig into the model selection process to choose the model to be loaded from repository) and move to the *Edit System model* activity. This should be followed by validating the edited model against the HiLLS metamodel to ensure that it is well formed and that it conforms to the language's syntax rules and constraints. The model is considered *invalid* if an error is found during validation; otherwise, it is considered *valid*. The user can return to the *Edit System model* activity to correct the errors in the case of "validation errors" or proceed to the next activity.

Given a syntactically correct HiLLS model of the system under study, the next action depends on the goal of the user; it depends on whether the user wants to do simulation, formal analysis or enactment. We present the activities to each of the three goals in separate swim lanes named *Simulation Expert*, *Formal Analysis Expert* and *Enactment Expert* as described in Figure 4.5. We will discuss activities in each of the three swim lanes under separate headings in the rest of this section.

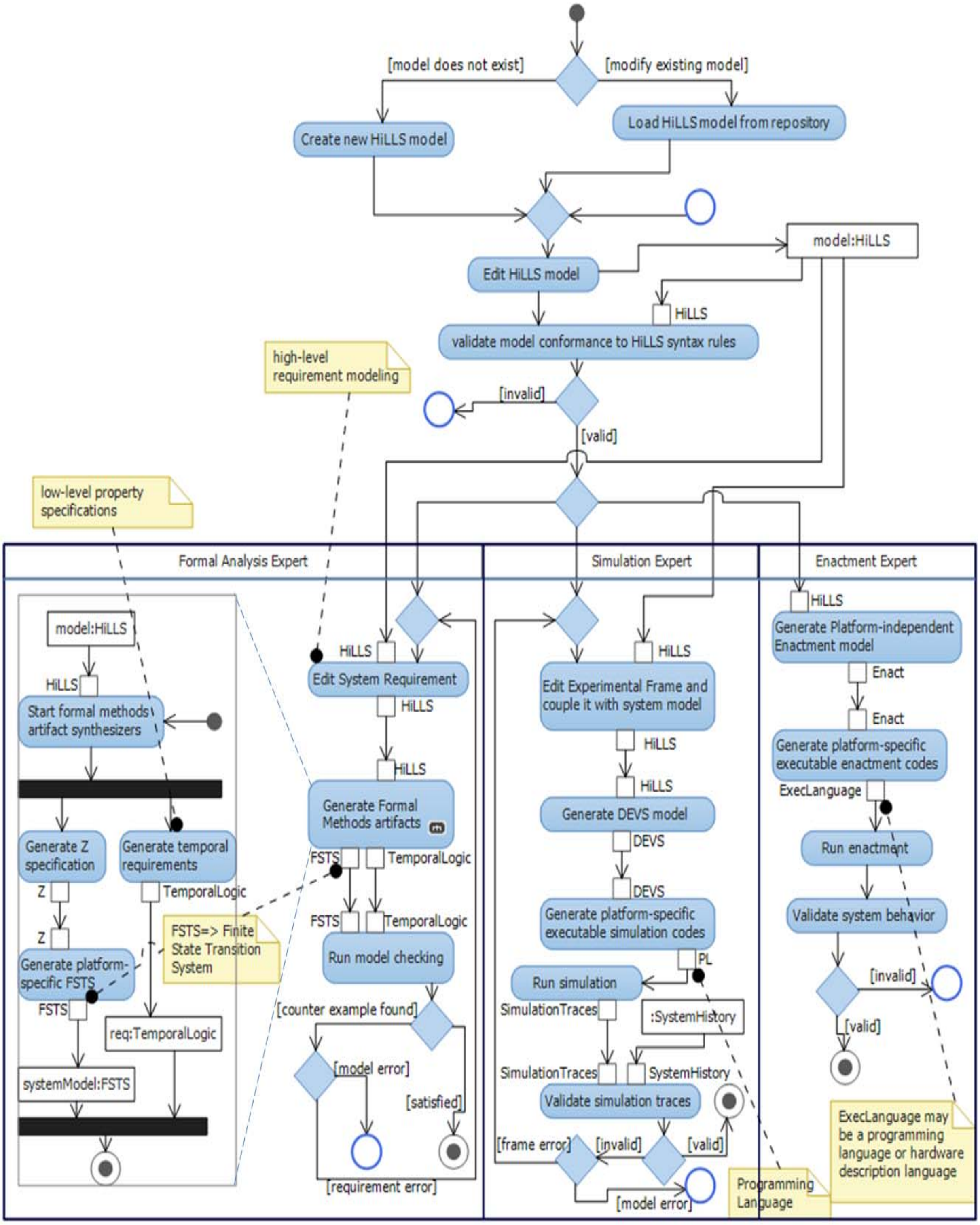


Figure 4.5 MDSE workflow in SimStudio II

4.5.1 Formal Analysis Activity

The formal analysis workflow proposed in the SimStudio II is described in the *Formal Analysis Expert* swim lane in Figure 4.5. It starts with the "*Edit System Requirement"* activity, which takes the system model obtained from the previous step as input. At this stage, the user can use the HiLLS-based notations for the TL property patterns presented in Section3.2.6.3 to model required behavioral properties of the system. The HiLLS model produced as output from this activity contains the *system model* and the *requirement model*.

The next activity, "*Generate formal methods artifacts"* is a function, which takes the HiLLS model from the previous activity as input and generates a finite state transition system (FSTS) and a TL specification as outputs; more details on the internal activity of the function are provided as a sub-activity diagram in the same swim lane.

The *"Run model checking"* activity is a representation of a model-checking tool, which takes the two outputs from the previous activity as inputs to perform an exhaustive exploration to verify that FSTS satisfies the properties specified in the TL specification. The process runs to termination if no counter examples are found in the model. A counter example, in this context, is an example of a violation of a property in the system model.

When the *"Run model checking"* activity produces a counter example, it is either there is an error in the system model or the requirement itself; it may even be that the requirement is not realistic and needs to the changed. In the case that the counterexample is due to an error in the system model, the user returns to the *"Edit system model"* activity to effect necessary changes and continue through the same swim lane. In the case that the counterexample is due to an error in the requirement, the user returns to the *"Edit system requirement"* to correct and/or vary the requirements. This procedure continues in iterations until all required properties are satisfied.

4.5.2 Simulation Activity

The simulation activity, described in the *"Simulation Expert"* swim lane starts with the *"Edit experimental frame and couple it with system model"* activity. The output obtained is passed to the *"Generate DEVS model"* activity to generate a platform-independent DEVS model of the coupled system and experimental frame models, which is then fed as an input to the *"Generate platform-specific executable simulation code"* activity to produce some program codes based on a DEVS-based simulation framework writing in programming language PL.

The program codes obtained from the previous stage are executed in the *"Run simulation"* activity to generate the simulation traces of the system model. The simulation traces are compares with the system's history in the *"Validate simulation traces"* activity to whether the behavior of the system model matches with that of the system under study or not. The process terminates if they match; otherwise, there is a problem in either the system model or experimental frame model. The process returns to the *"Edit system model"* activity if the problem is in the system model, otherwise it returns to the first activity in the swim lane. This procedure continues in iterations until the desired reports are obtained.

4.5.3 Enactment Activity

The enactment activity is described in the *"Enactment Expert"* swim lane starting with the *"Generate platform independent enactment model"* activity, which takes a HiLLS system model as input and produces a PIM enactment model. This is then used to generate a PSM enactment model in the next activity to obtain the executable program codes that are executed to enact the system under study during the *"Run enactment"* activity. Like in the other swim lanes, either the process runs to termination or iteratively returns to the *"Edit system model"* activity to refine the model until satisfactory behaviors are obtained.

4.6 CONCLUSION

In this chapter, we introduced the SimStudio II, an integrative MDSE framework to federate simulation, formal analysis and enactment analysis methodologies behind a unified modeling formalism. The chapter started with the introduction of the SimStudio project, a long-term research agenda that gave birth to this thesis, and a previous thesis that sought to address some of the issues identified in the initial research agenda.

Then we presented the methodology integration approach proposed in this thesis in comparison with the state of practice. The architecture of the proposed SimStudio II architecture was discussed in details in this chapter; we used megamodeling techniques to model the intricate relationships between the various software engineering artifacts in the framework. Finally, we presented a process model to guide the different users of the framework on how to use the framework given an implementation of the proposed architecture.

Having presented the framework's architecture and the roles of its various elements, the next line of actions is to elaborate on the different elements that have been presented in the architecture in black box views. Of course, the proposed architecture is itself another long-term research agenda that can hardly be fully implemented within a doctoral thesis. Nevertheless, we will present our designs of some of the key elements of the framework in subsequent chapters of the thesis. We have presented DEVS, a DES simulation paradigm, and some formal analysis formalisms in Section 3.2. However, little has been said about the enactment of DES in the literature; we start with the presentation of an evolving DEVS-based enactment framework in the next chapter before discussing the syntax and semantics of the framework's unified formalism - HiLLS - in subsequent chapters.

5 A DEVS-BASED ENACTMENT FRAMEWORK FOR DISCRETE EVENT SYSTEMS

5.1 INTRODUCTION

This chapter builds on the work we reported in [AMT15] to present a DEVS-based framework for enacting DESs. Discrete event simulation paradigms such as DEVS are very suitable for scenario-based analysis and verification of systems' behavioral properties. The simulation protocols execute system models using logical - rather than physical - advancements of execution time to the supposed times of occurrences of events of interest. With the use of logical time for the scheduling and execution of events, simulation processes can effectively forecast the future characteristics of a system's behavior.

Usually, the expected result of a simulation process can be in the form of trajectories of events like inputs, outputs and state transition events. The reconfigurations of state variables that lead to state transitions are considered instantaneous computations that occur just preceding the transitions. By courtesy of this time approximation of the real world, a simulation process can make substantial computational savings that it can run quite many times faster than the real world; this is considered as deliberately trading functional fidelity for scale and speed, in order to make the simulation task tractable in a reasonable time [BDM14]. For instance, the *state-preserving activities* (i.e., which do not result into a change of state) that occur during the stay of the system in some certain states are not of interest, and are not taken into consideration. However, the period of time required to perform such activities (e.g., time advance in DEVS) may be used to logically schedule the state transition event(s) that may occur at the end of the state's sojourn time while the system is considered to have logically (but not actually) stayed in the state for this period. Moreover, due to this time approximation, simulation processes do not allow live interaction with the physical environment, e.g., human-in-the-loop and hardware-in-the-loop. Thus, conventionally, all decision-making operations in a simulation process must be encoded in the model [BLC07].

Another computational analysis (cum implementation) methodology, which is more pronounced in business process management (BPM) [VTW03, JN14] than in systems engineering domain, is *enactment*. In the field of BPM, enactment may be simply described as the execution of process definitions created by a workflow [KGJ10] where a workflow is described as the complete or partial automation of business processes during which a set of procedure rules is used to pass information and work lists from one participant to another for necessary actions [OF07]. A more general software engineering description of the term enactment, provided in [DF94], is the execution or interpretation of software process definitions. According to the authors of [DF94], an enactment mechanism may also interact with the environment (e.g., human-in-the-loop, software and hardware devices) to provide supports that are consistent with the process definitions. This property of interaction with external actors is in fact another feature that differentiates enactment from mainstream simulation mechanisms in addition to the execution of system's functionalities in real time. Finally, in service engineering and Human-Computer Interaction, it can be inferred from [HE07] that enactment is used to describe the playing out of the functionalities represented by a prototype of a system. According to Holmquist [Hol05], a prototype is described as an object that represents the functionality but not the appearance of a finished artifact which can be used as a proof that a certain theory or concept or technology works or otherwise.

From the above descriptions, we describe enactment, in the context of systems engineering, as the execution of a software implementation of a system's behavior to verify its operational and functional characteristics in real clock time. To be able to verify a system's behavior in real time, there is need for an operational model of the system, which can be executed in a suitable software environment [BA95]. Analysis of traces generated from such executions can give further insights into the system's behavior as well as point out certain inconsistencies, missing requirements, verification of timing correctness in real-time systems etc.

An appreciation of the importance of system enactment can be seen from a closely related analysis methodology: *emulation*, which, according to Schiess [Sch01], is the marriage of simulation and controls designs to achieve ―virtual world” system operations. An emulation process is one in which a part of a real system is replaced by a model so that the functional parts of the process are carried out partly by the model and partly by some real systems. The model in the set up is expected to demonstrate functional fidelity by replicating the real world sufficiently faithfully that the connected equipment(s) cannot distinguish it from the real world it is standing in for [MGr02]. It is used, majorly, for testing process or control logics in the absence of the real facility in order to complete the logic testing in advance of the facility being built or modified [BDM14], and for risk-free trainings of the operators of a system [MGr02]. Some fundamental differences between simulation and emulation include [MGr02, BLC07, GRL05]:

- While simulation allows for the observance of the evolution of the internal states of a model in a predefined situation, emulation reproduces a system's dynamic interaction with its environment.
- A simulation process runs in virtual time; hence, the faster it is executed, the larger the search space it can explore in the same length of time. In contrast, an emulation process must execute in real time to interact with a system evolving in reality.
- All events that influence a simulation process are contained within the model, and are therefore repeatable. In contrast, absolute repeatability of the order of events is not

possible in an emulation process due to real time execution and, most commonly, a physically distributed computation infrastructure.

The enactment methodology we propose in this chapter shares, significantly, some philosophical and motivational bases with the emulation methodology. It is, however, slightly different in that:

- Its application is not specifically targeted at the domain of control systems, as is the case for emulation; rather we intend to apply it for any suitable DES.
- An enactment process does not necessarily consist of enactment model(s) and some real systems; rather, we could create enactment models for all components and execute everything as a software system. In fact, at this infant stage of our research in this direction, we are not yet considering software-hardware interfacing; though we are interested in human-in-the loop that is limited to live interactions with the running software through the general input devices like keyboard and mouse.

Using appropriate MDE techniques, we believe that such executable programs to enact systems' behaviors can be synthesized from models created in some modeling environments. Our target is to, systematically, derive enactment models from HiLLS-based models as described in the previous chapter (see Sections 4.3 and 4.4). Nevertheless, before then, some pertinent questions beg for answers: we must address questions such as "what formalism underlies the enactment model? What is the operational semantics of the model or its underlying formalism? ...". The operational semantics should precisely describe the real time execution of the enactment process. Our answers to these queries, for the moment, will be provided in an enactment framework proposed in the rest of this chapter.

In order to be general enough to meet the objective of accommodating a large category of DESs as stated previously, we extend DEVS to define the underlying formalism to express enactment models. We prefer DEVS for the same reason it is considered universal for expressing most kinds of DESs and even, approximated models for some kinds of non-DESs as demonstrated by Vangheluwe [Van00]. However, DEVS' operational semantics, which is a simulation protocol, was not defined to satisfy the enactment objective of this chapter. We explore the mapping of DEVS concepts onto the Object-Oriented (OO) *observer design pattern* defined by Gamma et al. [GHJ+95] to define semantics framework for enactment. We have chosen the observer design pattern to take benefit of its natural dialect for enacting the behavior of reactive systems and its ease of implementation in most general-purpose programming languages. This framework facilitates the synthesis (and specification) of operational (executable) representation of DES models for enactment processes. Of course, the pattern has some limitations that can potentially defeat its suitability for this purpose; we will discuss the most pertinent among them and the measures we have taken to palliate its potential effect on the accuracy of the order of events and computations.

Having earlier presented a detailed description of DEVS in Chapter 3 (Section 3.2.2), we present an extension of DEVS formalism to capture the concepts required for enactment in the next section. In Section 5.3, we present an overview of the observer pattern and its variant, which we have formulated for use in the proposed framework. This is followed, in Section5.4, by the specification and implementation of the enactment framework. Section 5.55.4presents the application of the framework to execute the enactment of the BVS running example before we conclude the chapter in Section5.6.

5.2 DEVS-BASED ENACTMENT FORMALISM

We recall from Section 3.2.2.1 that an atomic DEVS model is defined as a mathematical structure:

$$AM = < X, Y, S, \delta_{int}, \delta_{ext}, \delta_{conf}, \lambda, ta >$$

This definition provides abstract definitions of all constructs required to capture a DES for enactment except the functional activities of the states. As a reminder, a *functional activity* is a sequence of "state-preserving" operations that may be executed while a system is in a particular state. A state-preserving operation, in this context, is one that does not trigger a change of state; i.e., it does not modify any of the state variables, receive an input or produce an output. It may however use the instantaneous values of the variables in its computations. A typical example of an activity can be the display of caller's ID and the playing of some ringing tones when a cell phone is in the "incoming call" state. None of the two operations leads to a change of state since, despite their executions, the cell phone will remain in this state until the call is answered or the maximum waiting time set by the telecoms operator elapses.

To capture this concept of functional activity, we extend the atomic DEVS specification as follows:

$$AM_{enactment} \; = < X, Y, S, \delta_{int}, \delta_{ext}, \delta_{conf}, \lambda, ta, \mathbf{A}, \boldsymbol{\alpha} >$$

Where $\mathbf{A}$ is a set of operations and $\boldsymbol{\alpha}: \boldsymbol{S} \longrightarrow \{\boldsymbol{act_i} | \boldsymbol{act_i} \subset \mathbf{A}\}_{i \geq 0}$ is a mapping of each state of the system to an ordered subset (possibly empty) of A**,** i.e., given $s \in S$ and $act \in$ A, $\alpha: s \mapsto act$specifies that the ordered operations in set act will be executed whenever the system is in state s. The definitions of all other elements of atomic and coupled DEVS structures are preserved.

5.3 OVERVIEW OF OBJECT-ORIENTED DESIGN PATTERNS

Design patterns in OO modeling are documented solutions to some recurring problems that can be reused to build solutions to similar problems. In this section, we present the overviews of two design patterns, from the popular Gang of four book [GHJ+95], which will used to define the metamodel of our enactment framework.

5.3.1 Observer Design Pattern

The observer pattern is a behavioral pattern for establishing relationships between objects at runtime such that changes in the state of an object (referred to as *subject*) trigger some actions in another (the *observer*). It is defined by the Gang of Four as a pattern that "*defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.*"

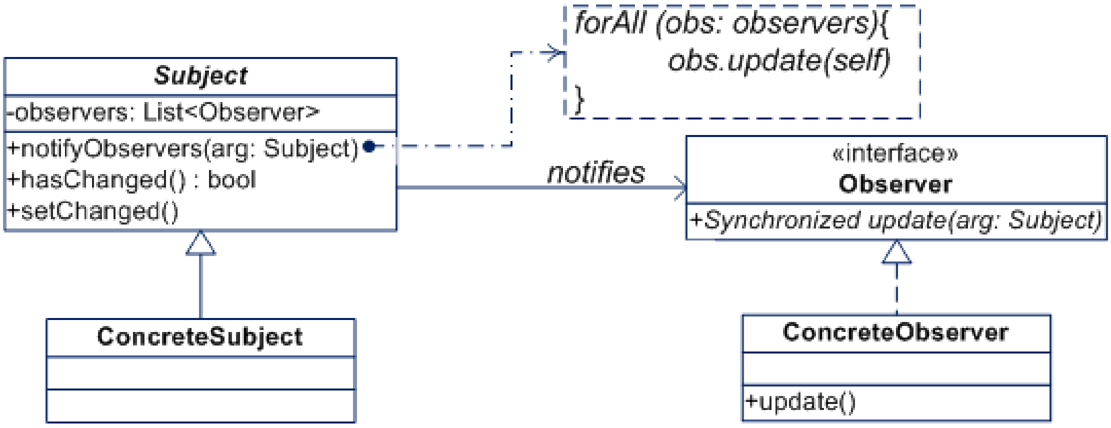


Figure 5.1Observer design pattern

Figure 5.1 describes the observer pattern. The basic idea is that *Subject* maintains a list of references (see attribute *observers*) to some independent objects called the *Observers*. Whenever there is a change of state in the subject, the operation *notifyObservers*() is executed, which notify all *observers* of *Subject* of its state transition the invocation of the *update* method of each of them. The notifications are done in a loop according to the small algorithm on the northeast region of Figure 5.1. Each observer (i.e., *ConcreteObserver*) must implement its update method to define the corresponding actions to be taken whenever a notification is received. This pattern is widely used in Graphical User Interface (GUI) programming and it provides the underlying principle for the Model-View-Controller (MVC) architecture [KP88] so that all views are automatically updated whenever there is a change of state in the model.

5.3.2 Command Design Pattern

The *command* design pattern is described in Figure5.2. A command in this context means a method call. The pattern provides a methodology to encapsulate a command in an object and issue it (the command) in such a way that the requested operation and the requesting object do not have to know each other.

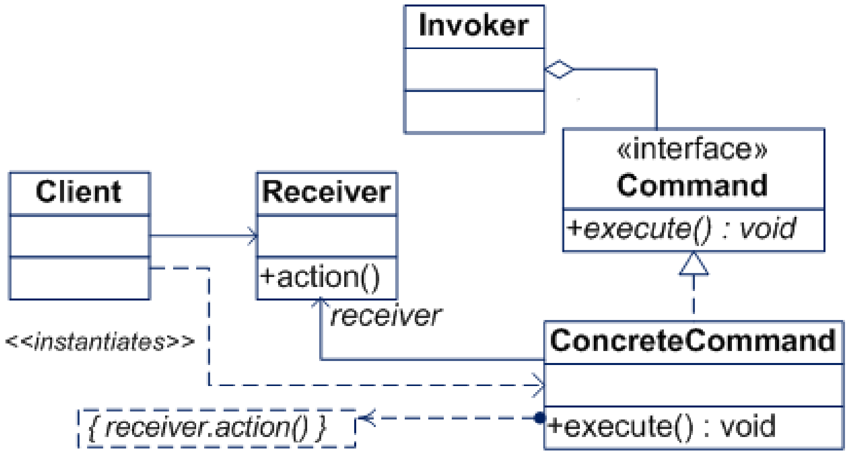


Figure5.2 Command design pattern

Client is the requesting object while the method *action*() of *Receiver* is the requested operation. *Client* creates the request command and delegates its execution to the *Invoker, which* manages a queue of command threads. The invoker identifies the receiver of the request carried by each command in its queue and then executes the command. When its "*execute()*" method is invoked, the command delivers its request by invoking the appropriate *action()* method. This pattern provides a methodology for asynchronous (i.e., non-blocking) method call, sharing of a method call among multiple objects, saving method calls in a queue so that they are executed when the necessary conditions have been satisfied, etc. It has also been used to decouple clients from server methods in Asynchronous Remote Method Invocation (ARMI) [RWB97].

5.3.3 Observer Pattern with Asynchronous Notifications for DES Enactment

The observer pattern offers a simple software design method to describe the exchange of messages and reactions to state transitions and reception of messages in reactive systems. For instance, in a network of interconnected system components, a component can influence some peer components when its state changes by notifying its influencee(s) with appropriate messages. An influenced component can also react to a received message/notification in its *update*() method.

We can observe, however, from the notification loop described in Figure 5.1 that the notification of each observer is done via a synchronous invocation of its *update*() method. This characteristic has two important effects that threaten the suitability of the pattern for the implementation of DESs:

i. When a subject notifies an observer, the processes in the former will be put on hold until the later executes its *update*() method and returns the control. It will even be more complicated if the observer is, itself, a subject to some other observers. This is a clear contradiction of the behavior of a DES; the exchanges of messages between components of a DES occur instantaneously and a sender does not have to keep track of how and when the message sent is processed. Thus, we must find a way to decouple the subject from the observer by doing the notifications asynchronously.

ii. When a subject has multiple observers, they will be notified sequentially, in a loop, in a non-determinate order. If we use this to implement a DES component sending a message to multiple destinations, the effect will be that, the message will not be received at some destinations until after some destinations have received and process it. The notifications of multiple observers must be done concurrently in order to conform to the through behavior of a DES.

We try to address these problems by using the command pattern to decouple the subject from its observer(s) during notifications. Figure 5.3 shows our attempt to introduce an asynchronous message passing between the subject and its observer(s) to make it more suitable for enacting systems' behaviors in real time. By comparing with the description of the command pattern in Figure5.2above, *Subject*, *Observer, Notifier, Notification* and *ConcreteNotification* are equivalent to *Client, Receiver, Invoker, Command* and *ConcreteCommand* respectively.

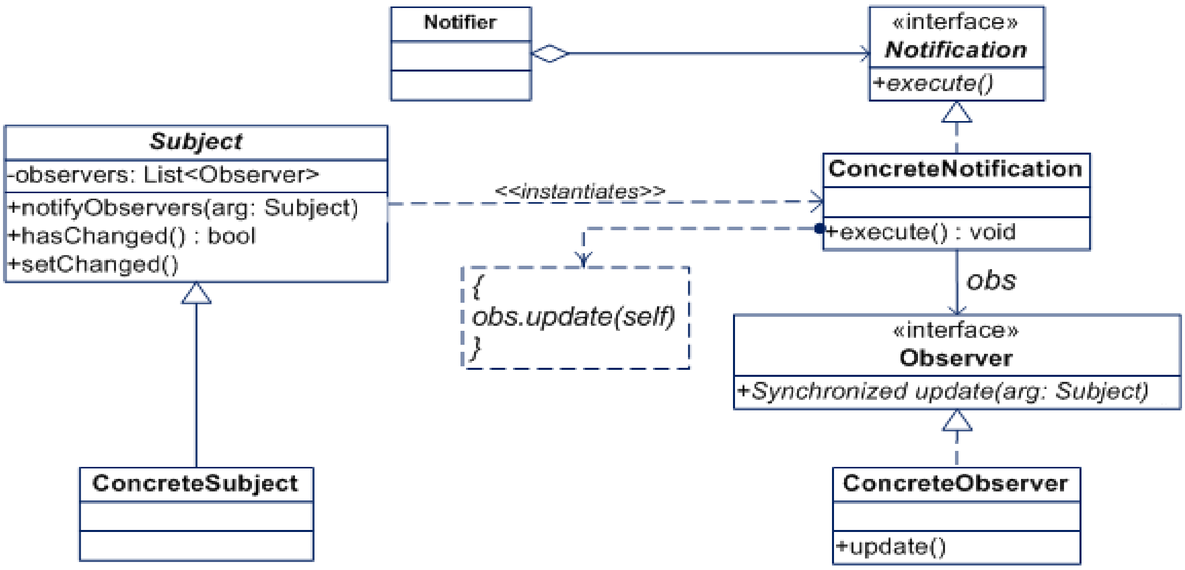


Figure 5.3 Observer design pattern with asynchronous notifications

Therefore, *Subject* will delegate the notifications of *observers* to *Notifier* and continue its activities. Since the subject does not expect any return value from these method calls, it is easy to use the "fire-and-forget" approach to solve problem (i). *Notifier* has a pool of threads to which the requests are assigned on arrival; hence, it does not create threads too often, thereby minimizing the overhead that may be incurred due to thread creation. In case of multiple observers to be notified of a change of sate, each notification request is assigned to a thread in the pool managed by the *Notifier*. With concurrent executions of the different notification threads, problem (ii) above is extremely mitigated if not completely solved.

5.4 ENACTMENT FRAMEWORK FOR DES

The methodology we propose is to use the dialect of observer design pattern to express DEVS-based concepts towards building a software framework for the enactment of DES. In this section we present the metamodel and enactment protocol of the framework and a Java-based implementation.

5.4.1 Metamodel of the Framework

We present the metamodel of the framework in Figure 5.4.In order not to inherit the limitations of the conventional observer pattern, the enactment framework reuses the observer pattern with asynchronous notifications presented earlier in Figure 5.3; this is represented by the classes *Subject*, *Observer, Notifier, Notification* and *ConcreteNotification,*on top of Figure 5.4, and the relationships between them.

The elements of the framework itself are described within the dashed box; the classes *ConcreteAtomicSystem* and *ConcreteCoupledSystem* at the bottom of the figure represent the enactment models of real systems that inherit the framework.

By virtue of their inheritance relationships with the *AbstractSystem*, *AbstractAtomicSystem* and *AbstractCoupledSystem* implement the *Observer* interface. Hence, both of them can be influenced by notifications from the objects they observe.

The *generic* class *Port* describes both input and output ports; the generic parameter T models the type of events admissible in the port, and must be provided at instantiation. Due to its relationships with *Subject* and *Observer*, a port can be an observer while it is itself an observable entity. Conceptually, the subject-observer relation can be used to express any kind of DEVS coupling. Recall that a DEVS coupling is a connection between two I/O ports in a coupled DEVS model for the purpose of exchange of messages/events so that whenever a message is placed on the source port, it is immediately transmitted to the target port. We express this in the enactment framework by registering the *target* port in the *observers* list of the source port so that whenever its value changes, it automatically notifies all target ports of all couplings in which it is the source with the new value. By the virtue of being a subject and an observer at the same time, a port may be involved in multiple couplings and play the role of source in some of the couplings while playing the role of target in the others.

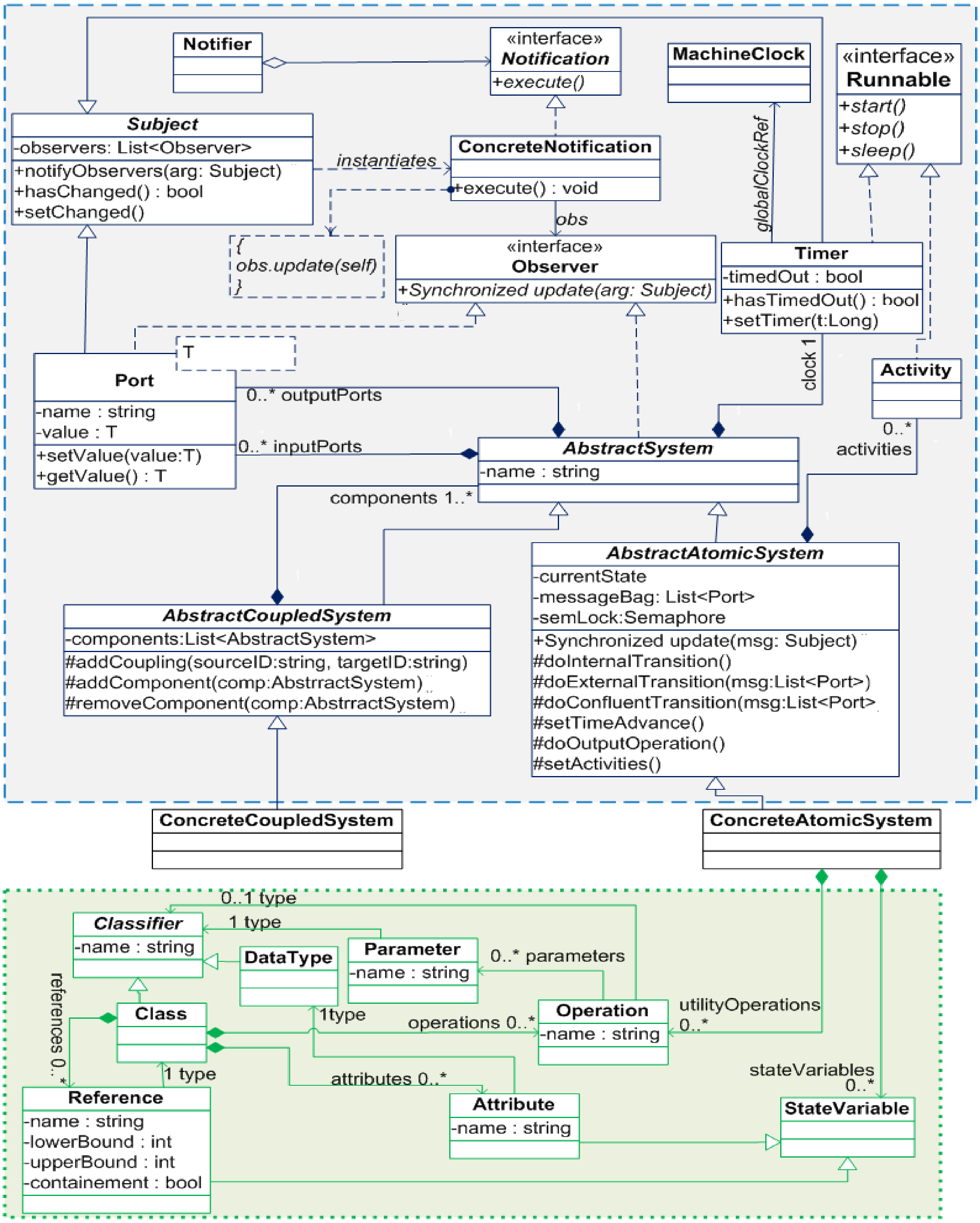


Figure 5.4 Metamodel of a DEV-based enactment framework for DES

An *AbstractAtomicSystem* is registered in the *observers* list of all its input ports; thus, the reception of an input event on an input port to naturally triggers an external state transition in the system that owns the port. i.e., an input port will automatically notify the host system whenever it receives an event. An *AbstractSystem* has a *clock* timer, which references the clock of the machine upon which it executes to manage the time advances of its states and the execution of activities. The *Timer* is an observable entity by virtue of its inheritance of the *Subject* class and only observer is its host system. Upon assumption of a new state, the system sets its *clock* to monitor the time advance of the state; once the time advance elapses, the *clock* sets its *timedOut* attribute to *true* and automatically sends notification to the system, thereby triggering an *internal state transition*. If the clock's notification coincides with that from an input port, a *confluent state transition* is triggered instead.

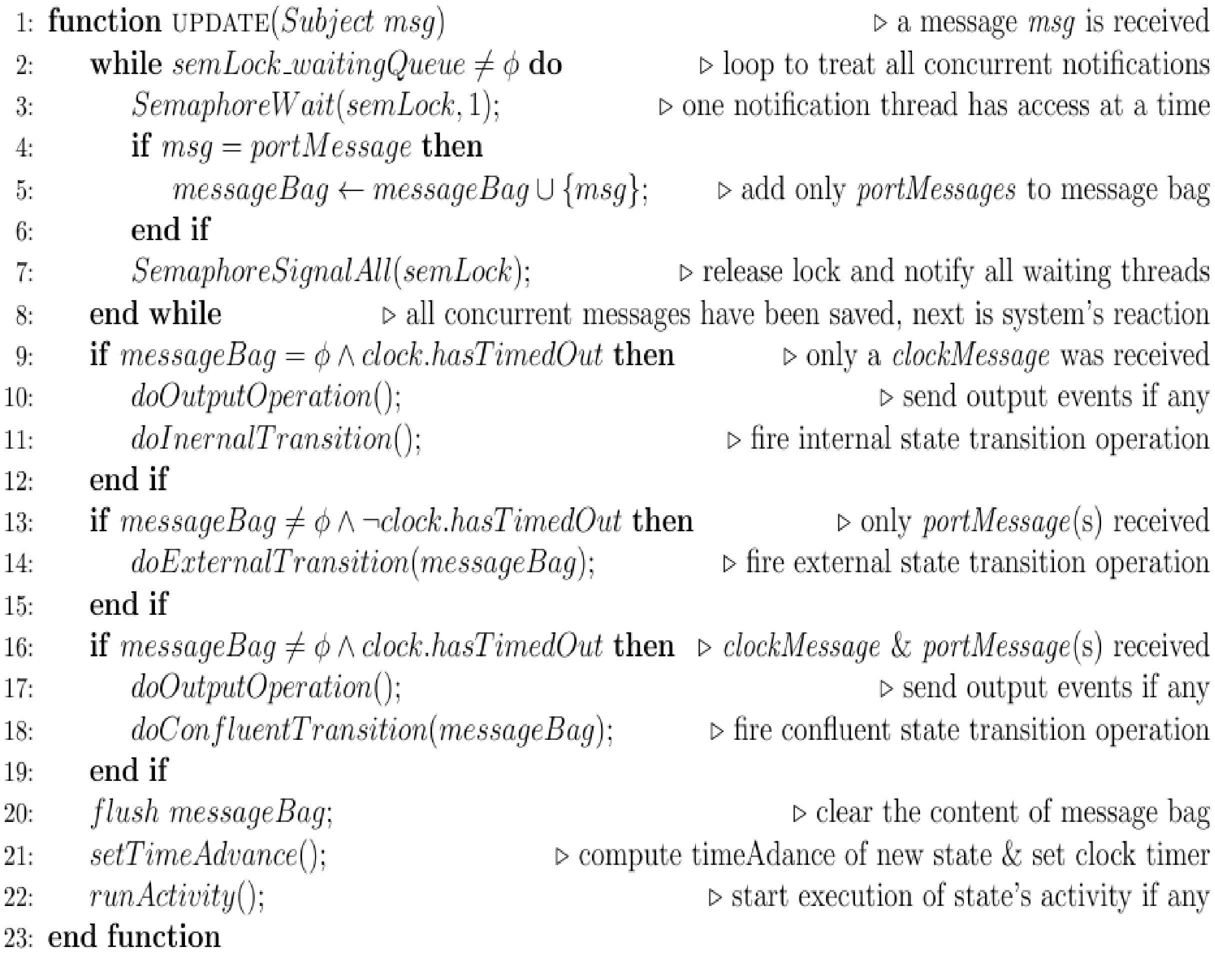
All methods in the *AtomicSystem* and *CoupledSystem* classes are abstract; therefore, the concrete atomic and coupled system classes using the framework must implement them to provide the specific elements of the system being modeled. They can also declare the necessary state variables (*stateVariables*) and user-defined special-purpose operations (*utilityOperations*) through a subset of the UML class framework described in the dotted box at the bottom of Figure 5.4above. *Utility operations* may be called from the framework-defined operations to do some computations. The *update* method of the *AbstractAtomicSystem* class implements the framework's enactment protocol, which will be provided in the next sub-section. The *doInternalTransition, doExternalTransition* and *doConfluentTransition* allow the user to describe the internal, external and confluent transition behaviors respectively. Similarly, *setTimeAdvance* and *setOutputEvents* methods must be implemented to provide the time advance and output functions respectively. Method *setActivities* can be used to define and associate activities to each state.

The modeler can specify a *coupling* by simply providing the names of the source and target ports to the *addCoupling*() method. The communications between the ports has been implemented in the framework based on their subject-observer relations.

5.4.2 Enactment Protocol

A state transition event occurs in an *AtomicSystem* whenever its *update*()method is invoked; this is typically when it receives notifications from the *subject(s)* it observes. As a reminder, an *AtomicSystem* is, by default, an observer of its clock and all its input ports; henceforth, we refer to the notifications received from the two as *clockMessage* and *portMessage* respectively. Technically, a *clockMessage* is received when the *clock* of the particular model has timed out (i.e., time advance of current state has elapsed) while a *portMessage* is transmitted from an input port upon the receipt of an external trigger. Since the notifications are independent, it is possible to receive multiple messages concurrently, which may consist of only *portMessages* (from different input ports) or a mixture of a *clockMessage* and one or more *portMessages*. Upon receipt, *portMessages* are "momentarily" stored in a global variable *messageBag* (see Figure 5.4, *messageBag* is an attribute of *AbstractAtomicSystem*); then the system's reaction will depend on the content of the bag, and whether the *clock*'s *timedOut* flag is *true* or *false*.

Figure 5.5 presents the *enactment protocol* of the atomic system, which is based on the receipt of, and reactions to port and clock messages. Recall that every *enactment model* is an "observer" (i.e., implements the Observer interface); thus such reactions are specified in its "update" method. Function *update* takes an argument *msg*, which is a *Subject* (line 1); recall from our descriptions of the observer pattern that the *subject* passes itself as an argument in the notifications to its *observers*.



```
 1: function UPDATE($Subject\ msg$)                              ▷ a message $msg$ is received
 2:     while $semLock\_waitingQueue \neq \phi$ do               ▷ loop to treat all concurrent notifications
 3:         $SemaphoreWait(semLock, 1)$;                        ▷ one notification thread has access at a time
 4:         if $msg = portMessage$ then
 5:             $messageBag \leftarrow messageBag \cup \{msg\}$;   ▷ add only $portMessages$ to message bag
 6:         end if
 7:         $SemaphoreSignalAll(semLock)$;                      ▷ release lock and notify all waiting threads
 8:     end while                    ▷ all concurrent messages have been saved, next is system's reaction
 9:     if $messageBag = \phi \wedge clock.hasTimedOut$ then           ▷ only a $clockMessage$ was received
10:         $doOutputOperation()$;                                      ▷ send output events if any
11:         $doInernalTransition()$;                          ▷ fire internal state transition operation
12:     end if
13:     if $messageBag \neq \phi \wedge \neg clock.hasTimedOut$ then         ▷ only $portMessage$(s) received
14:         $doExternalTransition(messageBag)$;               ▷ fire external state transition operation
15:     end if
16:     if $messageBag \neq \phi \wedge clock.hasTimedOut$ then  ▷ $clockMessage$ & $portMessage$(s) received
17:         $doOutputOperation()$;                                      ▷ send output events if any
18:         $doConfluentTransition(messageBag)$;             ▷ fire confluent state transition operation
19:     end if
20:     $flush\ messageBag$;                                        ▷ clear the content of message bag
21:     $setTimeAdvance()$;                      ▷ compute timeAdance of new state & set clock timer
22:     $runActivity()$;                                      ▷ start execution of state's activity if any
23: end function
```

Figure 5.5 Enactment protocol for atomic system models

Since multiple messages may be received simultaneously in the form of concurrent notifications, we need to enforce atomic access to the critical section of the update operation to ensure consistency of the shared variable, *messageBag*; this is realized with the acquisition of semaphore lock, *semLock*, in line 3 and its release in line 7.

In order to consider all concurrent messages in the system's reaction, they are all momentarily stored in a loop (see lines 2-8); *portMessages* are momentarily collected into the *messageBag* while the receipt of a *clockMessage* is remembered as long as the *timedOut* variable of the *clock* remains *true*. It is set to *true* whenever a scheduled time advance elapses and set to *false* whenever a new state is assumed and the new time advance scheduled. The loop in lines 2-8 iterates until no more messages are waiting to be recorded (i.e., to acquire the semaphore's lock).

Once the loop terminates, the system's state transition behavior follows from the content(s) of *messageBag*and the state of the clock's *timedOut* variable (inferred from the value returned by the function *clock.hasTimedOut()*) as follows:

i. When *messageBag* is empty and *timedOut* is *true*, lines 9-12, it implies that the notification is due to the expiration of the *timeAdvance* of the current state. Hence the system sends outputs (if any) to the appropriate output port(s) and immediately fires the internal state transition operation.
ii. When *messageBag* is not empty and *timedOut* is *false*, lines 13-15, this implies that the notification is due to the reception of a message on an(some) input port(s), which triggers an external state transition event in the system.
iii. When *messageBag* is not empty and *timedOut* is *true,*(lines 16-19), it implies that the expiration of the *timeAdvance* of the current state coincides with the arrival of a (some) message(s) at an (some) input port(s). The system reacts to this phenomenon by sending outputs (if any) on the appropriate output port(s) and follows it immediately with the confluent state transition operation.

Once the appropriate path of behavior has been chosen, the notification bag is cleared (line 20)in preparation for subsequent notifications. A new value of time advance is set (line 21) based on the specification of the new state. Technically, this is done by setting the *clock* to fire a notification on the expiration of the given time period; the clock resets its *timedOut* variable to *false* every time it sets a timer for new time advance. Finally, the activity execution is activated (line 23) based on the new state.

The enactment protocol for the coupled model is implicit in the coupling of appropriate ports based on the subject-observer relation of the observer pattern. Exchanges of messages between ports are automatically effected by courtesy of the subjects sending notifications to their observers.

5.4.3 An Implementation of the Enactment Framework

We have implemented the framework's metamodel and enactment protocol in Java. The UML package diagram in Figure 5.6 presents an overview of the implementation. The elements of the framework are structured into three packages: *enactment*, *enhancedObserverPattern* and *designException*. The framework takes benefit of some pre-defined infrastructure in Java, particularly the *Exception* class and *Runnable* interface in the *java.lang* package. As shown inFigure 5.6, *java.lang.Exception* is sub-classed to define some domain-specific exceptions used to signal violations of design constraints in the enactment model specified by the framework's user. Similarly, some elements of *enactment* and *enhancedObserverPattern* packages subclass *java.lang.Runnable* to provide the bases for the implementation of some concurrent and/or asynchronous operations in the enactment protocol. The elements of package *enactment* are presented in the rest of this subsection; we invite an interested reader to check Appendix A for the implementations of packages *designException* and *enhancedObserverPattern*.

To use the framework, we need to subclass *AbstractAtomicSystem* and *AbstractCoupledSystem* in package *enactment* and provide problem-specific implementations of their abstract methods. We will demonstrate this in the next section by using the framework to enact the running example in this thesis.

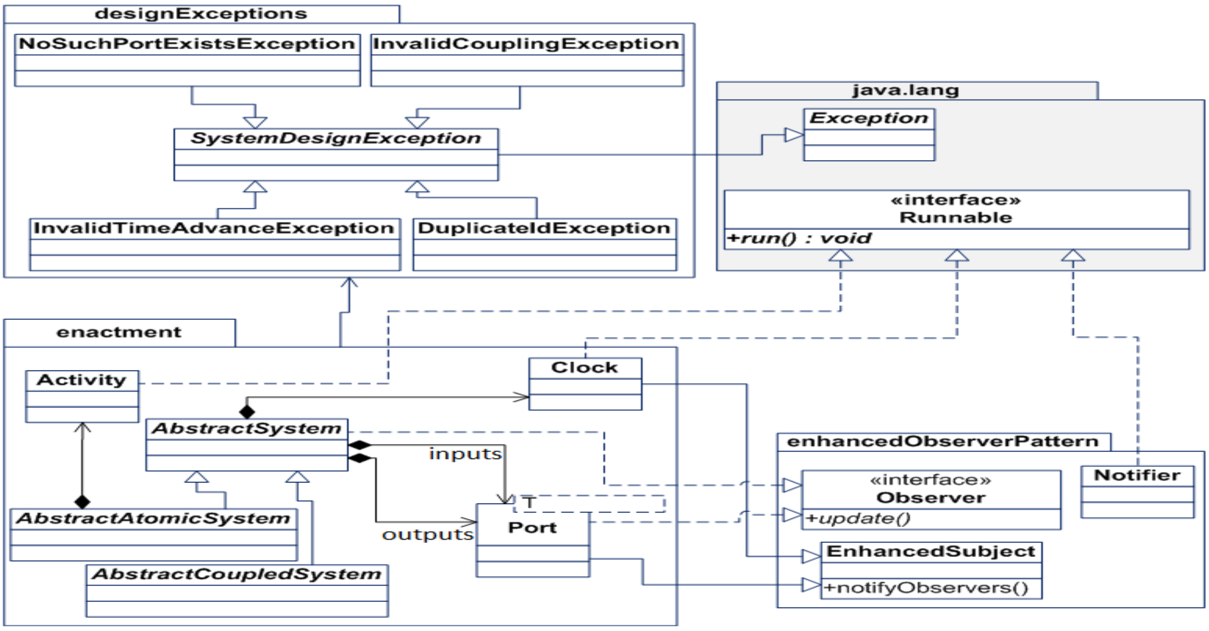
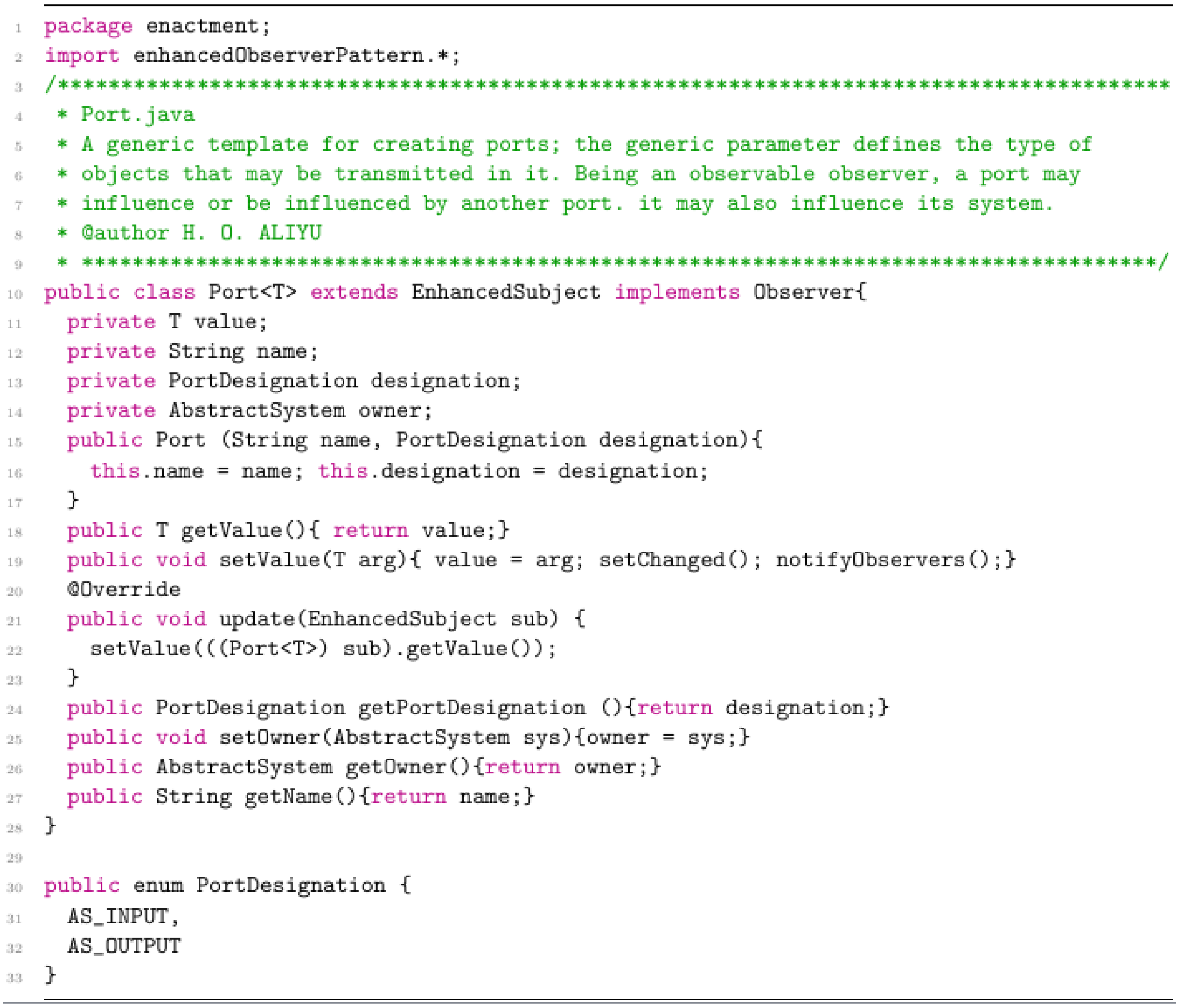


Figure 5.6Package diagram for a Java implementation of the enactment framework

5.4.3.1 Class Port[T]

Figure 5.7presents the Java implementation of the generic class *Port*. A port can be used either as input or as output as defined by the enumeration *PortDesignation* in lines 30-33. As port (either input or output) can listen to some observable entities and, itself, be listened to for state changes.

As specified in its *setValue*() method on line 19, whenever a port receives a new value, it automatically transmits the message to all its observers by invoking the *notifyObservers*() method. The details of the asynchronous notification of the observers are provided in the implementation of class *EnhancedSubject*.



```
1  package enactment;
2  import enhancedObserverPattern.*;
3  /*******************************************************************************************
4   * Port.java
5   * A generic template for creating ports; the generic parameter defines the type of
6   * objects that may be transmitted in it. Being an observable observer, a port may
7   * influence or be influenced by another port. it may also influence its system.
8   * @author H. O. ALIYU
9   * ****************************************************************************************/
10 public class Port<T> extends EnhancedSubject implements Observer{
11   private T value;
12   private String name;
13   private PortDesignation designation;
14   private AbstractSystem owner;
15   public Port (String name, PortDesignation designation){
16     this.name = name; this.designation = designation;
17   }
18   public T getValue(){ return value;}
19   public void setValue(T arg){ value = arg; setChanged(); notifyObservers();}
20   @Override
21   public void update(EnhancedSubject sub) {
22     setValue(((Port<T>) sub).getValue());
23   }
24   public PortDesignation getPortDesignation (){return designation;}
25   public void setOwner(AbstractSystem sys){owner = sys;}
26   public AbstractSystem getOwner(){return owner;}
27   public String getName(){return name;}
28 }
29
30 public enum PortDesignation {
31   AS_INPUT,
32   AS_OUTPUT
33 }
```

Figure 5.7Java implementation of generic class Port

5.4.3.2 Class AbstractSystem

```java
package enactment;
import enhancedObserverPattern.Observer;
import java.util.ArrayList;
import enactment.designExceptions.*;
/*********************************************************************************
 * AbstractSystem.java
 * AbstractSystem is the base class for modelling systems for enactment
 * It describes some general system's structural and behavioural properties
 * @author H. O. ALIYU
 * *******************************************************************************/
public abstract class AbstractSystem implements Observer{
  private ArrayList<Port> inputPorts; //input interface, a set of input ports
  private ArrayList<Port> outputPorts;//output interface, a set of output ports
  private String name;     //models system's ID
  private AbstractCoupledSystem container;
  public AbstractSystem(String name){
    this.name = name;
    inputPorts = new ArrayList<Port>();
    outputPorts = new ArrayList<Port>();
    validateInputOutputPorts();
  }
  /**Abstract methods to be implemented by sub-classes. registerInputOutputPorts() must
      be implemented in a concrete atomic or coupled model to register the system's
      input and output ports by calling methods addInputPort(String name, T type) and
      addOutputPort(String name, T type) respectively*/
  protected abstract void registerInputOutputPorts() throws DuplicateIdException;
  protected abstract void init(); //to initialize the model for enactment
  private void validateInputOutputPorts(){
    try {
      registerInputOutputPorts();
    } catch (DuplicateIdException e){e.printStackTrace();}
  }
  /** @throws DuplicateIdException when two input ports are identical*/
  protected final <T> void addInputPort(String name) throws DuplicateIdException{
    if (portExists(inputPorts, name))
      throw new DuplicateIdException("Duplicate input port: '"+name+"'");
    Port<T> newPort = new Port<T>(name, PortDesignation.AS_INPUT);
    newPort.addObserver(this);//a system is, by default, an observer of its 'input' ports
    newPort.setOwner(this); // a port is solely owned by one and only one system
    inputPorts.add(newPort);//add port to the input interface
  }
  /**@throws DuplicateIdException when two output ports are identical*/
  protected final <T> void addOutputPort(String name) throws DuplicateIdException{
    if (portExists(outputPorts, name))
      throw new DuplicateIdException("Duplicate output port: '"+name+"'");
    Port<T> newPort = new Port<T>(name, PortDesignation.AS_OUTPUT);
    newPort.setOwner(this);
    outputPorts.add(newPort);
  }
47   /**@throws NoSuchPortExistsException when the requested port is unknown to the system*/
48   public <T> Port<T> getInputPort(String name) throws NoSuchPortExistsException {
49     if (!portExists(inputPorts, name))
50       throw new NoSuchPortExistsException("Input port' "+name+"' does not exist");
51     Port<T> requiredPort = null;
52     for(Port<T> p:inputPorts)
53       if (p.getName().equals(name))
54         requiredPort=p;
55     return requiredPort;
56   }
57   /**@throws NoSuchPortExistsException when the requested port is unknown to the system*/
58   public <T> Port<T> getOutputPort(String name) throws NoSuchPortExistsException {
59     if (!portExists(outputPorts, name))
60       throw new NoSuchPortExistsException("Output port' "+name+"' does not exist");
61     Port<T> requiredPort = null;
62     for(Port<T> p:outputPorts)
63       if (p.getName().equals(name)) requiredPort=p;
64     return requiredPort;
65   }
66   public ArrayList<Port> getInputInterface(){ return inputPorts;}
67   public ArrayList<Port> getOutputInterface(){ return outputPorts;}
68   protected boolean portExists(ArrayList<Port> portList, String name){
69     for(Port p:portList)// checks whether there is a port registered with the given name
70       if (p.getName()==name) return true;
71     return false;
72   }
73   public String getName(){return name;}
74   public AbstractCoupledSystem getContainer(){return container;}
75   public void setContainer(AbstractCoupledSystem cont){container = cont;}
76 }
```

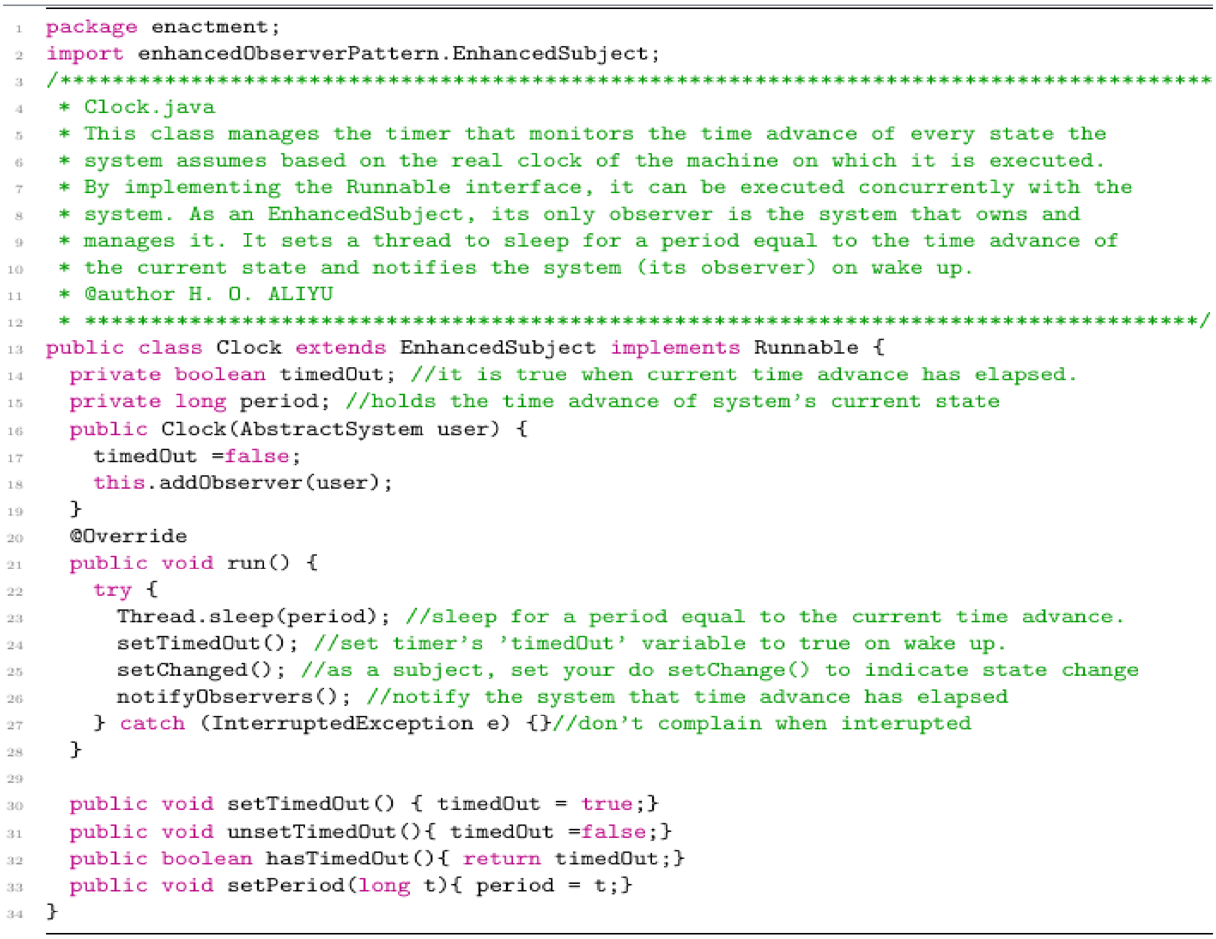
Figure 5.8 Java implementation of class AbstractSystem

The Java implementation of class *AbstractSystem* is presented in Figure 5.8. It implements structural properties that are common to both atomic and coupled systems. These are operations to register and validate input/output ports and setting a reference to the system's container if any. Note that upon the creation of an input port, lines 30-38, the system that owns it is immediately added to its list of observers. This is to ensure that the system is automatically notified every time the port receives a new value.

5.4.3.3 Class Clock

Figure 5.9below presents our Java implementation of class *Clock*. It implements interface *Runnable* so that its operations can be managed by process that starts and stops it when necessary. It also extends class *EnhancedSubject* so that it can be monitored, by some entities, for changes in its state. As specified in lines 16-19, immediately a system creates a clock, the former (referred to as *user* in lines 16 and 18) is registered as the sole observer of the latter so that the former can automatically react to changes of state in the latter.

The abstract method *run*() inherited from interface *Runnable* is implemented in lines 21-28; the method is invoked every time *user* assumes a new state. On invocation, the clock runs as a *thread*, which is scheduled to "sleep" for a physical period equal to the time advance of the current state of *user*. Upon wake up from sleep, clock immediately notifies user to take necessary action(s). *user* may also interrupt and shutdown the sleeping clock thread before its scheduled wake up time; technically, this interruption occurs just before an external state transition in *user*.



```
package enactment;
import enhancedObserverPattern.EnhancedSubject;
/*******************************************************************************
 * Clock.java
 * This class manages the timer that monitors the time advance of every state the
 * system assumes based on the real clock of the machine on which it is executed.
 * By implementing the Runnable interface, it can be executed concurrently with the
 * system. As an EnhancedSubject, its only observer is the system that owns and
 * manages it. It sets a thread to sleep for a period equal to the time advance of
 * the current state and notifies the system (its observer) on wake up.
 * @author H. O. ALIYU
 * ******************************************************************************/
public class Clock extends EnhancedSubject implements Runnable {
  private boolean timedOut; //it is true when current time advance has elapsed.
  private long period; //holds the time advance of system's current state
  public Clock(AbstractSystem user) {
    timedOut =false;
    this.addObserver(user);
  }
  @Override
  public void run() {
    try {
      Thread.sleep(period); //sleep for a period equal to the current time advance.
      setTimedOut(); //set timer's 'timedOut' variable to true on wake up.
      setChanged(); //as a subject, set your do setChange() to indicate state change
      notifyObservers(); //notify the system that time advance has elapsed
    } catch (InterruptedException e) {}//don't complain when interupted
  }

  public void setTimedOut() { timedOut = true;}
  public void unsetTimedOut(){ timedOut =false;}
  public boolean hasTimedOut(){ return timedOut;}
  public void setPeriod(long t){ period = t;}
}
```

Figure 5.9 A Java implementation of Clock class

5.4.3.4 Class AbstractAtomicSystem

Figure 5.10 below shows the implementation of class *AbstractAtomicSystem*. Upon creation (lines 27-27), an atomic system class creates a list of port references (*messageBag*) for momentary storage of "*portMessages*" during excitations, a Clock object (*clock*) to monitor the time advances of states as they evolve and a semaphore lock (*semLock*) to maintain the integrity of the contents of *messageBag* when multiple messages are received concurrently. When initialized (lines 28-31), the system initializes the state variables and invokes method *doTransition*() to assume the initial state. *initializeStateVariables*() is an abstract method so that the user of the framework can provide problem-specific implementations for each system.

Recall that the system is, by default, an observer of all its input ports and the sole observer of its clock and that its behavioral protocol is based on its reactions to automated notifications from these sets of subjects. This behavioral protocol is implemented in the method *update*() (lines 35-45) inherited from interface *Observer* via class *AbstractSystem*. Concurrent notification messages are first examined in a loop (lines 37-41) while all *portMessages* among them are stored in *messageBag* before invoking method *doTransition*() to react as described in the enactment protocol described in Section 5.4.2. *doTransition*() (lines 47-70) invokes methods *doOutputOperation*(), *doInternalTranstion*(), *doExternalTranstion*(), *doConfluentTranstion*() and *computeTimeAdvance*() when appropriate, all of which are abstract methods (lines 95-101) that require problem-specific implementations by the user of the framework. Immediately a state transition is complete, an asynchronous process is provided to run the activities of the new state (lines 31-32). Class *Activity* is implemented as an inner class in this class (see lines 103-110); when its *run*() method is executed, it runs the interruptible activities of the current state by invoking method *runActivities*() (line 108), which is also an abstract method (see line 100) to which the framework's user must provide a problem-specific implementation.

```
1  package enactment;
2  import java.util.ArrayList;
3  import java.util.concurrent.Semaphore;
4  import java.util.concurrent.Executors;
5  import java.util.concurrent.ExecutorService;
6  import enhancedObserverPattern.EnhancedSubject;
7  import enactment.designExceptions.*;
8  /*********************************************************************************
9   * AbstractAtomicSystem.java
10  * This class implements the enactment protocol of an atomic system.
11  * A concrete atomic class must extend this class and implement its abstract methods.
12  * @author H. O. ALIYU
13  * ******************************************************************************/
14 public abstract class AbstractAtomicSystem extends AbstractSystem {
15   private ArrayList<Port<?>> messageBag;//to store port messages temporarily
16   private Clock clock; //the clock that schedules and monitors time advances of states
17   private long timeAdvance; //a variable to hold the instantaneous time advance values
18   private Semaphore semLock; //a lock to enforce atomic write access to messageBag
19   private long startTime;// a variable to document when the system enters a state
20   private ExecutorService activityProcess;//a thread to run activities
21   private ExecutorService timeKeeper; //a thread to execute the clock timer when needed
22   public AbstractAtomicSystem(String name) {
23     super(name);
24     messageBag = new ArrayList<Port<?>>();
25     clock = new Clock(this);
26     semLock = new Semaphore(1, true);//atomic access with fairness to waiting threads
27   }
28   protected void init(){
29     initializeStateVariables();
30     doTransition();
31     activityProcess = Executors.newCachedThreadPool();
32     activityProcess.execute(new Activity());//execute the activities for new state
33   }
34   /**update method implements the enactment protocol of an atomic system.*/
35   @Override
36   public void update(EnhancedSubject subject){
    do { //concurrent messages are momentarily registered in messageBag before action
      semLock.tryAcquire();//ensure atomic access to the critical section
        {if (subject instanceof Port) messageBag.add((Port<?>) subject);}
      semLock.release();
    }while (semLock.hasQueuedThreads());//loop terminates when no thread is waiting
    doTransition(); //fire the appropriate state transition operation
    activityProcess = Executors.newCachedThreadPool();
    activityProcess.execute(new Activity());//execute the activities for new state
  }
  /**fire a state transition operation based on the state of messageBag and the clock*/
  private void doTransition() {
    if (clock.hasTimedOut() && messageBag.isEmpty()){ //only clock message received
      doOutputOperation();
      doInternalTranssition();
    }
    else if (clock.hasTimedOut() && !messageBag.isEmpty()){//clock and port messages
      doOutputOperation();
      doConfluentTransition(messageBag);
      flushMessageBag(); //clear the content of messageBag
    }
    else if (!clock.hasTimedOut() && !messageBag.isEmpty()){//port message(s) received
      interruptClock();                //stop the clock timer
      long eTime = System.currentTimeMillis()-startTime;//time spent in current state
      doExternalTransition(messageBag, eTime);
      flushMessageBag();
    }else { /*The system should never be in this situation */}
    try {
      setTimeAdvance(computeTimeAdvance()); //compute time advance for new state
    } catch (InvalidTimeAdvanceException e){ //time advance must be non-negative
      e.printStackTrace();
    }
    startClock(getCurrentTimeAdvance()); //start the clock to monitor the time advance
    startTime = System.currentTimeMillis();//time stamp of the beginning of the new state
  }
  public long getCurrentTimeAdvance(){ return timeAdvance;}
  protected final void setTimeAdvance(long duration) throws InvalidTimeAdvanceException{
    if (duration < 0)
      throw new InvalidTimeAdvanceException("Negative time advance in "+ this.getName());
    timeAdvance =duration;
  }
  private void flushMessageBag(){ messageBag.clear();}
  /** This method handles asynchronous delivery of an output event to an output port*/
  protected final void sendMessage(String portAddress, Object message){
    ExecutorService transporter = Executors.newCachedThreadPool();
    PostMaster postMaster = new PostMaster(portAddress, message);
    transporter.execute(postMaster);
    transporter.shutdown();
  }
  /** start the clock to monitor time advance duration => timeAdvance of current state*/
  public void startClock(long duration){
    clock.unsetTimedOut(); //reset the timedOut flag to false
    clock.setPeriod(duration);
    timeKeeper = Executors.newCachedThreadPool();
    timeKeeper.execute(clock);
  }
  /** This method is invoked just before an external transition to stop the clock*/
  public void interruptClock(){if(!timeKeeper.isTerminated())timeKeeper.shutdownNow();}
  /**The following abstract methods require problem-specific implementations*/
  protected abstract long computeTimeAdvance();
  protected abstract void doInternalTranssition();
  protected abstract void doExternalTransition(ArrayList<Port<?>> eventBag, long
      elapsedTime);
  protected abstract void doConfluentTransition(ArrayList<Port<?>> eventBag);
```

```java
  protected abstract void doOutputOperation();
  protected abstract void runActivities();
  protected abstract void initializeStateVariables();
  /**The runnable inner class Activity executes the state activities*/
  private class Activity implements Runnable {
    public Activity() {
    }
    @Override
    public void run() {
      runActivities();
    }
  }
  /**The runnable inner class PostMaster asynchronously puts messages on output ports*/
  private class PostMaster implements Runnable{
    String address; // id of the port on which output is sent
    Object message; // the value to be sent
    public <T> PostMaster(String address, T message){
      this.address = address; this.message = message;
    }
    @Override
    public void run() {
      try {
        getOutputPort(address).setValue(message);
      } catch (NoSuchPortExistsException e){e.printStackTrace();}
    }
  }
}
```

Figure 5.10 A Java implementation of class AbstractAtomicSystem

5.4.3.5 Class AbstractCoupledSystem

```java
package enactment;
import enactment.designExceptions.*;
import enhancedObserverPattern.EnhancedSubject;
import java.util.ArrayList;
/****************************************************************************************
 * AbstractCoupledSystem.java
 * This class implements the enactment protocol of a coupled system. It provides the
 * infrastructure to specify and validate the components of a composite system and their
 * coupling relationships. Every concrete coupled system model must extend this class
 * and provide the problem-specific implementations of its two abstract methods
 * @author H. O. ALIYU
 * ************************************************************************************/
public abstract class AbstractCoupledSystem extends AbstractSystem {
  private ArrayList<AbstractSystem> components;
  public AbstractCoupledSystem(String name) {
    super(name);
    components = new ArrayList<AbstractSystem>();
  }
  @Override
  public void init(){
    validateComponents();//ensures that there are no duplicate component identifiers
    validatePortCouplings();//ensure that all couplings satisfy the coupling constraints
    for (AbstractSystem comp: components) comp.init();//initialize all components
  }
  /**User must implement the following two abstract methods. every statement in
      registerComponents() uses addComponent(AbstractSystem sys) to register a component
      of the present system while each statement in registerPortCouplings() uses one of
      methods connectIC, connectEIC and connectEOC to specify couplings*/
  protected abstract void registerComponents() throws DuplicateIdException;
  protected abstract void registerPortCouplings() throws InvalidCouplingException,
      NoSuchPortExistsException;
/** addComponent(AbstractSystem sys) checks to ensure that 'sys' is not already a
    component of the present system before adding it to the list of components.*/
protected void addComponent(AbstractSystem sys) throws DuplicateIdException{
  if (this.getComponents().contains(sys)) throw new DuplicateIdException("Duplicate
      component identifier: "+ sys.getName()+" in "+ this.getName());
  components.add(sys);
  sys.setContainer(this);
}
/**It executes all the instances of addComponent(AbstractSystem sys) specified in
    registerComponents() & report any duplicate id found*/
protected void validateComponents(){
  try { registerComponents();
  } catch (DuplicateIdException e){e.printStackTrace();}
}
/** validatePortCouplings() executes all the instances of connectEIC, connectEOC and
    connectIC specified in registerPortCouplings() by the user and reports any
    violation of coupling constraints found*/
private void validatePortCouplings(){
  try {
    registerPortCouplings();
  } catch (InvalidCouplingException e) {e.printStackTrace();
  } catch (NoSuchPortExistsException e) {e.printStackTrace();}
}
/**if given parameters do not violate loop coupling constraint and specific EIC
    requirements,add the receiving port to the list of observers of sending port.
    otherwise, throw exception*/
protected final void connectEIC(AbstractSystem sender,String sendingPort,
    AbstractSystem receiver, String receivingPort)
    throws InvalidCouplingException, NoSuchPortExistsException{
  if(loopCoupling(sender,receiver)|invalidEIC(sender, sendingPort, receiver,
      receivingPort)) throw new InvalidCouplingException("Illegal EIC coupling:EIC
      requirements not satisfied");
  sender.getInputPort(sendingPort).addObserver(receiver.getInputPort(receivingPort));
}
/**if given parameters do not violate loop coupling constraint and specific EOC
    requirements, add the receiving port to the list of observers of sending port.
    otherwise, throw exception*/
protected final void connectEOC(AbstractSystem sender,String sendingPort,
    AbstractSystem receiver, String receivingPort)
    throws InvalidCouplingException,NoSuchPortExistsException{
  if(loopCoupling(sender,receiver)|invalidEOC(sender, sendingPort, receiver,
      receivingPort)) throw new InvalidCouplingException("Illegal EOC coupling:EOC
      requirements not satisfied");
  sender.getOutputPort(sendingPort).addObserver(receiver.getOutputPort(receivingPort));
}
/** if given parameters do not violate loop coupling constraint and specific IC
    requirements, add the receiving port to the list of observers of sending port.
    otherwise, throw exception*/
protected final void connectIC(AbstractSystem sender,String sendingPort,
    AbstractSystem receiver, String receivingPort)
    throws InvalidCouplingException, NoSuchPortExistsException{
  if(loopCoupling(sender,receiver)|invalidIC(sender, sendingPort, receiver,
      receivingPort)) throw new InvalidCouplingException("Illegal IC coupling:IC
      requirements not satisfied");
  sender.getOutputPort(sendingPort).addObserver(receiver.getInputPort(receivingPort));
}
/** in any kind of coupling, sender and receiver must be different*/
private boolean loopCoupling(AbstractSystem sender, AbstractSystem receiver){
  return (sender.equals(receiver))?true:false;
}
/**IC requirements: 1)sender and receiver must have the same container
 * 2) sending port is output port, 3) receiving port is input port*/
private boolean invalidIC(AbstractSystem sender, String sendingPort,AbstractSystem
    receiver, String receivingPort) {
  if(!(sender.getContainer().equals(receiver.getContainer()) &&
      sender.portExists(sender.getOutputInterface(),sendingPort)&&
    receiver.portExists(receiver.getInputInterface(), receivingPort))) return true;
  return false;
```

```
74   }
75   /**EIC requirements: 1)sender must be the container of receiver
76    * 2) sending port is input port, 3) receiving port is input port*/
77   private boolean invalidEIC(AbstractSystem sender,String sendingPort, AbstractSystem
         receiver, String receivingPort){
78     if(!(receiver.getContainer().equals(sender)&&(sender.portExists(sender.getInputInterface(),
79         sendingPort)&&receiver.portExists(receiver.getInputInterface(),receivingPort))))
             return true;
80     return false;
81   }
82   /**EOC requirements: 1)receiver must be the container of sender
83    * 2) sending port is output port, 3) receiving port is output port*/
84   private boolean invalidEOC(AbstractSystem sender, String sendingPort,AbstractSystem
         receiver, String receivingPort){
85     if(!(sender.getContainer().equals(receiver)&&(sender.portExists(sender.getOutputInterface(),
86         sendingPort)&&receiver.portExists(receiver.getInputInterface(),receivingPort))))
             return true;
87     return false;
88   }
89   /** returns a list containing all components of a composite system */
90   public ArrayList<AbstractSystem> getComponents(){ return this.components;}
91   @Override
92   public void update(EnhancedSubject subject){/*not required in a coupled system*/}
93 }
```

Figure 5.11 A Java implementation of class AbstractCoupledSystem

Finally, we present the implementation of class *AbstractCoupledSystem*in Figure 5.11. It implements the constructs and constraints for hierarchical modeling and enactment of composite systems.

Upon creation (lines 15-18), a coupled system class creates a list (*components*)to store the references to all its components. When initialized (lines 20-24), it validates the components and coupling specifications defined in a coupled system model and then initializes all its components. We believe the comments provided in the code itself are sufficient to aid the reader's understanding of the implementation. The next section demonstrates an application of the framework to the enactment of the BVS running example.

5.5 ENACTMENT OF THE BEVERAGE VENDING SYSTEM

In this section we present the application of the DEVS-based enactment framework introduced in this chapter to the modeling and enactment of our running example, the Beverage Vending System (BVS), (see Section 3.2.1). We will first present the enactment models based on the framework's implementation; and then the enactment traces obtained from their executions.

5.5.1 Enactment Models of the Beverage Vending System

We recall that the BVS was described in Section 3.2.1 as a composite system consisting of two atomic components: Beverage Vending Machine (BVM) the User. Hence the model will comprise two atomic system models, BVM and BVMUser, and one coupled system model BVS.

In addition, we have models of coin and beverage objects that are exchanged between the BVM and the BVMUser. In this chapter, we present the BVS and the most essential parts of BVM and BVMUser while the complete model is documented in Appendix A.

5.5.1.1 BVM enactment model

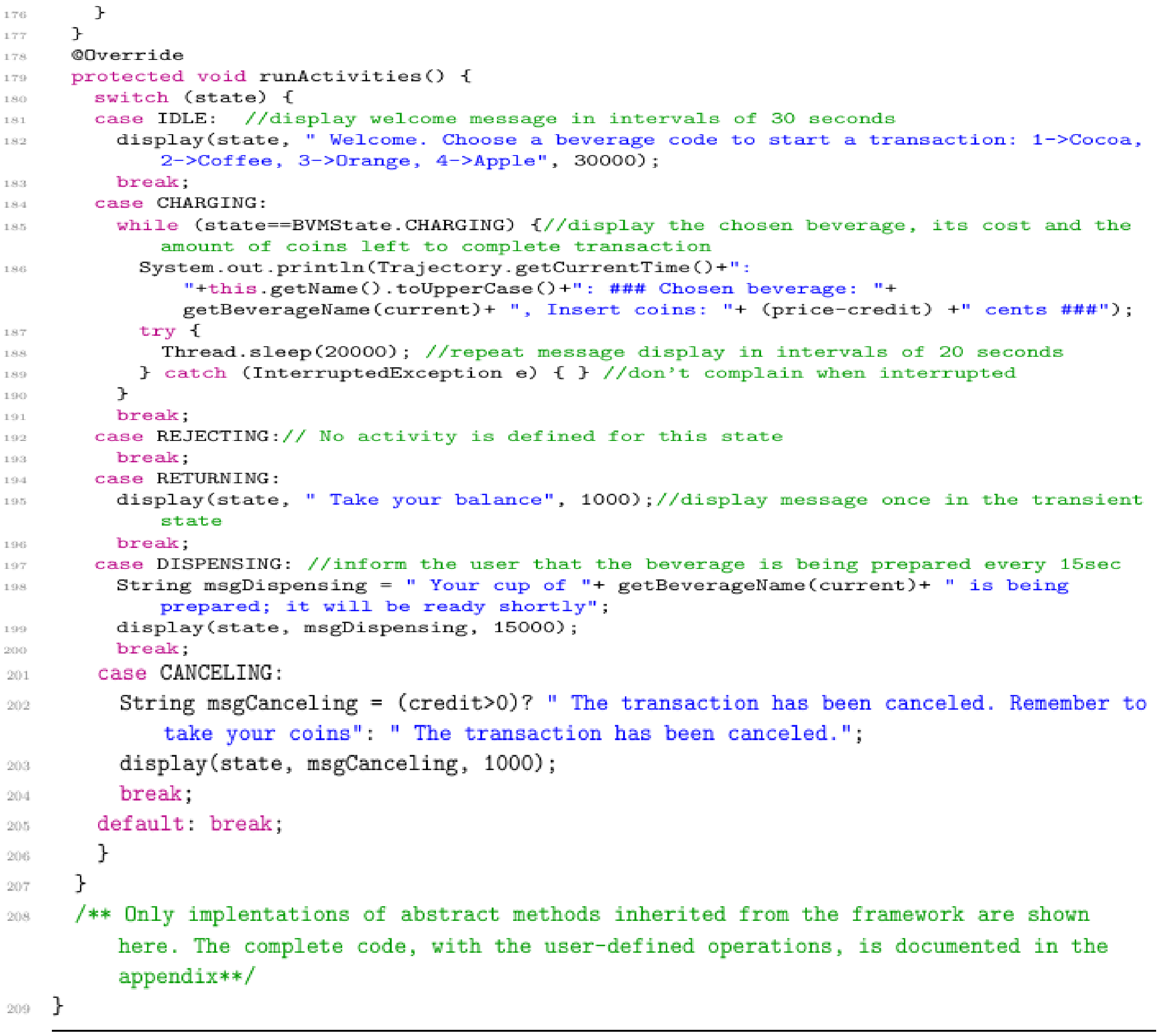
The Java class in Figure 5.12 presents the enactment model of the BVM. Being a model of an atomic system, class BVM extends the framework class *AbstractAtomicSystem* as suggested in the discussion of the framework in the previous section. BVM declares the state variables; we reuse the same state variables presented in the simulation model in Section 3.2.3.1. Recall that class *AbstractAtomicSystem*, as discussed previously in Section 5.4.3.4, declares some abstract methods that require problem-specific implementations; these are exactly the methods we implement in BVM. In order to implement the framework-based operations, we have defined some special-purpose methods, which are not declared in the framework, to perform some specific operations; such operations are presented in the full version of the model in Appendix A. We believe that the comments provided in the code, in addition to the description of the framework in the previous section, is sufficient to help the reader follow the system's structural and behavioral properties specified in the code.

```java
package bvs.enactment;
import java.util.ArrayList;
import java.util.Random;
import enactment.AbstractAtomicSystem;
import enactment.Port;
import enactment.designExceptions.DuplicateIdException;
/****************************************************************************************
 * BVM.java
 * An enactment model of the BVM based on the DEVS-based enactment framework.
 * It declares state variables and provide the problem-specific implementations of the
 * abstract methods specified in AbstractAtomicSystem.java.
 * @author H. O. ALIYU
 * *************************************************************************************/
public class BVM extends AbstractAtomicSystem {
  /*************** State variables *****************************************
   * state is a derived variable whose values in the enum BVMState depend on the
   * instantaneous values of the main state variables as defined in method setState()*/
  private int credit; //holds the cummulated value of coins accepted for a transaction
  private int price; //price of the transaction
  private int current; //code number of selected beverage
  private Coin badC;  //temporarily hold invalid (non-acceptable) coins
  private ArrayList<Coin> vault; //permanent storage for accepted coins
  private ArrayList<Coin> escrow; //interim storage for accepted coins in a transaction
  private BVMState state; //its value depends on the set of values of the state variables
  private enum BVMState {IDLE, CHARGING, DISPENSING, RETURNING, REJECTING, CANCELING};

  public BVM(String name) {
    super(name);
    vault = new ArrayList<Coin>();
    escrow = new ArrayList<Coin>();
  }
```

```
@Override
protected void registerInputOutputPorts() throws DuplicateIdException {
  super.<Coin>addInputPort("inC"); //to receive coins during transactions
  super.<Integer>addInputPort("code"); //to receive order and cancellation codes
  super.<Beverage>addOutputPort("cup"); //to deliver cups of beverages
  super.<ArrayList<Coin>>addOutputPort("outC"); //to deliver rejected or returned coins
}
@Override
protected long computeTimeAdvance() {
  switch (state) {
  case IDLE: return Long.MAX_VALUE;  //timeAdvace = +ve infinity
  case CHARGING: return 60*1000;   //timeAdvace = 1 minute (60 * 1000 milliseconds)
  case DISPENSING: return 45 * 1000;  //timeAdvace = 3/4 minute
  case RETURNING: return 0;    //timeAdvace = 0 minute
  case REJECTING: return 0;    //timeAdvace = 0 minute
  case CANCELING: return 0;    //timeAdvace = 0 minute
  default: return 0;
  }
}
@Override
protected void initializeStateVariables() {
  credit = 0; price = 0; current = 0; badC = null;
  initializeVault(30);//initialize vault with 30 randomly generated coins
  setState();
  /** The next two lines are used to print the system's traces during enactment*/
  System.out.println(Trajectory.getCurrentTime()+": "+this.getName().toUpperCase()+ ":
      Initialized to state: "+state);
  System.out.println(Trajectory.getCurrentTime()+": "+this.getName().toUpperCase()+ ":
      [current= "+current+ ", price= "+price+ ", credit= "+credit+ ", badCIsNull= "+
      (badC==null)+ ", vaultSize= "+vault.size()+", vaultValue= "+getBagValue(vault)+",
      escrowSize= "+escrow.size()+", escrowValue= "+getBagValue(escrow)+"]\n");
}
@Override
protected void doInternalTranssition() {
  BVMState sourceState = state;
  switch (state) {
  case CHARGING: current=5;break;//if no coin is received, auto-cancel the transaction
  case REJECTING: badC = null; break;
  case DISPENSING: current = 0; price=0; credit = 0; break;
  case CANCELING: current=0;price=0;credit=0;escrow.clear();badC=null;break;
  case RETURNING:credit -= (credit-price); break;
  default: break;
  }
  setState();
  /** The next two lines are used to print the system's traces during enactment*/
  System.out.println(Trajectory.getCurrentTime()+": "+this.getName().toUpperCase()+ ":
      [current= "+current+ ", price= "+price+ ", credit= "+credit+ ", badCIsNull= "+
      (badC==null)+ ", vaultSize= "+vault.size()+", vaultValue= "+getBagValue(vault)+",
      escrowSize= "+escrow.size()+", escrowValue= "+getBagValue(escrow)+"]");
  System.out.println(Trajectory.getCurrentTime()+": "+this.getName().toUpperCase()+ ":
      "+ sourceState + " -> "+state+"\n");
}
```

```
@Override
protected void doExternalTransition(ArrayList<Port<?>> eventBag, long elapsedTime){
  BVMState sourceState = state; // this is just to print the state trajectory
  switch (state) {
  case IDLE:
    if (eventBag.get(0).getName()=="code"){//input (portMessage) received on port code
      if ((Integer)eventBag.get(0).getValue()!=5) {//transaction code received
        current = (Integer)eventBag.get(0).getValue();
        setPrice(current);
        credit=0;
        System.out.println(Trajectory.getCurrentTime()+":
            "+this.getName().toUpperCase()+ ": Received transaction code "+ current);
      }
      else; //do nothing if canceling code is received in sate IDLE.
    }
    break;
  case CHARGING:
    if (eventBag.get(0).getName()=="code"){ //input (portMessage) received on port code
      if ((Integer)eventBag.get(0).getValue()==5)//canceling request received
        current = 5;
      else; // ignore any transaction request received while one is ongoing.
    }
    else { //input (portMessage) received on port inC
      Coin c = (Coin)eventBag.get(0).getValue();
      System.out.println(Trajectory.getCurrentTime()+":
          "+this.getName().toUpperCase()+ ": Received a coin of value: " +
          c.getValue()+ " cents");
      if (isAcceptable(c)) {//received coin is within the range acceptable to BVM
        escrow.add(c);   //temporarily store received coin in escrow
        credit+=c.getValue();  //update credit
        if (credit>=price) { //latest value of credit to complete the transaction
          vault.addAll(escrow); //transfer all coins in escrow to vault
          escrow.clear();
        }
        else; //accepted coins not sufficient to complete transaction
      }
      else //received coin is NOT within the range acceptable to BVM
        badC = c; //keep coin in badC momentarily to return it to the user.
    }
    break;
  default: break;
  }
  setState();
  /** The next two lines are used to print the system's traces during enactment*/
  System.out.println(Trajectory.getCurrentTime()+": "+this.getName().toUpperCase()+ ":
      [current= "+current+ ", price= "+price+ ", credit= "+credit+ ", badCIsNull= "+
      (badC==null)+ ", vaultSize= "+vault.size()+", vaultValue= "+getBagValue(vault)+",
      escrowSize= "+escrow.size()+", escrowValue= "+getBagValue(escrow)+"]");
  System.out.println(Trajectory.getCurrentTime()+": "+this.getName().toUpperCase()+ ":
      "+ sourceState + " --> "+state+"\n");
}
@Override
protected void doConfluentTransition(ArrayList<Port<?>> eventBag) {
  BVMState sourceState = state;
  switch (state) {
  case CHARGING:
    if (eventBag.get(0).getName()=="inC") {//input (portMessage) received on port inC
      Coin c = (Coin)eventBag.get(0).getValue();
      if (isAcceptable(c)&& credit+c.getValue()>=price) {
        if (!escrow.isEmpty())
          vault.addAll(escrow);
        escrow.clear(); vault.add(c); credit+=c.getValue();
      }
      if (isAcceptable(c)&& credit+c.getValue()<price) {
        credit += c.getValue(); escrow.clear(); current=5;
      }
```

```
        if (!isAcceptable(c)) {
          badC = c; current=5;
        }
      }
      break;
    default: break;
    }
    setState();
    /** The next two lines are used to print the system's traces during enactment*/
    System.out.println(Trajectory.getCurrentTime()+": "+this.getName().toUpperCase()+ ":
        [current= "+current+ ", price= "+price+ ", credit= "+credit+ ", badCIsNull= "+
        (badC==null)+ ", vaultSize= "+vault.size()+", vaultValue= "+getBagValue(vault)+",
        escrowSize= "+escrow.size()+", escrowValue= "+getBagValue(escrow)+"]");
    System.out.println(Trajectory.getCurrentTime()+": "+this.getName().toUpperCase()+ ":
        "+ sourceState + " .-.-> "+state+"\n");
  }
  @Override
  protected void doOutputOperation() {
    switch (state) {
    case DISPENSING:
      Beverage drink = new Beverage(current);
      sendMessage("cup", drink); //send out a cup of requested drink on port "cup"
      System.out.println(Trajectory.getCurrentTime()+": "+this.getName().toUpperCase()+
          ": Dispensed a cup of "+ drink.getContent());
      break;
    case REJECTING:
      ArrayList<Coin> msgBag = new ArrayList<Coin>();
      msgBag.add(badC);
      sendMessage("outC", msgBag);//wrap badC in a bag and send it out on port "outC"
      System.out.println(Trajectory.getCurrentTime()+": "+this.getName().toUpperCase()+
          ": Rejected a coin of value "+ badC.getValue());
      break;
    case RETURNING: //withdraw the balance from vault and send it out on port "outC"
      sendMessage("outC", removeChangeFromVault(credit-price));
      System.out.println(Trajectory.getCurrentTime()+": "+this.getName().toUpperCase()+
          ": Returned a balance of "+ (credit-price));
      break;
    case CANCELING:
      if (!escrow.isEmpty()||badC!=null) {
        ArrayList<Coin> refunds = new ArrayList<Coin>();
        if (!escrow.isEmpty()) refunds.addAll(escrow);
        if (badC!=null) refunds.add(badC);
        sendMessage("outC", refunds);//refund all coins received for the transaction
        System.out.println(Trajectory.getCurrentTime()+":
            "+this.getName().toUpperCase()+ ": Refunded a bag of coins of total value "+
            getBagValue(refunds));
      }
      break;
    default:
      break;
```



```
    }
  }
  @Override
  protected void runActivities() {
    switch (state) {
    case IDLE:  //display welcome message in intervals of 30 seconds
      display(state, " Welcome. Choose a beverage code to start a transaction: 1->Cocoa,
          2->Coffee, 3->Orange, 4->Apple", 30000);
      break;
    case CHARGING:
      while (state==BVMState.CHARGING) {//display the chosen beverage, its cost and the
          amount of coins left to complete transaction
        System.out.println(Trajectory.getCurrentTime()+":
            "+this.getName().toUpperCase()+": ### Chosen beverage: "+
            getBeverageName(current)+ ", Insert coins: "+ (price-credit) +" cents ###");
        try {
          Thread.sleep(20000); //repeat message display in intervals of 20 seconds
        } catch (InterruptedException e) { } //don’t complain when interrupted
      }
      break;
    case REJECTING:// No activity is defined for this state
      break;
    case RETURNING:
      display(state, " Take your balance", 1000);//display message once in the transient
          state
      break;
    case DISPENSING: //inform the user that the beverage is being prepared every 15sec
      String msgDispensing = " Your cup of "+ getBeverageName(current)+ " is being
          prepared; it will be ready shortly";
      display(state, msgDispensing, 15000);
      break;
    case CANCELING:
      String msgCanceling = (credit>0)? " The transaction has been canceled. Remember to
          take your coins": " The transaction has been canceled.";
      display(state, msgCanceling, 1000);
      break;
    default: break;
    }
  }
  /** Only implentations of abstract methods inherited from the framework are shown
      here. The complete code, with the user-defined operations, is documented in the
      appendix**/
}
```

Figure 5.12 Enactment model (code) of the BVM

5.5.1.2 BVMUser enactment model

Figure 5.13 presents the executable code for the enactment of the BVM's user following the state variables and behaviors specified in the simulation model (see Section 3.2.3.2). An alternative way to enact the BVM's user could be for us to execute the BVM model and interact directly with it at runtime. However, creating another atomic model would allow us to demonstrate how to create and enact coupled system models using the proposed framework.

```java
package bvs.enactment;
import java.util.ArrayList;
import java.util.Random;
import enactment.AbstractAtomicSystem;
import enactment.Port;
import enactment.designExceptions.DuplicateIdException;
/****************************************************************************************
 * BVMUser.java
 * An enactment model of the BVM's user based on the DEVS-based enactment framework.
 * It declares state variables and provide the problem-specific implementations of the
 * abstract methods specified in AbstractAtomicSystem.java.
 * @author H. O. ALIYU
 * *************************************************************************************/
public class BVMUser extends AbstractAtomicSystem {
  /*************** State variables ****************************************
   * state is a derived variable whose values in the enum UserState depend on the
   * instantaneous values of the main state variables as defined in method setState()*/
  private ArrayList<Coin> wallet; //a bag of coins to make transactions
  private int bill;     //cost of current transaction
  private int advance;   //total amount of coins expended on current transaction
  private int choice;    //reference code for an ordered beverage
  private Beverage cup; //holds a cup of beverage received at the end of a transaction
  private ArrayList<Coin> purse; //a bag to store coins rejected by BVM
  private UserState state;//value depends on the instantaneous values of state variables
  private enum UserState {AWAY, INSERTING, ORDERING, CANCELING, WAITING};

  public BVMUser(String name) {
    super(name);
    wallet = new ArrayList<Coin>();
    purse = new ArrayList<Coin>();
    cup = new Beverage();
  }
  @Override
  protected void registerInputOutputPorts() throws DuplicateIdException {
    super.<Beverage>addInputPort("drink");  //to receive cups of beverage
    super.<ArrayList<Coin>>addInputPort("inC"); //to receive bags of coins
    super.<Coin>addOutputPort("outC");    //to send out coins
    super.<Integer>addOutputPort("request");//to send out order and cancellation codes
  }
  @Override
  protected long computeTimeAdvance() {
    switch (state) {
    case AWAY: Random rand = new Random(System.currentTimeMillis());
         return (rand.nextInt(2)+2)*60*1000;  //timeAdvace >= 2 minutes
    case INSERTING: return 250*60;     //timeAdvace = 1/4 minute
    case WAITING: return 1500*60;     //timeAdvace = 1.5 minutes
    case CANCELING: return 0;      //timeAdvace = 0 minute
    default: return 0;
    }
```

```
}
@Override
protected void initializeStateVariables() {
  bill=0; advance=0; choice=0; cup = null; initializeWallet(20);//initialize wallet
      with 20 coins
  setState();
  /** The next two lines are used to print the system's traces during enactment*/
  System.out.println(Trajectory.getCurrentTime()+": "+this.getName().toUpperCase()+ ":
      Initialized to state: "+state);
  System.out.println(Trajectory.getCurrentTime()+": "+this.getName().toUpperCase()+ ":
      [choice= "+choice+ ", bill= "+bill+ ", advance= "+advance+ ", cupIsNull= "+
      (cup==null)+ ", walletSize= "+wallet.size()+", walletValue=
      "+getBagValue(wallet)+", purseSize= "+purse.size()+", purseValue=
      "+getBagValue(purse)+"]");//traces
}
@Override
protected void doInternalTranssition() {
  UserState sourceState = state; //used to print the state trajectory
  switch (state) {
  case AWAY:
    Random rand = new Random(System.currentTimeMillis());
    choice = rand.nextInt(4)+1; //decide on a beverage to order
    cup =null;
    break;
  case ORDERING:
    bill = getBill(choice); //bill depends on the chosen beverage
    break;
  case INSERTING:
    if (!wallet.isEmpty()){ //if wallet is not empty, pick a coin from it
      advance += wallet.get(0).getValue(); wallet.remove(0);
    }
    else {
      choice=5; //cancel transaction if wallet is empty
       System.out.println(Trajectory.getCurrentTime()+":
           "+this.getName().toUpperCase()+ " [Ran out of coins]");//traces
    }
    break;
  case CANCELING:
    if (advance>0) choice = 0; //advance>0 => some coins have already been inserted
    else choice = bill = 0;//advance==0 => no coin has been expended
    break;
  case WAITING:
    choice = 5; //cancel transaction if bvm fails to deliver order after a long wait
    break;
  default:
    break;
  }
  setState();
  /** The next two lines are used to print the system's traces during enactment*/
  System.out.println(Trajectory.getCurrentTime()+": "+this.getName().toUpperCase()+ ":
      [choice= "+choice+ ", bill= "+bill+ ", advance= "+advance+ ", cupIsNull= "+
      (cup==null)+ ", walletSize= "+wallet.size()+", walletValue=
      "+getBagValue(wallet)+", purseSize= "+purse.size()+", purseValue=
      "+getBagValue(purse)+"]");
  System.out.println(Trajectory.getCurrentTime()+": "+this.getName().toUpperCase()+ ":
      "+ sourceState + " -> "+state+"\n");
}
@Override
protected void doExternalTransition(ArrayList<Port<?>> eventBag, long elapsedTime) {
  UserState sourceState = state;
  switch (state) {
  case INSERTING:
    if (eventBag.get(0).getName()=="inC"){
      Port<ArrayList<Coin>> p = (Port<ArrayList<Coin>>) eventBag.get(0);
      ArrayList<Coin> rejectedCoins = (ArrayList<Coin>) p.getValue();
       System.out.println(Trajectory.getCurrentTime()+":
           "+this.getName().toUpperCase()+ ": Received coin(s) of total value "+
           getBagValue(rejectedCoins));
      advance-=getBagValue(rejectedCoins);
      purse.addAll(rejectedCoins);
    }
    break;
  case WAITING:
    if (eventBag.get(0).getName()=="inC"){
      Port<ArrayList<Coin>> pp = (Port<ArrayList<Coin>>) eventBag.get(0);
      ArrayList<Coin> balance = (ArrayList<Coin>) pp.getValue();
      System.out.println(Trajectory.getCurrentTime()+":
          "+this.getName().toUpperCase()+ ": Received balance coin(s) of total value "+
          getBagValue(balance));
      wallet.addAll(balance);
      advance-=getBagValue(balance);
      if (advance==0) {bill=0; choice=0;}
    }
    if (eventBag.get(0).getName()=="drink"){
      cup = (Beverage)((Port<Beverage>)eventBag.get(0)).getValue();
      System.out.println(Trajectory.getCurrentTime()+":
          "+this.getName().toUpperCase()+ ": Received a cup of "+cup.getContent());
      bill=0; advance=0; choice=0;
    }
    break;
  default:
    break;
  }
  setState();
  /** The next two lines are used to print the system's traces during enactment*/
  System.out.println(Trajectory.getCurrentTime()+": "+this.getName().toUpperCase()+ ":
      [choice= "+choice+ ", bill= "+bill+ ", advance= "+advance+ ", cupIsNull= "+
      (cup==null)+ ", walletSize= "+wallet.size()+", walletValue=
      "+getBagValue(wallet)+", purseSize= "+purse.size()+", purseValue=
      "+getBagValue(purse)+"]");
    System.out.println(Trajectory.getCurrentTime()+": "+this.getName().toUpperCase()+ ":
        "+ sourceState + " --> "+state+"\n");
  }
  @Override
  protected void doConfluentTransition(ArrayList<Port<?>> eventBag) {
    UserState sourceState = state;
    switch (state) {
    case INSERTING:
      if (eventBag.get(0).getName()=="inC")
        purse.add((Coin)eventBag.get(0).getValue());
      break;
    case WAITING:
      if (eventBag.get(0).getName()=="inC") // a bag of coins received on port "inC"
        wallet.add((Coin)eventBag.get(0).getValue());
      else      //a cup of beverage received on port "cup"
        cup = (Beverage)eventBag.get(0).getValue();
      break;
    default: break;
    }
    setState();
    /** The next two lines are used to print the system's traces during enactment*/
    System.out.println(Trajectory.getCurrentTime()+": "+this.getName().toUpperCase()+ ":
        [choice= "+choice+ ", bill= "+bill+ ", advance= "+advance+ ", cupIsNull= "+
        (cup==null)+ ", walletSize= "+wallet.size()+", walletValue=
        "+getBagValue(wallet)+", purseSize= "+purse.size()+", purseValue=
        "+getBagValue(purse)+"]");
    System.out.println(Trajectory.getCurrentTime()+": "+this.getName().toUpperCase()+ ":
        "+ sourceState + " .-.-> "+state+"\n");
  }
  @Override
  protected void doOutputOperation() {
    switch (state) {
    case ORDERING:
    case CANCELING:
      sendMessage("request", choice);
       System.out.println(Trajectory.getCurrentTime()+": "+this.getName().toUpperCase()+
           ": Sent request code "+ choice);//traces
      break;
    case INSERTING:
      if (advance<bill && !wallet.isEmpty()){
        sendMessage("outC", wallet.get(0));
        System.out.println(Trajectory.getCurrentTime()+":
            "+this.getName().toUpperCase()+ ": Sent a coin of value "+
            wallet.get(0).getValue());//traces
      }
      else; //No output if wallet is empty
      break;
    default: break;
    }
  }
  @Override
  protected void runActivities() {
    //No activities are specified for this component
  }

  /** Only implementation of abstract methods inherited from the enactment framework are
      shown here. The full code with the user-defined operations is presented in the
      appendix*/
}
```

Figure 5.13 Enactment model (executable code) of the BVM's user

5.5.1.3 BVS enactment model

```
1  package bvs.enactment;
2  import enactment.AbstractCoupledSystem;
3  import enactment.designExceptions.DuplicateIdException;
4  import enactment.designExceptions.InvalidCouplingException;
5  import enactment.designExceptions.NoSuchPortExistsException;
6  /*****************************************************************************************
7   * BVS.java
8   * An enactment model of the coupled beverage vending system. It creates the components
9   * and provides the problem-specific implementations of the abstract methods specified
10  * in AbstractCoupledSystem.java.
11  * @author H. O. ALIYU
12  * ************************************************************************************/
13 public class BVS extends AbstractCoupledSystem {
14   /*********** declare instances of BVM and BVMUser as components *************/
15   private BVM bvm;
16   private BVMUser user;
17
18   public BVS(String name) {
19     super(name);
20     bvm = new BVM("BVM");
21     user = new BVMUser("User");
22   }
23   @Override
24   protected void registerInputOutputPorts() throws DuplicateIdException {
25     // BVS is a closed system; it has no I/O ports
26   }
27   @Override
28   protected void registerComponents() throws DuplicateIdException {
29     addComponent(user);
30     addComponent(bvm);
31   }
32   @Override
33   protected void registerPortCouplings() throws
        InvalidCouplingException,NoSuchPortExistsException {
34     connectIC(user, "request", bvm, "code"); //output "request" -> input "code"
35     connectIC(user, "outC", bvm, "inC");  //output "outC" -> input "inC"
36     connectIC(bvm, "cup", user, "drink"); //output "cup" -> input "drink"
37     connectIC(bvm, "outC", user, "inC"); //output "outC" -> input "inC"
38   }
39 }
```

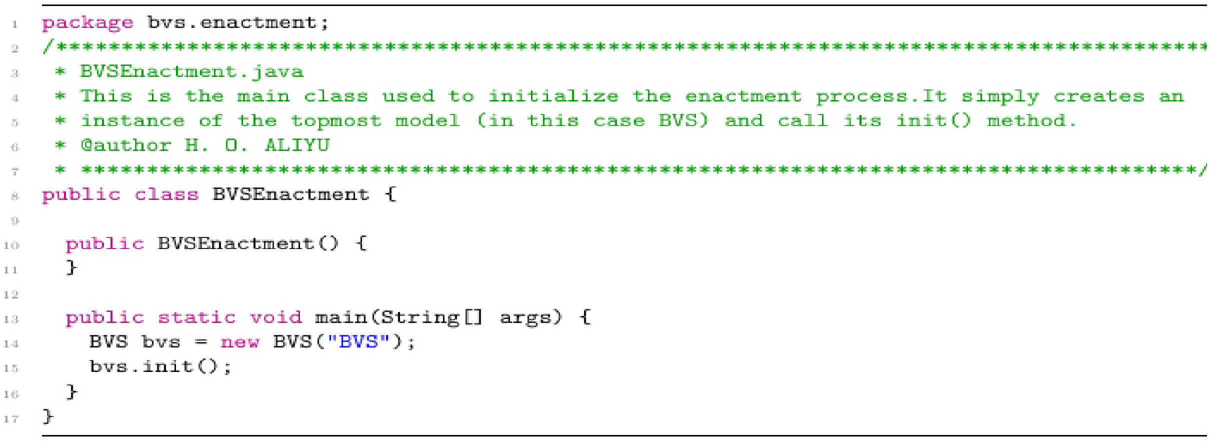
Figure 5.14 Enactment model of the BVS

Figure 5.14 presents the enactment code of the BVS. As a coupled system, BVS extends the framework's class *AbstractCoupledSystem*. BVS has two components, *bvm* and *user*, which are instances of BVM and BVMUser respectively; these components are registered in inherited method *registerComponents*() (lines 27-31). BVS is a closed system; hence, no input/output ports are registered. We specify four internal couplings in lines 32-38. The coupling in line 34 specifies that the output port "*request*" of component *user* be coupled with the input port "*code*" of component *bvm*; the other three couplings can be read in similar manner.

5.5.2 Enactment Execution and Enactment Traces

5.5.2.1 Initialization of the enactment process

The class *BVSEnactment* in Figure 5.15 defines the main method that initializes the enactment process. This is simply done by creating an instance of the BVS, which is the topmost system in the composition hierarchy of the model, and invoking its *init*() method. From this point, the enactment framework takes charge and initializes all components down the hierarchy tree.



```
package bvs.enactment;
/*************************************************************************************
 * BVSEnactment.java
 * This is the main class used to initialize the enactment process.It simply creates an
 * instance of the topmost model (in this case BVS) and call its init() method.
 * @author H. O. ALIYU
 * **********************************************************************************/
public class BVSEnactment {

  public BVSEnactment() {
  }

  public static void main(String[] args) {
    BVS bvs = new BVS("BVS");
    bvs.init();
  }
}
```

Figure 5.15Initialization of the enactment process of the BVS

5.5.2.2 Execution traces

We present excerpts from the traces of the enactment of seven consecutive transactions involving the interactions of the *BVM* and the *USER* in Figure 5.16 and Figure 5.17. The entire traces are documented in Appendix B.

The traces are presented in the format: `Time: System: Event` is the wall clock time of the occurrence of the event and is presented in the format `Hour:Minute:Second:Millisecond`. For instance; a trace`23:21:23:153: BVM:` E_i documents that an event E_i occurred in `BVM` at 23 hour, 21 minutes, 23 seconds and 153 milliseconds. The events are represented as follows:

Internal state transition events are presented in the format `SOURCE_STATE -> TARGET_STATE`. For instance, the trace "`23:23:23:154: USER: AWAY -> ORDERING`" documents an internal state transition from state `AWAY` to state `ORDERING` that occurred in `USER` at time `23:23:23:154`.

External state transition events are presented in the format `SOURCE_STATE--> TARGET_STATE`. For instance, the trace `23:23:23:168: BVM: IDLE --> CHARGING` documents that an external state transition from state `IDLE` to state `CHARGING` occurred in `BVM` at time `23:23:23:168`.

Confluent state transition events are in the format SOURCE_STATE -.-.> TARGET_STATE. There are no confluent transition events in the execution traces presented in Figure 5.16below.

Input and output events are described written in natural language. For example; "23:23:23:168: BVM: Received transaction code 3" documents an input event and 23:23:38:168: USER: Sent a coin of value 50 records an output event.

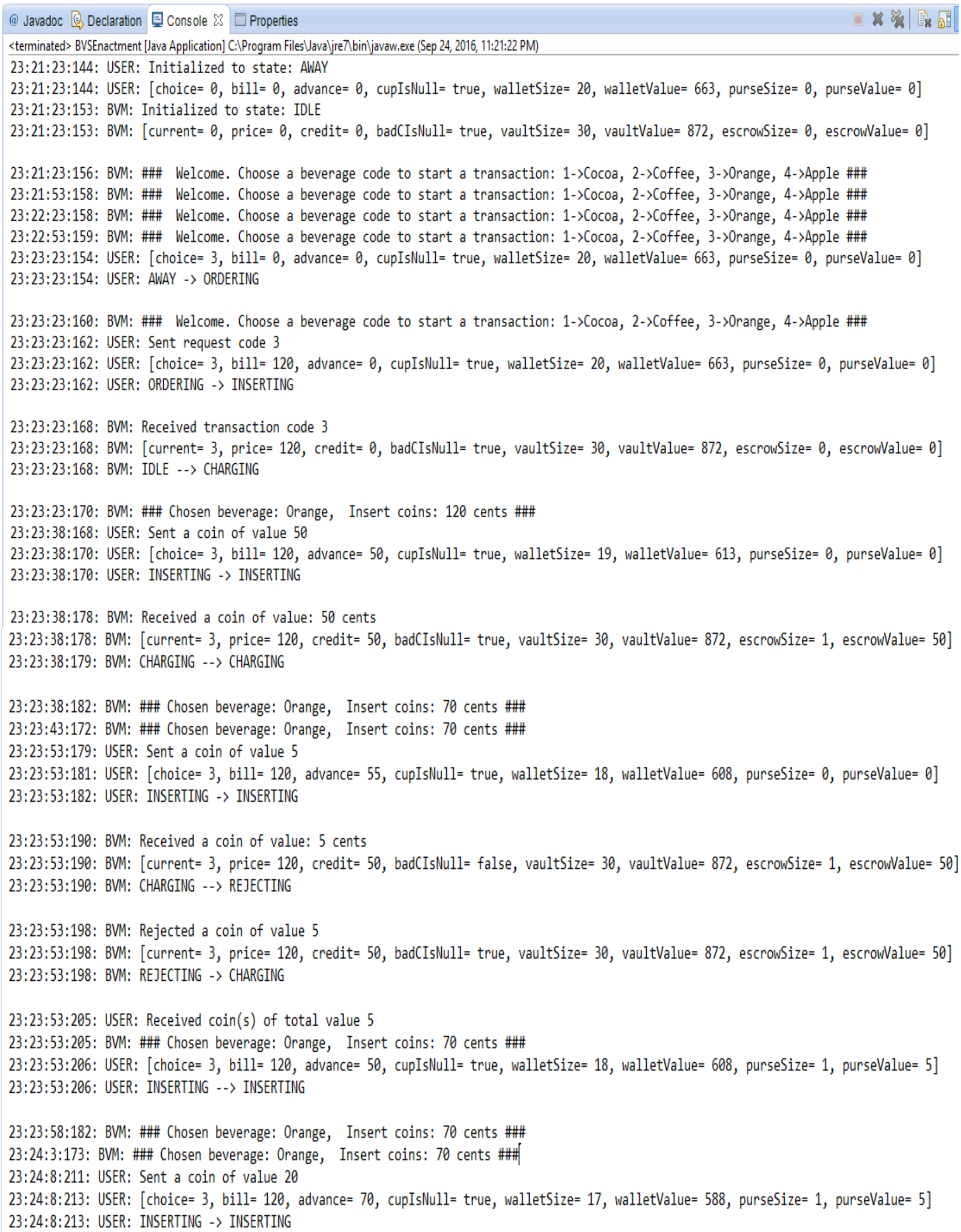
Instantaneous values of state variables are presented between square brackets; i.e. [and]. For example, "23:23:38:178: BVM:[current=3,price= 120,credit= 50,...]"documents the status of each state variable of BVM at the specified time.

Instantaneous activities are recorded between two groups of three hash symbols (###). All activities illustrated in this case study are in the form of displaying some messages; hence, the displayed messages are recorded between the hash symbols. For instance; 23:23:23:170: BVM: ###Chosen beverage: Orange, Insert coins: 120 cents###documents that at 23:23:23:170, BVM was displaying information about the chosen beverage for the ongoing transaction and the total value of coins expected to complete the transaction. 23:24:23:234: BVM: ###Take your balance###indicates that the BVM was informing the user that he/she has some balance from the ongoing transaction while 23:24:23:250: BVM: ###Your cup of Orange is being prepared; it will be ready shortly###shows that the system was displaying another information at time 23:24:23:250.

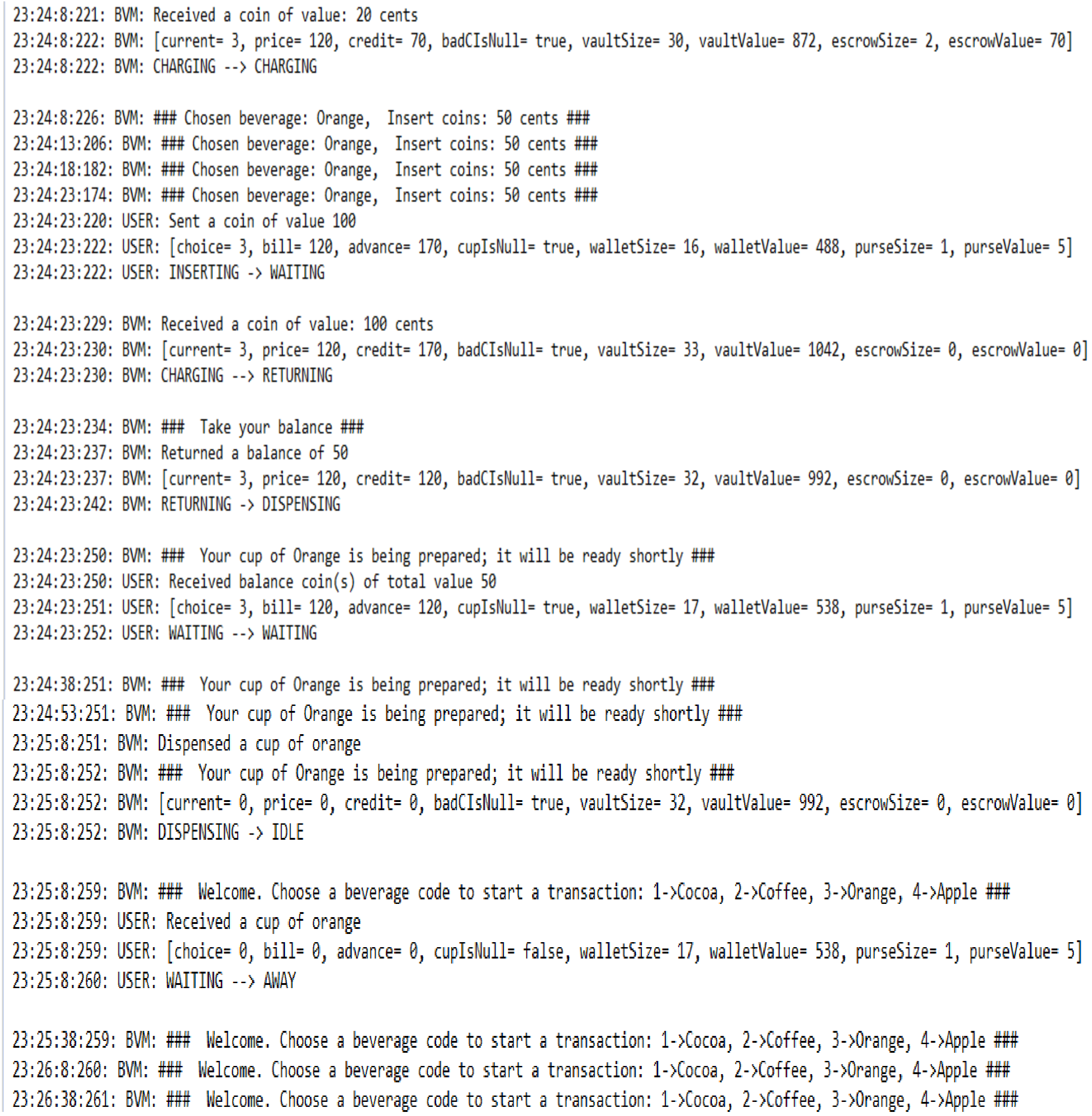
Having understood the format of the information presented in the traces, we can now discuss the traces themselves. We can see in Figure 5.16 that from the beginning that the enactment of USER (resp. BVM) started from the initial state AWAY (resp. IDLE) at 23:21:23:144 (resp. 23:21:23:153) with 20 (resp. 30) randomly generated coins in its *wallet* (resp. *vault*). The initial total value of all coins in the *wallet* (resp. *vault*) was 663 (resp. 872) cents. While in the IDLE state, BVM was displaying the welcome messages in intervals of 30 seconds as specified in the activity of state IDLE in enactment model. No activity was specified for USER.

The IDLE activity of BVM continued until 23:23:23:168, when it received a transaction code (3) which had been sent by USER about 6 milliseconds earlier (23:23:23:162) just before an internal state transition (ORDERING -> INSERTING) in the latter. The transaction code received in BVM (port message) triggered, an external transition, 23:23:23:168: BVM: IDLE --> CHARGING followed by the launching of another *activity*, at 23:23:23:170,which displays information about the requested beverage and the amount of coins left to complete the transaction. Note the evolutions of the state variables of the two subsystems.

USER released a 50-cent coin at 23:23:38:168, then updated its state variable *advance* and did an internal state transition, 23:23:38:170: USER: INSERTING -> INSERTING.



```
<terminated> BVSEnactment [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Sep 24, 2016, 11:21:22 PM)
23:21:23:144: USER: Initialized to state: AWAY
23:21:23:144: USER: [choice= 0, bill= 0, advance= 0, cupIsNull= true, walletSize= 20, walletValue= 663, purseSize= 0, purseValue= 0]
23:21:23:153: BVM: Initialized to state: IDLE
23:21:23:153: BVM: [current= 0, price= 0, credit= 0, badCIsNull= true, vaultSize= 30, vaultValue= 872, escrowSize= 0, escrowValue= 0]

23:21:23:156: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:21:53:158: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:22:23:158: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:22:53:159: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:23:23:154: USER: [choice= 3, bill= 0, advance= 0, cupIsNull= true, walletSize= 20, walletValue= 663, purseSize= 0, purseValue= 0]
23:23:23:154: USER: AWAY -> ORDERING

23:23:23:160: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:23:23:162: USER: Sent request code 3
23:23:23:162: USER: [choice= 3, bill= 120, advance= 0, cupIsNull= true, walletSize= 20, walletValue= 663, purseSize= 0, purseValue= 0]
23:23:23:162: USER: ORDERING -> INSERTING

23:23:23:168: BVM: Received transaction code 3
23:23:23:168: BVM: [current= 3, price= 120, credit= 0, badCIsNull= true, vaultSize= 30, vaultValue= 872, escrowSize= 0, escrowValue= 0]
23:23:23:168: BVM: IDLE --> CHARGING

23:23:23:170: BVM: ### Chosen beverage: Orange,  Insert coins: 120 cents ###
23:23:38:168: USER: Sent a coin of value 50
23:23:38:170: USER: [choice= 3, bill= 120, advance= 50, cupIsNull= true, walletSize= 19, walletValue= 613, purseSize= 0, purseValue= 0]
23:23:38:170: USER: INSERTING -> INSERTING

23:23:38:178: BVM: Received a coin of value: 50 cents
23:23:38:178: BVM: [current= 3, price= 120, credit= 50, badCIsNull= true, vaultSize= 30, vaultValue= 872, escrowSize= 1, escrowValue= 50]
23:23:38:179: BVM: CHARGING --> CHARGING

23:23:38:182: BVM: ### Chosen beverage: Orange,  Insert coins: 70 cents ###
23:23:43:172: BVM: ### Chosen beverage: Orange,  Insert coins: 70 cents ###
23:23:53:179: USER: Sent a coin of value 5
23:23:53:181: USER: [choice= 3, bill= 120, advance= 55, cupIsNull= true, walletSize= 18, walletValue= 608, purseSize= 0, purseValue= 0]
23:23:53:182: USER: INSERTING -> INSERTING

23:23:53:190: BVM: Received a coin of value: 5 cents
23:23:53:190: BVM: [current= 3, price= 120, credit= 50, badCIsNull= false, vaultSize= 30, vaultValue= 872, escrowSize= 1, escrowValue= 50]
23:23:53:190: BVM: CHARGING --> REJECTING

23:23:53:198: BVM: Rejected a coin of value 5
23:23:53:198: BVM: [current= 3, price= 120, credit= 50, badCIsNull= true, vaultSize= 30, vaultValue= 872, escrowSize= 1, escrowValue= 50]
23:23:53:198: BVM: REJECTING -> CHARGING

23:23:53:205: USER: Received coin(s) of total value 5
23:23:53:205: BVM: ### Chosen beverage: Orange,  Insert coins: 70 cents ###
23:23:53:206: USER: [choice= 3, bill= 120, advance= 50, cupIsNull= true, walletSize= 18, walletValue= 608, purseSize= 1, purseValue= 5]
23:23:53:206: USER: INSERTING --> INSERTING

23:23:58:182: BVM: ### Chosen beverage: Orange,  Insert coins: 70 cents ###
23:24:3:173: BVM: ### Chosen beverage: Orange,  Insert coins: 70 cents ###
23:24:8:211: USER: Sent a coin of value 20
23:24:8:213: USER: [choice= 3, bill= 120, advance= 70, cupIsNull= true, walletSize= 17, walletValue= 588, purseSize= 1, purseValue= 5]
23:24:8:213: USER: INSERTING -> INSERTING
```



```
23:24:8:221: BVM: Received a coin of value: 20 cents
23:24:8:222: BVM: [current= 3, price= 120, credit= 70, badCIsNull= true, vaultSize= 30, vaultValue= 872, escrowSize= 2, escrowValue= 70]
23:24:8:222: BVM: CHARGING --> CHARGING

23:24:8:226: BVM: ### Chosen beverage: Orange,  Insert coins: 50 cents ###
23:24:13:206: BVM: ### Chosen beverage: Orange,  Insert coins: 50 cents ###
23:24:18:182: BVM: ### Chosen beverage: Orange,  Insert coins: 50 cents ###
23:24:23:174: BVM: ### Chosen beverage: Orange,  Insert coins: 50 cents ###
23:24:23:220: USER: Sent a coin of value 100
23:24:23:222: USER: [choice= 3, bill= 120, advance= 170, cupIsNull= true, walletSize= 16, walletValue= 488, purseSize= 1, purseValue= 5]
23:24:23:222: USER: INSERTING -> WAITING

23:24:23:229: BVM: Received a coin of value: 100 cents
23:24:23:230: BVM: [current= 3, price= 120, credit= 170, badCIsNull= true, vaultSize= 33, vaultValue= 1042, escrowSize= 0, escrowValue= 0]
23:24:23:230: BVM: CHARGING --> RETURNING

23:24:23:234: BVM: ###  Take your balance ###
23:24:23:237: BVM: Returned a balance of 50
23:24:23:237: BVM: [current= 3, price= 120, credit= 120, badCIsNull= true, vaultSize= 32, vaultValue= 992, escrowSize= 0, escrowValue= 0]
23:24:23:242: BVM: RETURNING -> DISPENSING

23:24:23:250: BVM: ###  Your cup of Orange is being prepared; it will be ready shortly ###
23:24:23:250: USER: Received balance coin(s) of total value 50
23:24:23:251: USER: [choice= 3, bill= 120, advance= 120, cupIsNull= true, walletSize= 17, walletValue= 538, purseSize= 1, purseValue= 5]
23:24:23:252: USER: WAITING --> WAITING

23:24:38:251: BVM: ###  Your cup of Orange is being prepared; it will be ready shortly ###
23:24:53:251: BVM: ###  Your cup of Orange is being prepared; it will be ready shortly ###
23:25:8:251: BVM: Dispensed a cup of orange
23:25:8:252: BVM: ###  Your cup of Orange is being prepared; it will be ready shortly ###
23:25:8:252: BVM: [current= 0, price= 0, credit= 0, badCIsNull= true, vaultSize= 32, vaultValue= 992, escrowSize= 0, escrowValue= 0]
23:25:8:252: BVM: DISPENSING -> IDLE

23:25:8:259: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:25:8:259: USER: Received a cup of orange
23:25:8:259: USER: [choice= 0, bill= 0, advance= 0, cupIsNull= false, walletSize= 17, walletValue= 538, purseSize= 1, purseValue= 5]
23:25:8:260: USER: WAITING --> AWAY

23:25:38:259: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:26:8:260: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:26:38:261: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
```

Figure 5.16 Enactment traces of the BVS: Excerpt A

`USER` released a 50-cent coin at `23:23:38:168`, then updated its state variable *advance* and did an internal state transition, `23:23:38:170: USER: INSERTING -> INSERTING.` The trace `23:23:38:178: BVM: Received a coin of value: 50cents` shows that `BVM` received the coin about 10 milliseconds later, kept it in its *escrow* and update its *credit* (see the evolution of variables *credit*, *escrowSize* and *escrowValue* in the traces) and immediately did an external state transition `23:23:38:179: BVM: CHARGING --> CHARGING`. Though `BVM` was in state

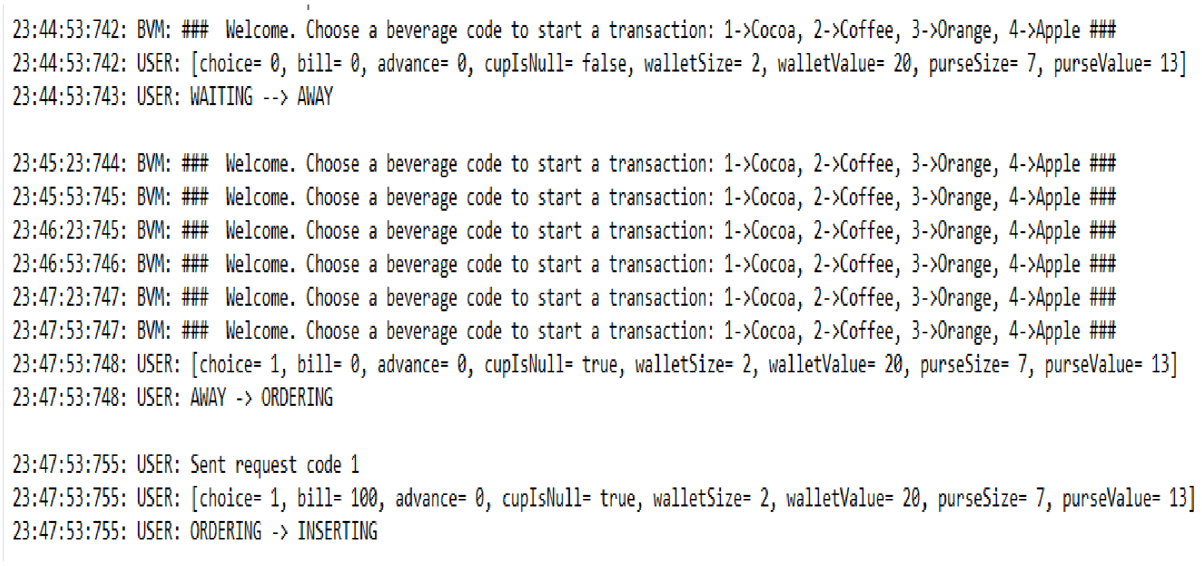
CHARGING at both times 23:23:23:170and 23:23:38:182, the state activities were slightly different due to the evolution of the system's state variables: while it displayed 120 cents at the former instant, the activity at the latter instant displayed 70 cents.

The 5-cent coin released by USER at 23:23:53:179was received by BVM at 23:23:53:190and momentarily kept in its *badC* variable instead of *escrow* (see the evolution of badCIsNull), followed immediately by an external state transition CHARGING --> REJECTING to expel the newly received coin. The rejected coin made its round trip back to USER at 23:23:53:205 and was immediately committed to the *purse* (see the evolution of purseSize and purseValue) while *advance* is updated.

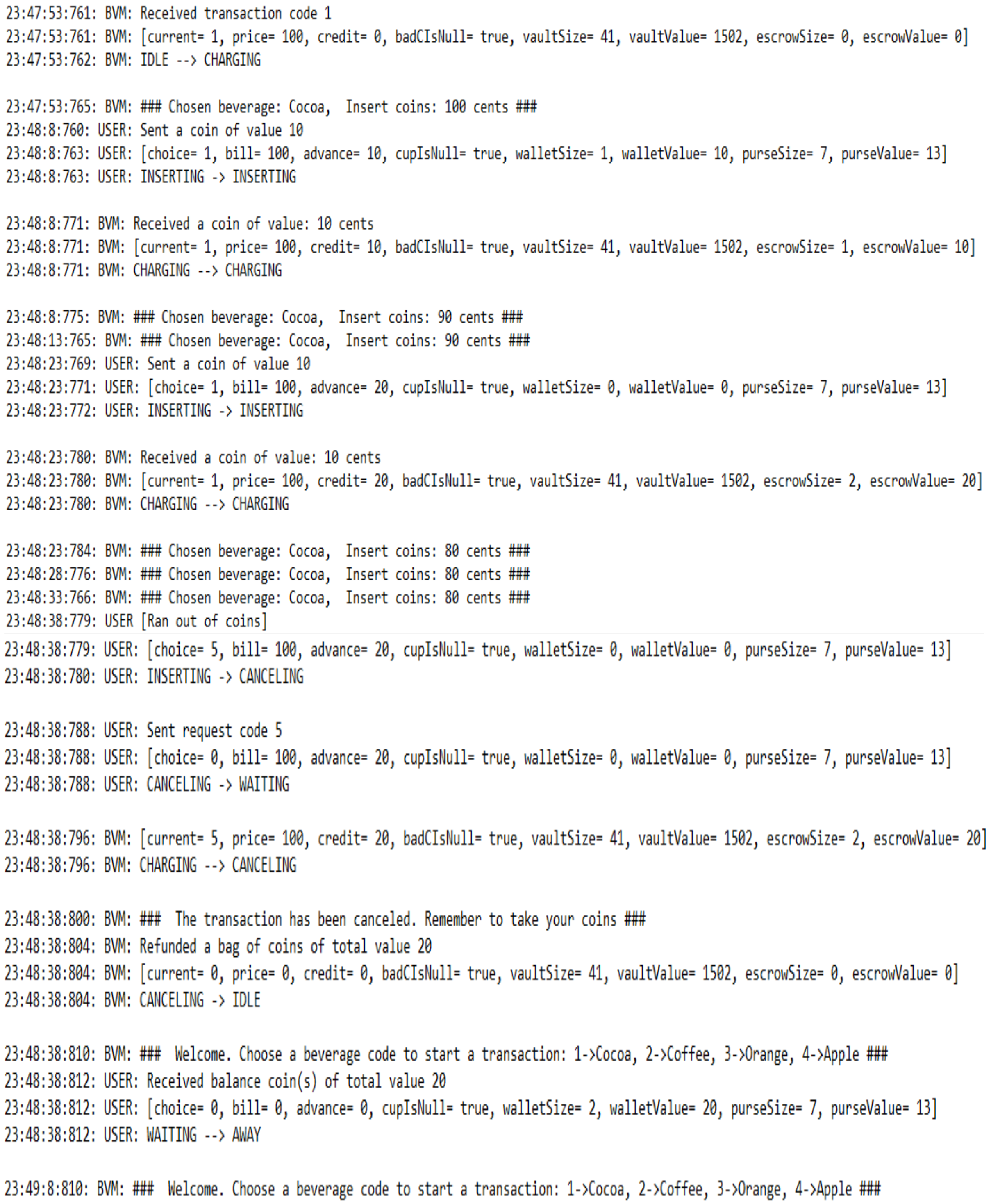
This exchange of coins continued until 23:24:23:229, when BVM received the 100-cent coin, which had been released by USER at 23:24:23:220. With this latest coin, *credit* became greater than required; leading to the transition 23:24:23:230:BVM:CHARGING-->RETURNING to refund the balance before assuming state DISPENSING at 23:24:23:242 with yet another activity while USER waited to get the order.

Finally, the cup of orange released by BVM at 23:25:8:251 was received by USER at 23:25:8:259 while BVM and USER closed the transaction with state transitions 23:25:8:252: BVM: DISPENSING -> IDLE and 23:25:8:260: USER: WAITING --> AWAY respectively.

Another case that might interest the reader is the system's behavior when USER runs out of coins in the midst of a transaction. This scenario is demonstrated in the last transaction recorded in these traces. The excerpt of the traces that contains this is presented in Figure 5.17.



```
23:44:53:742: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:44:53:742: USER: [choice= 0, bill= 0, advance= 0, cupIsNull= false, walletSize= 2, walletValue= 20, purseSize= 7, purseValue= 13]
23:44:53:743: USER: WAITING --> AWAY

23:45:23:744: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:45:53:745: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:46:23:745: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:46:53:746: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:47:23:747: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:47:53:747: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:47:53:748: USER: [choice= 1, bill= 0, advance= 0, cupIsNull= true, walletSize= 2, walletValue= 20, purseSize= 7, purseValue= 13]
23:47:53:748: USER: AWAY -> ORDERING

23:47:53:755: USER: Sent request code 1
23:47:53:755: USER: [choice= 1, bill= 100, advance= 0, cupIsNull= true, walletSize= 2, walletValue= 20, purseSize= 7, purseValue= 13]
23:47:53:755: USER: ORDERING -> INSERTING
```



```
23:47:53:761: BVM: Received transaction code 1
23:47:53:761: BVM: [current= 1, price= 100, credit= 0, badCIsNull= true, vaultSize= 41, vaultValue= 1502, escrowSize= 0, escrowValue= 0]
23:47:53:762: BVM: IDLE --> CHARGING

23:47:53:765: BVM: ### Chosen beverage: Cocoa,  Insert coins: 100 cents ###
23:48:8:760: USER: Sent a coin of value 10
23:48:8:763: USER: [choice= 1, bill= 100, advance= 10, cupIsNull= true, walletSize= 1, walletValue= 10, purseSize= 7, purseValue= 13]
23:48:8:763: USER: INSERTING -> INSERTING

23:48:8:771: BVM: Received a coin of value: 10 cents
23:48:8:771: BVM: [current= 1, price= 100, credit= 10, badCIsNull= true, vaultSize= 41, vaultValue= 1502, escrowSize= 1, escrowValue= 10]
23:48:8:771: BVM: CHARGING --> CHARGING

23:48:8:775: BVM: ### Chosen beverage: Cocoa,  Insert coins: 90 cents ###
23:48:13:765: BVM: ### Chosen beverage: Cocoa,  Insert coins: 90 cents ###
23:48:23:769: USER: Sent a coin of value 10
23:48:23:771: USER: [choice= 1, bill= 100, advance= 20, cupIsNull= true, walletSize= 0, walletValue= 0, purseSize= 7, purseValue= 13]
23:48:23:772: USER: INSERTING -> INSERTING

23:48:23:780: BVM: Received a coin of value: 10 cents
23:48:23:780: BVM: [current= 1, price= 100, credit= 20, badCIsNull= true, vaultSize= 41, vaultValue= 1502, escrowSize= 2, escrowValue= 20]
23:48:23:780: BVM: CHARGING --> CHARGING

23:48:23:784: BVM: ### Chosen beverage: Cocoa,  Insert coins: 80 cents ###
23:48:28:776: BVM: ### Chosen beverage: Cocoa,  Insert coins: 80 cents ###
23:48:33:766: BVM: ### Chosen beverage: Cocoa,  Insert coins: 80 cents ###
23:48:38:779: USER [Ran out of coins]
23:48:38:779: USER: [choice= 5, bill= 100, advance= 20, cupIsNull= true, walletSize= 0, walletValue= 0, purseSize= 7, purseValue= 13]
23:48:38:780: USER: INSERTING -> CANCELING

23:48:38:788: USER: Sent request code 5
23:48:38:788: USER: [choice= 0, bill= 100, advance= 20, cupIsNull= true, walletSize= 0, walletValue= 0, purseSize= 7, purseValue= 13]
23:48:38:788: USER: CANCELING -> WAITING

23:48:38:796: BVM: [current= 5, price= 100, credit= 20, badCIsNull= true, vaultSize= 41, vaultValue= 1502, escrowSize= 2, escrowValue= 20]
23:48:38:796: BVM: CHARGING --> CANCELING

23:48:38:800: BVM: ###  The transaction has been canceled. Remember to take your coins ###
23:48:38:804: BVM: Refunded a bag of coins of total value 20
23:48:38:804: BVM: [current= 0, price= 0, credit= 0, badCIsNull= true, vaultSize= 41, vaultValue= 1502, escrowSize= 0, escrowValue= 0]
23:48:38:804: BVM: CANCELING -> IDLE

23:48:38:810: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:48:38:812: USER: Received balance coin(s) of total value 20
23:48:38:812: USER: [choice= 0, bill= 0, advance= 0, cupIsNull= true, walletSize= 2, walletValue= 20, purseSize= 7, purseValue= 13]
23:48:38:812: USER: WAITING --> AWAY

23:49:8:810: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
```

Figure 5.17 Enactment traces of the BVS: Excerpt B

At 23:47:53:755, USER initiated the last transaction in the traces by sending a transaction code 1. This was followed by the routine exchange of coins between USER and BVM until the former suddenly ran out of coins at 23:48:38:779 (walletSize= 0 and walletValue= 0) after having expended coins worth 20 cents (advance= 20) on the transaction. This triggered the internal state transition recorded in 23:48:38:780: USER: INSERTING -> CANCELING, and consequently, the issuance of a request to cancel the transaction as recorded in trace 23:48:38:788: USER: Sent request code 5, followed by the internal state transition 23:48:38:788: USER: CANCELING -> WAITING to await the refund of the coins already expended.

BVM, which was expecting a coin towards completing the transaction, reacted to the transaction code received with the transition 23:48:38:796: BVM: CHARGING --> CANCELING; this was followed by a refund of the content of its *escrow* at 23:48:38:804 before transiting to the idle state as recorded in trace 23:48:38:804: BVM: CANCELING -> IDLE. USER received the refund at 23:48:38:812, kept it in its wallet (walletSize= 2, walletValue= 20) and walked away as indicated in trace 23:48:38:812: USER: WAITING --> AWAY without a cup of beverage (cupIsNull= true).

5.6 CONCLUSION

In this chapter, we have proposed a DEVS-based framework for the enactment of DESs. We described enactment, in the context of systems engineering, as a methodology for the execution of a software implementation of a system's behavior to verify its operational and functional characteristics in real clock time. Our intent is to complement the conventional DES simulation with a novel methodology that supports the runtime (occurring in real wall clock time) observation and analysis of the evolution of the state and input/output trajectories of a system. It also the runtime observation and analysis of the *activities* (state-preserving operations) executed during the system's sojourn in certain states, as well as the possibility of live interactions with the running system, either by human or by machine.

Relying upon the considered universality of DEVS to express DESs for simulation, we extended the atomic DEVS formalism with the concepts of *activity* to define the underlying formalism for our enactment framework. In contrast to the DEVS simulation algorithm, which uses virtual time to schedule states and state transitions, we use a variant of the Object-Oriented observer design pattern to orchestrate the state transition events in the enactment protocol we define for the framework. In the framework design, every system has a *clock*, which references the real clock of the machine on which the enactment process is being executed.

Using the observer pattern's terminology, a system is an *observer* of its clock and all its input ports (the clock and all ports are *subjects*) and a port can be an *observer* of the ports of other systems. Hence, the system is automatically *notified* of changes in the state of its clock and/or any of its input ports. Upon assuming a new state, a system uses its clock as a timer to monitor the time advance, which automatically notifies the system when the deadline expires. This notification triggers an internal state transition in the system. In addition, whenever an input is received, the input port involved automatically notifies the system thereby triggering an external state transition. A confluent state transition event is triggered when notifications from the clock and an input port occur concurrently. We realize couplings between the components of a coupled system model by making receiving ports observers of the corresponding sending ports so that the formers are automatically notified whenever there are changes of states in the latter.

We have done a Java implementation of the framework; the most essential parts of the implementation have been presented in the chapter and the remaining parts are documented in Appendix A.

To demonstrate the use of the framework, we presented a case study of the modeling and enactment of the beverage vending system. The traces obtained from running the enactment with a 64-bit Windows operating system running on a 2.40GHz processor and 8GB installed memory was also presented in the chapter. We observed from the traces that exchange of messages between components occur in a "maximum" delay of 10 milliseconds; we claim that this coupling performance is reasonable except may be in some "very time-critical" systems.

6 HiLLS' SYNTAX

6.1 INTRODUCTION

In Chapter 4, we proposed the SimStudio II framework to address the research problems in this thesis: the integration of MDSE theories and methodologies, based on simulation, formal methods and enactment, with the goal of harnessing the synergy of the diverse theories, tools and experiences for complementary, rather than competitive, analyses of DESs. At the kernel of the architecture of the proposed framework is HiLLS, which is meant to serve two major purposes: 1) to be the unified formalism at the front-end for creation and updating of system models and 2) to be the seam that integrates the three computational analysis methodologies considered in the thesis. By virtue of its position and roles in the framework's architecture, HiLL provides a comprehensive answer to the research questions RQ2 (*which formalism should we adopt to write the shared unified model?*) and RQ3 (*how can the disparate concerns of the different methodologies be captured in the so-called unified model?*).Some of our research efforts and results on HiLLS have been reported in [MAT15, AMT16, AT16]; in this chapter we build on the preliminary results to present the current state of HiLLS's abstract and concrete syntaxes.

Recall from Section 3.5 that a language specification may consists of an abstract syntax, one or more concrete syntaxes and one or more semantics. The abstract syntax precisely defines the concepts expressed in the language and the legal relationships between them while a concrete syntax describes a set of human-comprehensible notations that physically, and unambiguously, render the entities and relationships specified in the abstract syntax.

The HiLLS' abstract syntax has been built from a disciplined integration of system-theoretic and software engineering concepts to capture, in a considerably generic form, the different concerns of simulation, formal analysis and enactment methodologies for DES in a coherence whole; this will be presented in details in Section 6.2. Section 6.3presents the HiLLS' concrete syntax, which contains, in addition to the notations specifically defined for HiLLS, variants of some notations adopted from Z schema and the UML family languages. We will demonstrate system specification with the language by presenting, in Section 6.4, the HiLLS model of the running example in this thesis - the Beverage Vending System (BVS) - and its required properties before concluding the chapter in Section6.4.4

6.2 HiLLS' ABSTRACT SYNTAX

In order to be expressive enough to actualize the visions of SimStudio II, we have derived the HiLLS’ abstract syntax from a disciplined integration of system concepts from disparate, but related sources to express coherently unified models that can serve the purposes of the three target platforms of the framework. Figure 6.1 presents a revised version of a diagram we reported in [AT16], which informally describes the steps taken to determine the concepts and relations to be specified in the HiLLS' abstract syntax.

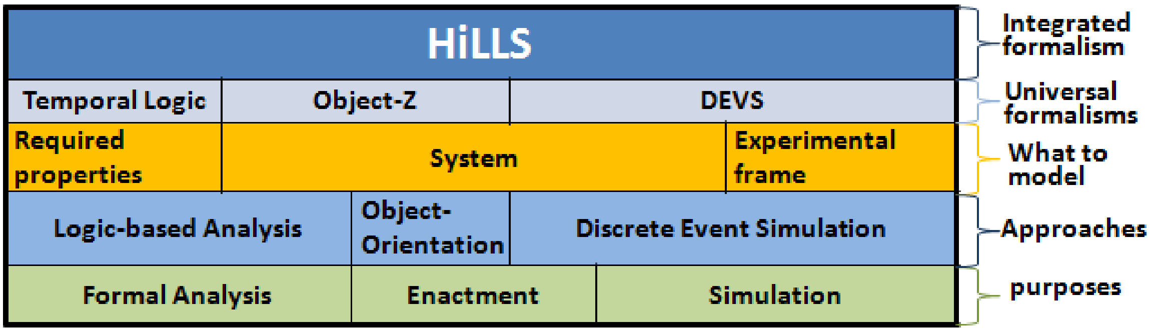


Figure 6.1 Build-up to HiLLS' abstract syntax

Starting with the purposes of modeling in the SimStudio II framework at the bottom row, we identified the conventional approaches (in the "Approaches" row) to study DESs for the corresponding purposes. We realized at this stage that an enactment process for DES could be established through a blend of discrete event simulation and object-orientation approaches to design a software prototype of a system. We then identified, in the middle row, the computational models that may be required to realize the different purposes using the corresponding approaches. While the model of the system under study may be sufficient for an enactment process, simulation and formal analysis methodologies often require, in addition to the modeling of the system under study, models of the experimental frame and required properties respectively. Next, we identified universal and highly expressive formalisms often employed for the different approaches.

Vangheluwe [Van00] has shown that DEVS is considerably universal for modeling most kinds of DESs, as well as provide approximated models of non-DESs, for simulation; it may also be used to model the experimental frame for a DEVS-based simulation process; hence we relied on DEVS for a comprehensive study of the concepts required for simulation processes. Object-Z is an object-orient variant of Z, which is extensively used by Formal Methods practitioners to specify state-based systems for logical analysis so much so that it has been recognized by the International Organization for Standardization. Thus, we considered to study the system constructs expressed in Object-Z to take benefit of the universality of its base formalism, Z, and the Object-Oriented structuring of system specifications. Moreover, it offered the opportunity of exploring the combination of object-orientation with discrete event system concepts to serve the purpose of enactment. Finally, we have chosen the Temporal Logics (TL) to study the concepts for specifying the required system properties for its rigor and succinctness in expressing temporal properties and its usability in combination with system specification formalisms such as Z for requirement verifications.

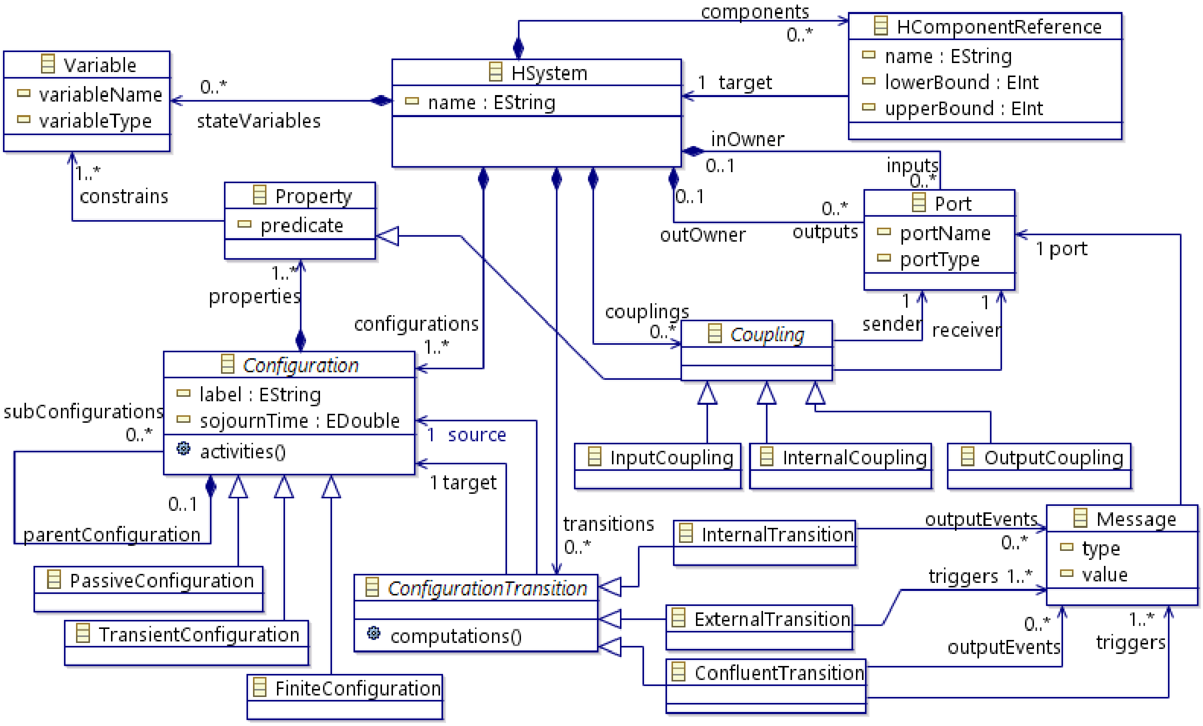
After thorough examinations and comparisons of the significances of the different concepts expressed in DEVS and Object-Z in their respective paradigms, we realized that the purposes served by their common concepts are sufficiently related to warrant the specification of common and high level representations for them. For instance, DEVS' state set and Object-Z's state schema describe the same aspect of a system, though at different levels of refinement and for different purposes; similarly, we can find a common representation for the state changing operations in Object-Z and DEVS' state transition functions. Another interesting discovery we made is that TL formulas can be expressed in the form of a state-transition system with similar representations as the behavior of the system itself. This discovery gave further credence to the prospect of a unified model from which the artifacts for the disparate methodologies can be systematically derived. Thus, we have built the HiLLS' abstract syntax from a systematic integration, using metamodel integration techniques presented in Section 3.3.3.4, of system-theoretic concepts from DEVS and of software engineering concepts from Object-Z and TL.

In the rest of this section, we will first present overviews of the metamodels that describe the concepts and relations we adopt from the different source formalisms, this will be followed by the integration of the different concepts for incremental definition of the HiLLS' syntax.

6.2.1 System-Theoretic Concepts Adopted from DEVS

Figure 6.2 presents a simple metamodel of the DES concepts adopted from DEVS plus a few other concepts to kick-start the incremental definition of the HiLLS' abstract syntax. As shown in Figure 6.2(a), class *HSystem* describes a DES. *HSystem* may have zero or more state variables (*stateVariables*), input ports (*inputs*), output ports (*outputs*),and references to components (*components*), each of which is *HSystem* (*target*).A composite system defines *couplings* between its components to facilitate the exchange of messages between them. A coupling belongs to one of three kinds: *InputCoupling*, *InternalCoupling* and *ExternalCoupling,* which are equivalent to DEVS's EIC, IC and EOC respectively. More details on the properties and distinguishing features of each of the three couplings are provided in the OCL (Object Constraint Language) constraints specified in Figure 6.2(b).

The behavior of a *HSystem* is described by *configurations* and *transitions* between configurations. A configuration is a labeled cluster of states that satisfy some unique properties defined on the state variables. i.e., configurations are disjointed subsets of the state space; the elements of each subset are states that satisfy a unique property defined by some logical constraints on the state variables.



(a) DEVS-based discrete event system concepts

```
context Port
    inv owner_Constraint ('A port is used for either input or output and not both'):
        inOwner->isEmpty() xor outOwner->isEmpty()
    def:
        owner:System = if (inOwner->isEmpty()) then outOwner else inOwner endif

context Coupling
        inv No_feedback_coupling ('Coupling ports of same system is illegal'):
            sender.owner <> receiver.owner

context InputCoupling
    inv EIC_Constraints ('sender = input of container, receiver = output of a component'):
        sender.owner.components.target-> includes(receiver.owner) and
        sender.owner.inputs -> includes(sender) and --sender is an input port of its owner
        receiver.owner.inputs -> includes(receiver) --receiver is an input port of its owner

context OutputCoupling
    inv EOC_Constraints ('sender = input of a component, receiver = output of container'):
        receiver.owner.components.target->includes(sender.owner) and
        sender.owner.outputs -> includes(sender) and --sender is an output port of its owner
        receiver.owner.outputs -> includes(receiver)--receiver is an output port of its owner

context InternalCoupling
    inv IC_Constraints ('sender = output of a component, receiver = input of a component'):
        sender.owner.outputs -> includes(sender) and
        receiver.owner.inputs -> includes(receiver)
```

(b) Static constraints

Figure 6.2Metamodel of system-theoretic concepts adopted from DEVS

A typical example of the use of *configurations* is the variable ϕ in our DEVS specification of the BVS in Section 3.2.3; each of the values of ϕ defines a configuration. In addition to unique property, a configuration is characterized by a unique *label* and a *sojournTime* (same as *timeAdvance* in DEVS) that defines the maximum possible length of the system's sojourn in the state. It may also define some *activities*, which are operations that are executed whenever the system is in the configuration but which do not lead to a change in the value of any state variable, neither do they involve input and/or output operations. In a similar manner as in DEVS states, a configuration can be classified into one of three kinds based on the value of its sojourn time: *TransientConfiguration* (*sojournTime* = 0), *PassiveConfiguration* (*sojournTime* = positive infinity) and *FiniteConfiguration* (0 <*sojournTime*< positive infinity).

Finally, *HSystem* may define zero or more configuration transitions that specify the system's behavior in terms of the evolution of the configurations. Like in DEVS, a *ConfigurationTransition* can be one of three kinds: *InternalTransition*, *ExternalTransition* and *ConfluentTransition* all of which are the effects of *computations* that result in the re-configuration of the state variables, and consequently, leading to the satisfaction of the *target* configuration. i.e., to transit from the *source* configuration to the *target* configuration, the *computations* of the *transition* must be executed. This execution results in the modification of the values of some state variables, thereby making the new values satisfy the constraints of the target configuration. An *InternalTransition* occurs when the *sojournTime* of the *source* configuration expires and it may be preceded by some output events. An *ExternalTransition* occurs when *input event(s)* is (are) received on at least one of the *input ports* before the expiration of the *timeAdvance* of the *source* configuration. A *ConfluentTransition* occurs when *input event(s)* is (are) received at the expiration of the *sojournTime* of the *source* configuration and may be preceded by some output events. Every event (message) is associated with a transition and has a reference to a port; a received message has a reference to the input port on which it was received while an output message generated within the system has a reference to an output port on which it will be sent.

6.2.2 Software Engineering Concepts Adopted from Object-Z

The author of Object-Z provides a grammar-based specification of its syntax in [Smi12]; the detailed presentation of all elements of the grammar is beyond the scope of this thesis. We have derived a simplified metamodel of the concepts that are most relevant to our work for integration with concepts of other formalisms in a modelware technological space. The derived metamodel is presented in Figure 6.3.An Object-Z specification consists of *paragraphs*, each of which is an *OZClass* or a *FreeType* definition.

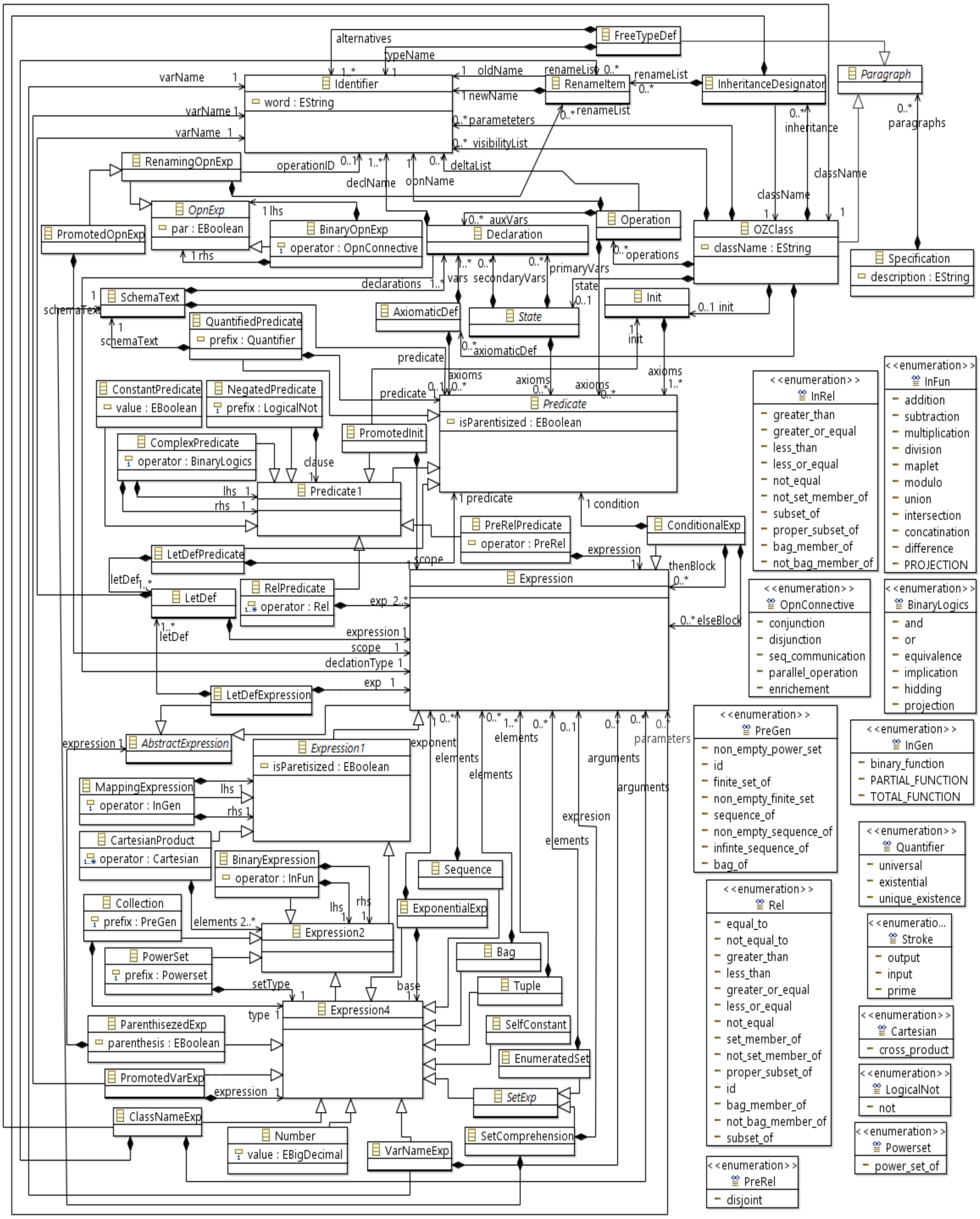


Figure 6.3 Simplified Object-Z metamodel

We presented, in Section 3.2.5, an abstract template of the structure of an Object-Z *class schema* in Figure 3.13; let us use this template to explain the metamodel in Figure 6.3 for the sake of clarity and brevity. Class *OZClass* in the metamodel describes the Object-Z class schema. *OZClass* has an attribute *className*, which represents the class' identifier, zero or more *parameters*, an optional *visibilityList*, an optional list of *InheritanceDesignators*, zero or one state schema, *Sate*, which declares primary and/or secondary state variables (whose types are defined by expressions) and possibly a list of *predicates* specifying constraints on the declared variables. It also has an optional axiomatic definition, *AxiomaticDef*, which may define constants and global variables, an optional *Init* that specifies the predicates defining the initial state of the class at creation, and finally, zero or more *operations* that use and/or manipulate the variables declared in the state schema.

Object-Z specifies different kinds of *predicates* for writing many kinds of logical expressions, which may be true or false. While simple predicates use *relational operators* such as described in enumerations *Rel* and *InRel* to define logical expressions, *complex predicates* use *logical connectives* such as enumerations *BinaryLogics* and *LogicalNot* for hierarchical composition of simpler predicates. The remaining parts of the metamodel give further refinements of the different elements of the Object-Z class schema.

6.2.3 Metamodel of TL Property Patterns

We discussed, in Section 3.2.6.3, the patterns of commonly checked temporal properties in system as compiled by Dwyer et al. and used as a guide for the specification of complex temporal properties in system requirement specifications. We present a metamodel of the property patterns in Figure 6.4 to facilitate our attempt to establish the relationships between the elements of the patterns and the system concepts in DEVS and/or Object-Z and unify them in the abstract syntax of HiLLS.

As described in Figure 6.4, a *RequirementSpecification* consists of *temporalProperties* each of which conforms to a *PropertyPattern* and a *Scope*. There are five kinds of *Scope*: *Before, After, Global, AfterUntil and Between.* A *Before* (resp. *After*) scope refers to a proposition (*delimiter*), which when it becomes true marks the "end" (resp. "beginning") of the segment of execution within which the *pattern* of a *TLProperty* must be satisfied. A *Between* scope is a cascade of *After* and *Before* scopes; it refers to two propositions - *startDelimiter* and *endDelimiter* - between which the specified pattern must be satisfied. The *AfterUntil* scope is similar to *Between* except that the occurrence of the truth of the *endDelimiter* is not guaranteed in the former. A *Global* scope implies that the specified pattern must be satisfied throughout the execution.

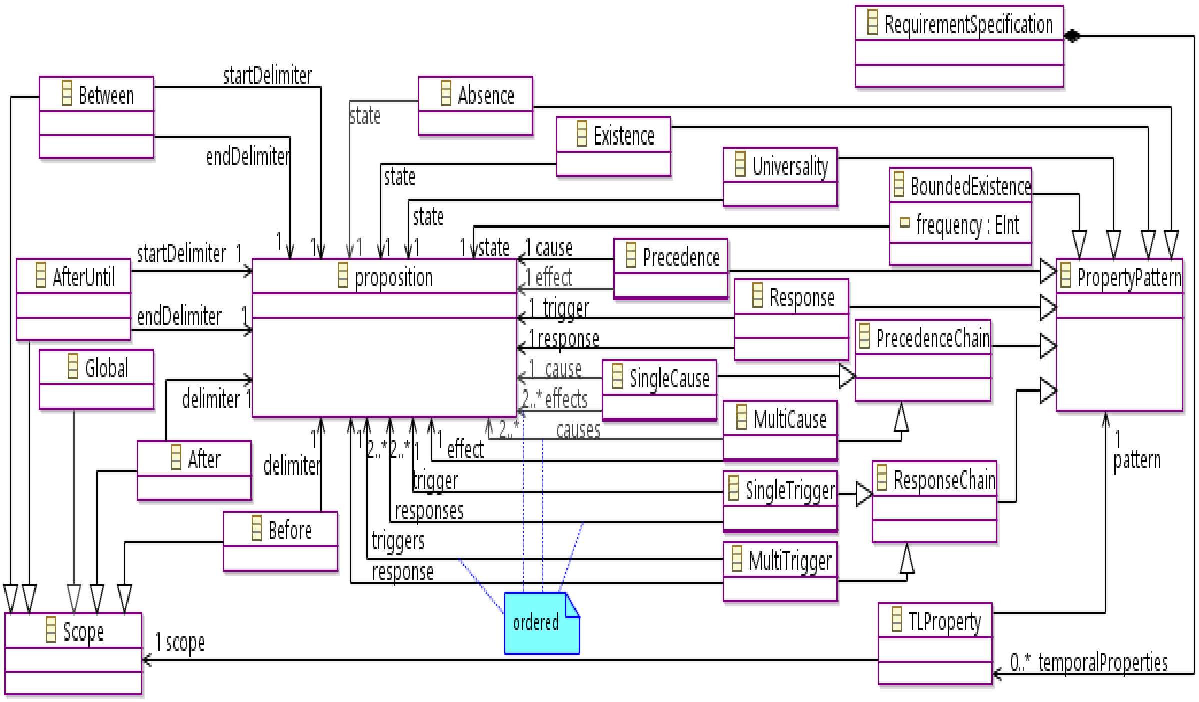


Figure 6.4 Metamodel of Dwyer's TL property patterns

Similarly, a *TLPattern* can be of the kind *absence, existence, universality, bounded existence, precedence, response, precedence chain (with single cause and multiple ordered effects or with multiple ordered causes and single effect),* and *response chain (with single trigger and multiple ordered responses* or *with multiple ordered triggers and single response)* as described in Section 3.2.6.3.

6.2.4 Derivation of the HiLLS' Metamodel

The HiLLS metamodel has been built incrementally by successive applications of some of the metamodel composition techniques described in Section 3.3.3.4 for the integration of concepts from the different metamodels presented earlier. This sub-section presents the major steps taken to arrive at the final metamodel. We will start with the integration of the concepts to model system, and then follow it up with the integration of the concepts to model requirements.

6.2.4.1 Integration of System Modeling Concepts in HiLLS Metamodel

As a reminder, we intend to integrate the system-theoretic concepts in Figure 6.2 and some concepts in the Object-Z metamodel (Figure 6.3) in order to exploit the strength of one to complement the other; it would be interesting if concepts such as *stateSchema* and *operations* in the latter are shared with the former. For instance, the state schema offers a more precise and verifiable way to specify the state space compared to the abstract description of class *Variable* in

Figure 6.2. Moreover, incorporating the *operation* schema and *axiomatic definition* can further enrich the language with constructs to specify some system-specific operations and global constants or variables. One quick solution to come to mind would be to create an inheritance from *HSystem* to *OZClass*; this is, however, not a panacea in this case because the latter has many other elements that are not required in the former.

We recall that metamodel interfacing (see Section 3.3.3.4) suggests the introduction of new classes and relations to combine two metamodels describing distinct but related domains in order to explore the relationships between them. Using this technique, we introduce an interface class, *HClassifier*, to kick-start the integration of the DEVS-based and Object-Z concepts as shown in Figure 6.5. Consequently, *HSystem* and *OZClass* become kinds of *HClassifier* but no relations have been established between their components.

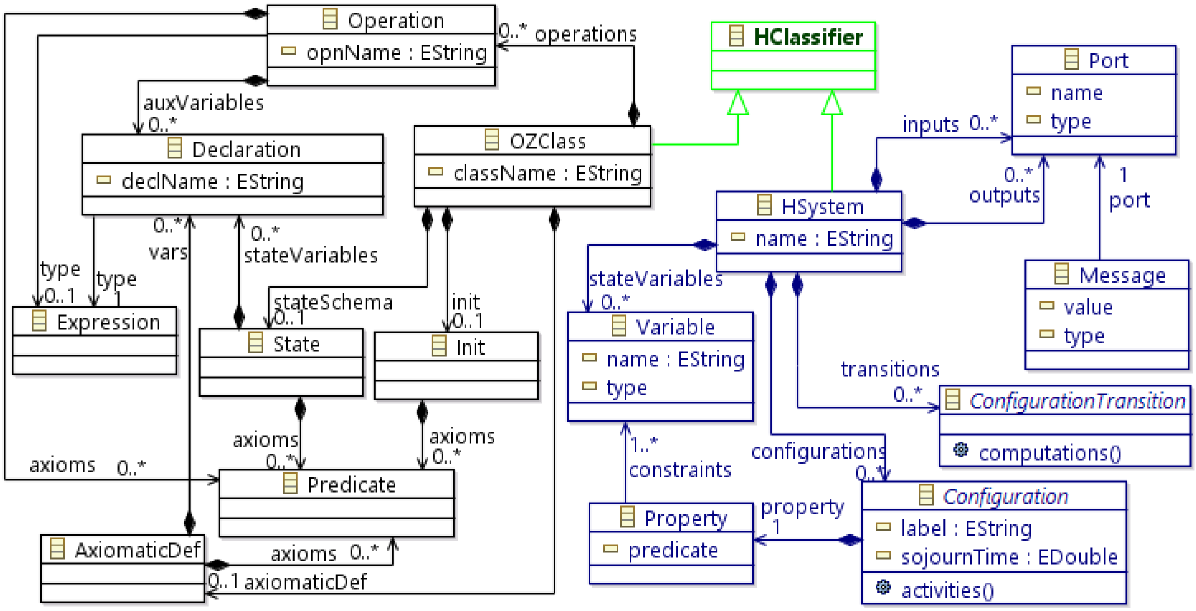


Figure 6.5 Interfacing the DEVS-based and Object-Z concepts in HiLLS metamodel

We restructure the metamodel by filtering the *OZClass* concepts required by *HSystem* and associate them to the interface class (*HClassifier*) as shown in Figure 6.6; *OZClass* has now been renamed to *HClass* for uniformity of nomenclature. Hence, *HClass* can preserve its associations with other elements as in the original metamodel while maintaining relations with the shared elements via *HClassifier*. At this point, the set of variables *stateVariables* directly owned by *HSystem* becomes redundant since a more rigorous mechanism (i.e., *stateSchema*) to specify the state space has been inherited from *HClassifier*. Thus, class *Variable* is marked for deletion to remove the redundancy.

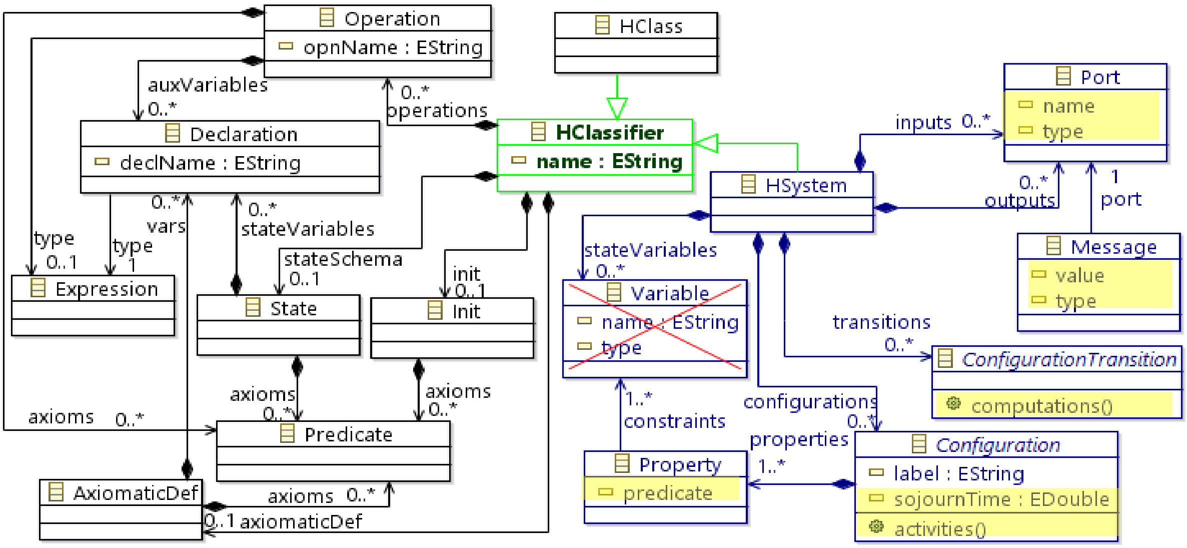


Figure 6.6Reorganization of concepts and marking of abstract concepts for refinement

Almost all components of *HSystem* require more details to clarify their true natures. For instance, the attribute *predicate* of *Property* is actually a constraint on the state variables; it is, however, not clear, how the predicate is specified. Similarly, class *Port* (resp. *Message*) is defined as having a *name* (resp. *value*) and *type*, which require some clarifications. Each of the operations *computations* and *activities* of *ConfigurationTransition* and *Configuration* respectively is conceptually an ordered set of expressions; while the former is executed during a transition, the latter is executed during the sojourn of the system in the corresponding configuration. All these details can be obtained freely from the components of *HClassifier* using the class refinement technique (described in Section 3.3.3.4), which involves the reuse a fragment of a metamodel to provide a detailed specification of a considerably abstract concept in another metamodel.

Figure 6.7 presents the resulting metamodel after the application of the class refinement technique to the metamodel in Figure 6.6. Class*Property* refined by its new reference to *Predicate*, which offers a wide range of constructs to specify logical predicates (only a few are shown in this excerpt). Class *Configuration* is refined by its two relations *soujournTime* and *activates* with the class *Expression*. While the former allows the language's user to use the different kinds of expression to precisely specify how the *sojournTime* of a configuration is computed at runtime, the latter provides the means to clearly specify an ordered set of expression as the activity of a configuration. Similarly, the reference *computations* of class *ConfigurationTransition*allows for the precise specification of the ordered set of expressions, which when executed will lead to the corresponding transition. Reference *value* of class *Message* defines the expression that yields the actual message event. Finally, class *Port* refers to a declaration (*portDecl*) that precisely specifies a port's name and its type of admissible objects.

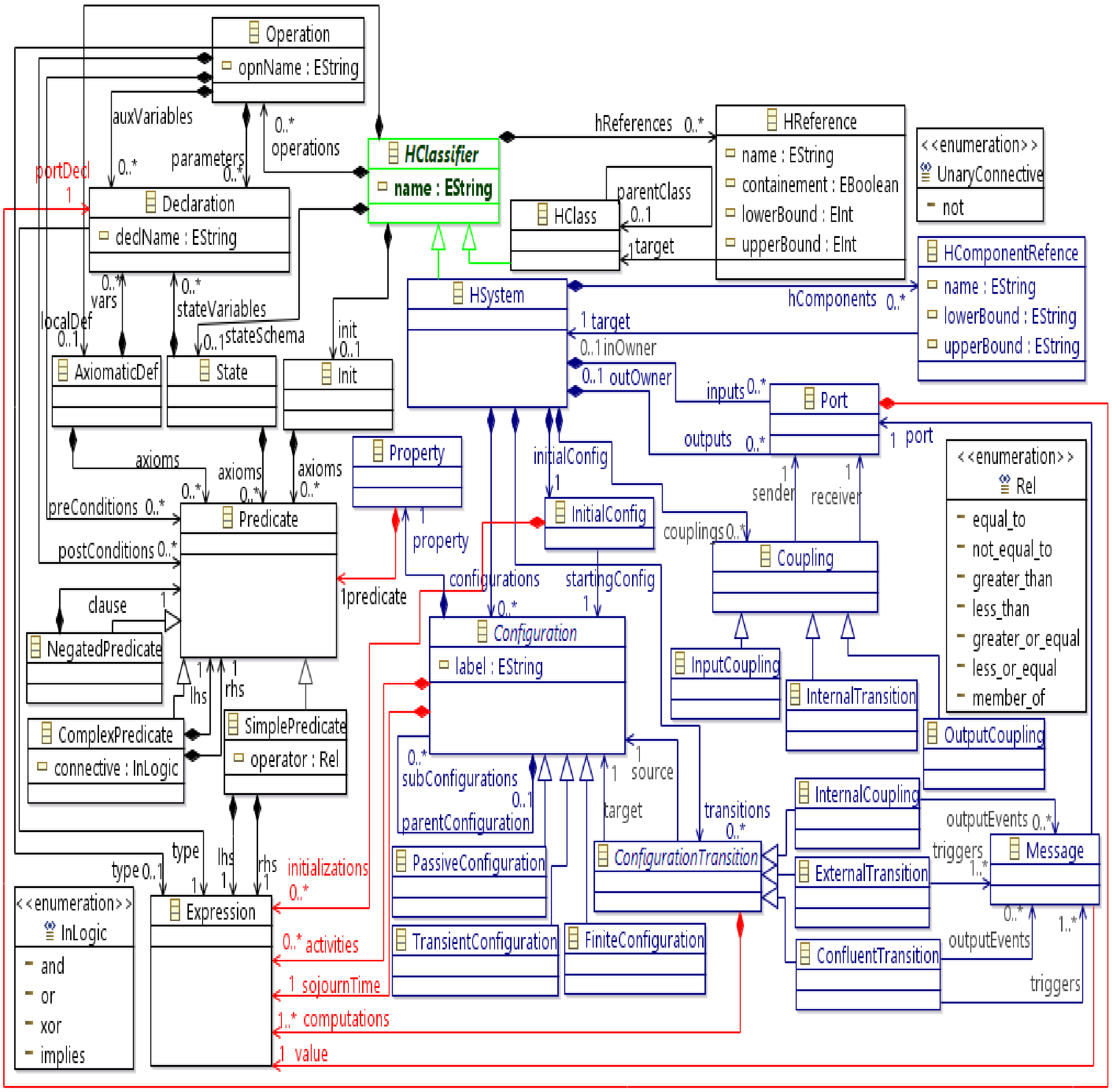


Figure 6.7Refinement of essential system modeling concepts in HiLLS metamodel

6.2.4.2 Integration of System and Requirement Modeling Concepts in HiLLS Metamodel

Figure 6.8 presents the integration of the system metamodel (Figure 6.7) and the metamodel of property patterns (see Figure 6.4).Using metamodel interfacing; we introduce an interface class *HiLLSSpecification* with references to *HClassifier* and *HRequirementSpecification*. Therefore, an hclassifier satisfies zero or more requirement specifications. Conceptually, the class *Proposition* in the property metamodel is a statement on the system's state, which may be true or false; we refine the class to provide this detail via its new reference to class Predicate.

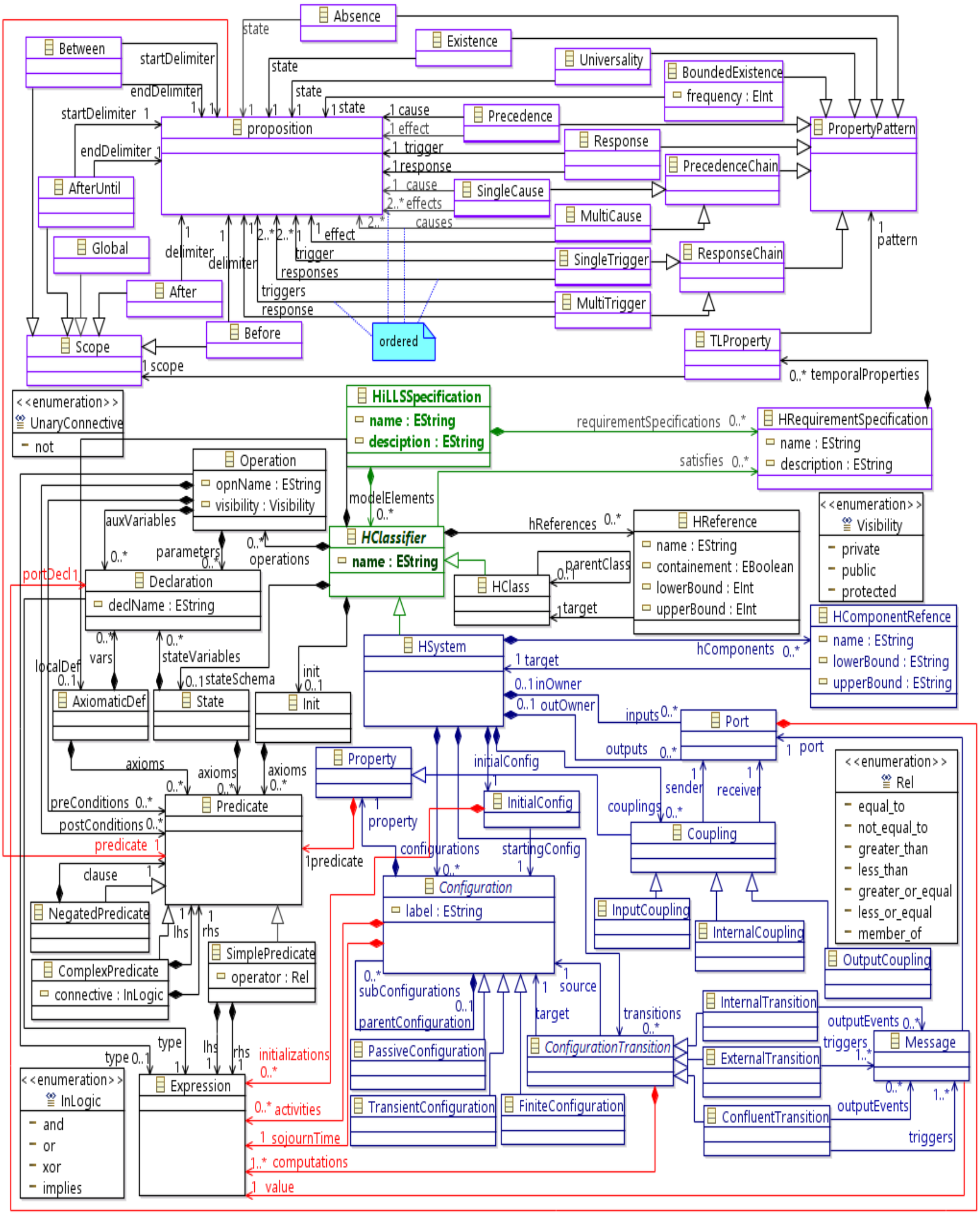


Figure 6.8 Essential elements in HiLLS metamodel

The OCL code in Figure 6.9 specifies some static constraints on the metamodel in Figure 6.8 to disambiguate some of its elements, especially the configurations, configuration transitions and port couplings specified in a HSystem.

```
import 'HiLLS.ecore'
package hills
context HiLLSSpecification
  inv unique_HClassifier_name ('HSystems and HClasses must have unique names'):
    modelElements->forAll(ent1:HClassifier, ent2:HClassifier| ent1 <> ent2 implies
        ent1.name <> ent2.name)
context HSystem
  inv unique_configurations ('configurations must have unique names and properties'):
    configurations->forAll(config1:Configuration, config2:Configuration|config1<>config2
        implies
    config1.label <> config2.label and config1.properties<>config2.properties)
context Configuration
  inv isolated_configuration_constraint ('Isolated configuration is illegal'):
    HSystem.transitions->exists(source = self xor target = self)
  inv nonPassive_Configurations_sonstraints ('non-passive config cannot be a final
      state'):
    not self.oclIsTypeOf(PassiveConfiguration) implies System.transitions->exists(source
        = self)
context ConfigurationTransition
  inv PassiveConfig_InternalTrans_Constraint ('Internal and confluent transitions cannot
      originate from a passive cofiguration'):
    self.oclIsTypeOf(InternalTransition) or self.oclIsTypeOf(ConfluentTransition)
    implies not self.source.oclIsTypeOf(PassiveConfiguration)
context Port
  def: owner:System = if (inOwner->isEmpty()) then outOwner else inOwner endif
context Coupling
    inv No_feedback_coupling ('Coupling ports of same system is illegal'):
      sender.owner <> receiver.owner
context InputCoupling
  inv EIC_Constraints ('sender = input of container, receiver = output of a component'):
    sender.owner.hComponents.target-> includes(receiver.owner) and
    sender.owner.inputs -> includes(sender) and --sender is an input port
    receiver.owner.inputs -> includes(receiver) --receiver is an input port
context OutputCoupling
  inv EOC_Constraints ('sender = input of a component, receiver = output of container'):
    receiver.owner.hComponents.target->includes(sender.owner) and --sender is a
        component of receiver
      sender.owner.outputs -> includes(sender) and --sender is an output port
    receiver.owner.outputs -> includes(receiver)--receiver is an output port
context InternalCoupling
  inv IC_Constraints ('sender = output of a component, receiver = input of a component'):
    sender.owner.outputs -> includes(sender) and --sender is an output port
    receiver.owner.inputs -> includes(receiver) --receiver is an input port
 endpackage
```

Figure 6.9 Static constraints on the HiLLS metamodel

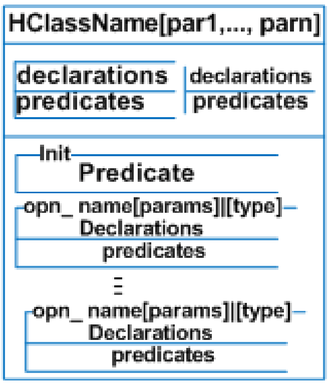
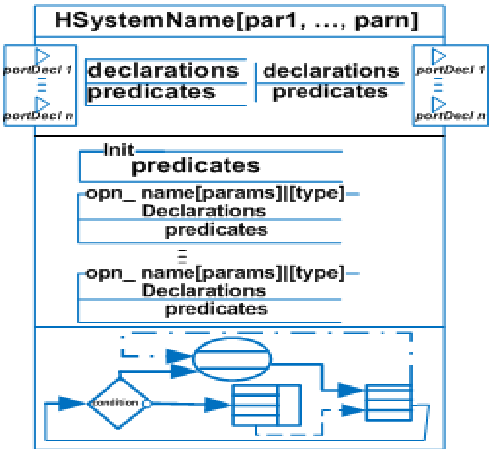
6.3 HiLLS' CONCRETE SYNTAX

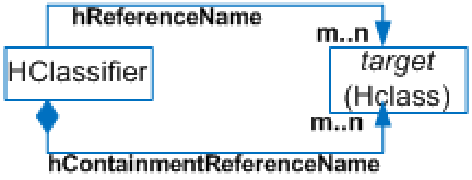
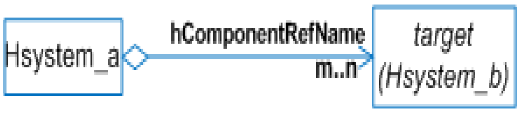
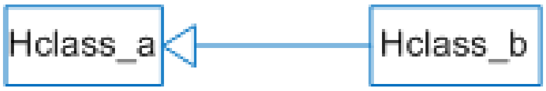
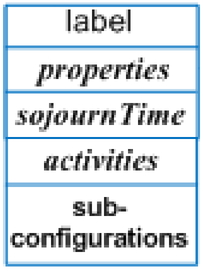
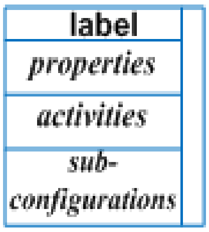
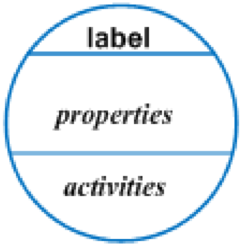
This section presents the concrete notations for expressing the concepts and relationships described in the HiLLLS' abstract syntax. In order to take benefit of its universality and ease of comprehension, we adopt and extend some notations from the UML family of languages and some notations from Object-Z to define the graphico-textual notations for HiLLS' concepts. This section is divided into two parts; we will present the notations for describing systems in the first part, and follow it with the notations for high-level modeling of temporal requirements in the second part.

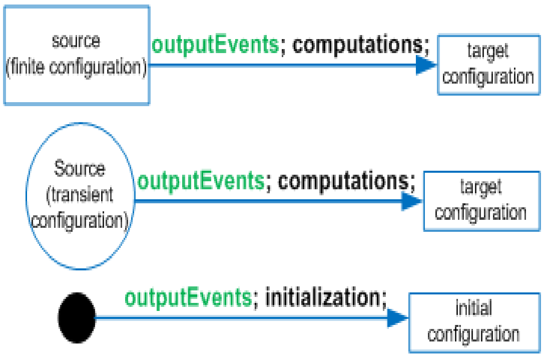
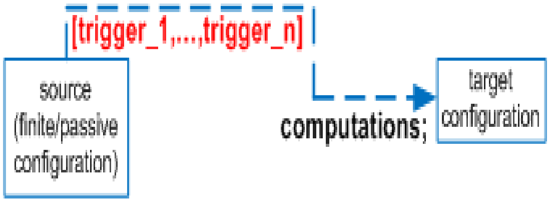
6.3.1 Concrete Notations for System Specification

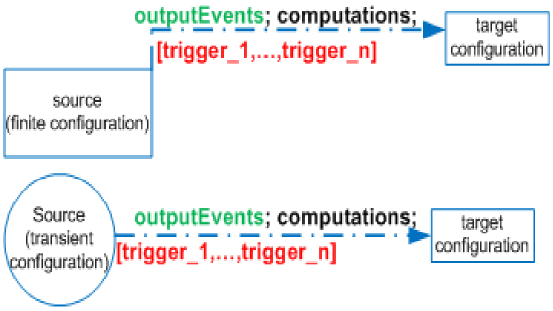
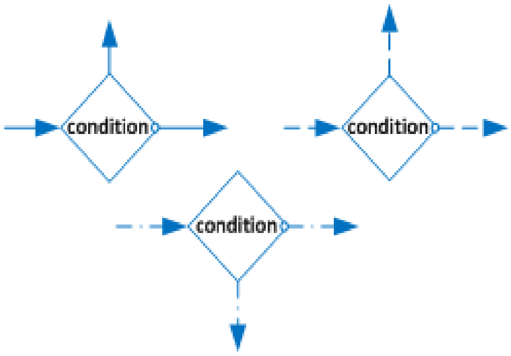
Table 6.1 presents the HiLLS notations for expressing system models.

Table 6.1 HiLLS' notations for system description

Notations	Descriptions
 HClassName[par1,..., parn] declarations predicates declarations predicates Init Predicate opn_ name[params]][type] Declarations predicates opn_ name[params]][type] Declarations predicates *HClass*	*HClass* is a box with three compartments similar to the UML class symbol. The first compartment contains the HClass' name and parameters if any. The second compartment houses the state and axiomatic schema if any; we adopt the notations of the state schema and axiomatic definition as used in Object-Z. A declaration is written in the format ***declarationName*:*Type*.** The third compartment houses the class' operations if any. An operation is similar to the state schema but with additional information indicated on its top side. The top bears the name attribute of the operation, the type and list of parameters (if any) of the operation in brackets.
 HSystemName[par1, ..., parn] portDecl 1 portDecl n declarations predicates declarations predicates portDecl 1 portDecl n Init predicates opn_ name[params]][type] Declarations predicates opn_ name[params]][type] Declarations predicates condition *HSystem*	*HSystem*'s notation extends that of HClass; it has four compartments with the first three serving similar functions as in HClass while the fourth contains the configuration transition diagram that describes the system's behavior. The input and output interfaces are denoted by windows attached to the left and right sides respectively of the second compartment. In each rectangular window, we express a port as a small arrowhead labeled with the port's declaration. The format of a port declaration is ***portName*:*Type***.

Notations	Descriptions
 hReferenceName m..n HClassifier target (Hclass) m..n hContainmentReferenceName *HReference*	*HReference* and *HComponentReference* use similar notations as reference notations in UML; they are labeled arrows from an *HClassifier* to a *target* entity. While the target of the former is an *HClass*, the latter has a *HSystem* as target. In all cases, *m..n* denotes the cardinality with m and n representing the *lowerBound* and *upperbound* attributes respectively. The relation *parentClass* between any two HClasses is represented exactly the same way as the generalization concept in UML class diagram.
 Hsystem_a hComponentRefName m..n target (Hsystem_b) *HComponentReference*	
 *parentClass*	
 label properties sojournTime activities sub- configurations *FiniteConfiguration*	*FiniteConfiguration* is denoted by a box with five compartments for its *label*, *properties* predicate, *sojournTime* expression, *activities* expression(s) and sub-configurations respectively from top to bottom.
 *PassiveConfiguration*	*PassiveConfiguration* is similar to finite configuration except that the compartment for sojournTime is not represented; a vertical stripe is attached to its right side as an indication of its infinite *sojournTime.*
 *TransientConfiguration*	*TransientConfiguration* is denoted by a circle with three compartments for its *label*, *properties* predicate and *activities* expressions if any. Its shape naturally depicts its zero *sojournTime*.

Notations	Descriptions
 source (finite configuration) outputEvents; computations; target configuration Source (transient configuration) outputEvents; computations; target configuration outputEvents; initialization; initial configuration ***InternalTransition***	*InternalTransition* is a labeled solid arrow from source configuration to target configuration. The *outputEvents* (if any) and the ordered *computations* expressions constitute the label of the transition in the order presented in the figure. An *outputEvent* is specified in the format ***outputPortName!expression***, indicating the name of the associated output port and the expression that specifies the value to be sent; the exclamation mark "!" is an indication that ***expression*** is produced as output on port ***outputPortName***. An internal transition may emanate from the right of a finite or transient source configuration and terminate on the left side of the *target*, which can be any kind of configuration. It may also emanate from an initial configuration notation and terminate on the actual *sartingConfiguration* of the system.
 [trigger_1,...,trigger_n] source (finite/passive configuration) computations; target configuration ***ExternalTransition***	An *ExternalTransition* is a labeled dashed arrow from the *source* configuration, which may be a finite or passive configuration, to the *target* configuration, which may be any kind of configuration. The transition arrow also terminates on the left side of the target; it, however, emanates from either the top or the bottom side of the source configuration. The *triggers* are specified in a comma-separated list within a bracket at the source end of the transition arrow. A *trigger* specification is in the format ***inputPortName?expression,*** which indicates that the value of ***expression*** is received on input port ***inputPortName***.

Notations	Descriptions
 outputEvents; computations; [trigger_1,...,trigger_n] target configuration source (finite configuration) Source (transient configuration) outputEvents; computations; [trigger_1,...,trigger_n] target configuration ***ConfluentTransition***	A*ConfluentTransition* is a labeled dotted-dashed arrow from the *source* configuration, which may be a finite or transient configuration, to the *target* configuration, which may be any kind of configuration. While the transition arrow terminates on the left side of the target like in the previous cases, it originates vertically from the right edge (either bottom or top) of a finite source configuration as an indication that the trigger(s) is (are) received at the end of the sojourn time of the source.
 ***Conditional transition paths***	In cases where the path taken by a transition depends on a condition, we use the diamond symbol to disambiguate the flow of the computations in the conditional expression. The condition is, conceptually, a graph node with one *inflow* and two *outflows*; the transition flows into the diamond from any of its four edges; the outflow from an edge with a small circle indicate the path taken when *condition* is *true* and the other outflow is the path taken when *condition* is false. In some of our previous publications [MAT15, AMT16], the rule has been that the inflow is strictly on the left edge of the diamond, the outflow the truth of *condition* is strictly on the right edge and the other outflow can be on either the top or bottom edge. From our experience in using the language, we have relaxed this rule for greater flexibility since the inflow is distinguishable from the two outflows with the direction of the arrow and the small circle at the base of the truth path is sufficient to differentiate it from the other irrespective of the diamond edge.

6.3.2 Concrete Notations for Requirement Specification

This sub-section presents the proposed graphical notations for expressing the temporal properties patterns described in the HiLLS abstract syntax. In order to simplify communications of requirement specifications among the different stakeholders, we propose to use variants of the elements of HiLLS' transition diagram for expressing temporal properties in HiLLS. We believe that uniformity of notations in both system and requirement models will aid the user's specification and understanding of required temporal properties for DES. Similar benefits have motivated Meyers et al. [MWV+13, MDL+14] to propose a framework to support the use of domain-specific notations for specifying properties in DSLs.

Firstly, we introduce the notation of a requirement specification and its relation with the*HClassifier*(s) that satisfy the properties specified in it. A requirement specification is represented as a rectangular box with two compartments (see right of Figure 6.10) for the *requirementName* attribute and the temporal properties. The relation *satisfies* between an HClassifier and a *RequirmentSpecification*is denoted by a generalization arrow with dashed tail similar to the UML symbol for interface implementation/realization.

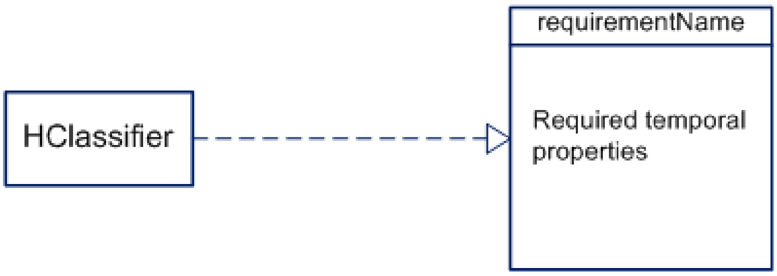


Figure 6.10Requirement notation

To express the temporal properties, we propose two notations, *generic configuration notation* and *hypothetical configuration notation* to express different aspects of a property pattern. A generic configuration notation, described on the left of Figure 6.11, is a variant of the *FiniteConfiguration* notation; its rounded corners indicate that it does not specifically represent a finite configuration or any of the three kinds of configuration. In addition, only the predicate and sub-configuration compartments may be filled; the sojourn time and activities compartments are empty. The predicate compartment is filled with the *predicate* that defines a proposition as described in the abstract syntax. Hence, a generic configuration notation is an abstract representation of any concrete configuration in which the proposition specified in it is true. A "hypothetical" configuration notation, described on the right of Figure 6.11, denotes a symbolic generic configuration notation, which is used to indicate that a configuration in which its specified proposition is satisfied is permissible at its location in the specification; it, however, does not enforce the existence of such configuration(s).

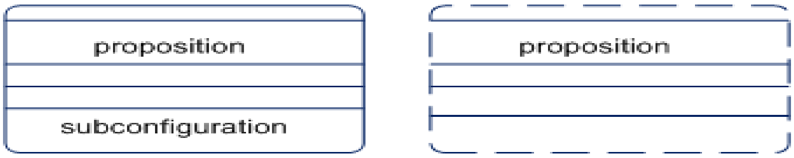


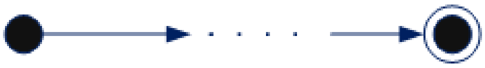
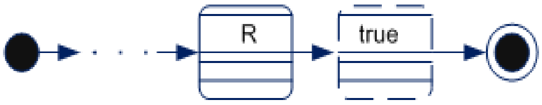
Figure 6.11 Generic and arbitrary configuration notations

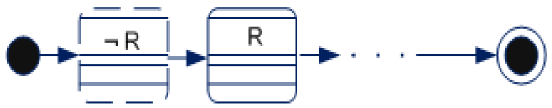
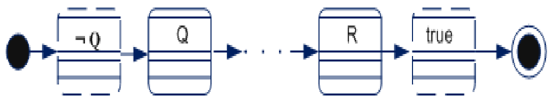
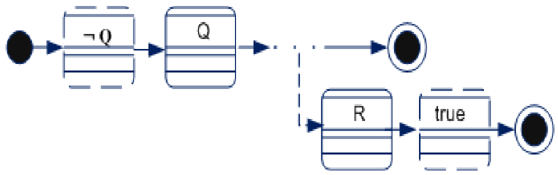
These two notations will be used, in the rest of this subsection, as the building blocks to specify temporal properties based on Dwyer's property patterns. Taking clue from the work of Klein and Giese [KG06, 07], we will first present the templates, based on the two notations above, for the property scopes; this is followed with the templates for the patterns themselves and how they fit into the different scopes.

6.3.2.1 Property scope notations

We present the concrete notations for expressing property scopes in Table 6.2. Recall from our previous discussion in Section 3.2.6 that a temporal property specification is an abstract assertion on a segment of the execution of a system; we denote the entire execution by the elements between the initial state (solid ball) and final state (bull's eye) symbols. Each of the scopes describes the segment of the entire execution within which the specified property pattern (represented by dotted lines) must hold. Thus, to use any of the scope templates, we replace the dotted lines with the property pattern to be checked. Note that the transitions between the generic and/or hypothetical configurations are abstract transitions without specific operations, triggers or output events. Hence, they do not specifically indicate any of the three kinds of configuration transition.

Table 6.2 HiLLS notations for property scope template

Scope	Property scope notations with examples	Descriptions
Globally	 Globally	The globally scope expresses that the property pattern... must be satisfied in every state throughout the execution between the initial and final states.
Before	 Before R	This specified property pattern ... must be satisfied *before* a transition into a state in which the predicate R is satisfied. The hypothetical state satisfying predicate *true* after R indicates that whatever property is satisfied in subsequent states is inconsequential.

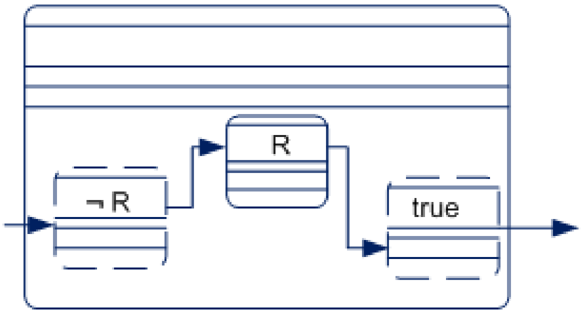
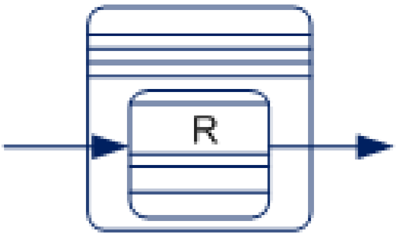
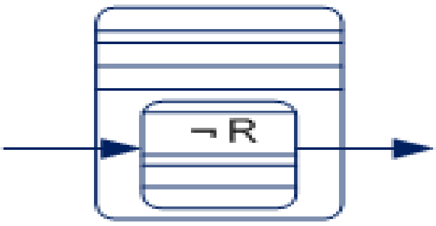
After	 **After R**	The property pattern ... must be satisfied *after* a transition into state in which the predicate R is satisfied. It does not matter whether R itself is satisfied from the initial state or not.
Between	 **Between Q and R**	Property pattern ... must be satisfied *after* a transition into a state in which the predicate Q is satisfied and *before* a transition into a state in which R is satisfied. Whatever happens afterwards is immaterial for this scope.
After until	 **After Q until R**	Property pattern ... must be satisfied *after* a transition into a state in which the predicate Q is satisfied, and continue to hold until a state in which R is satisfied occurs. If R does not occur then the scope of the specified pattern continues until the end of execution. Since the occurrence of R interrupts the scope of the specified pattern, we express the transition to the state satisfying R with an external transition notation. It, however, does not imply that the transition between such states in the system specification must be an external transition.

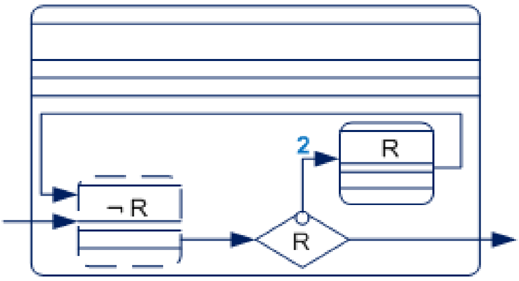
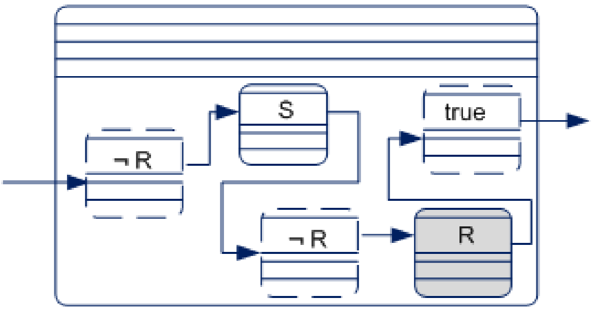
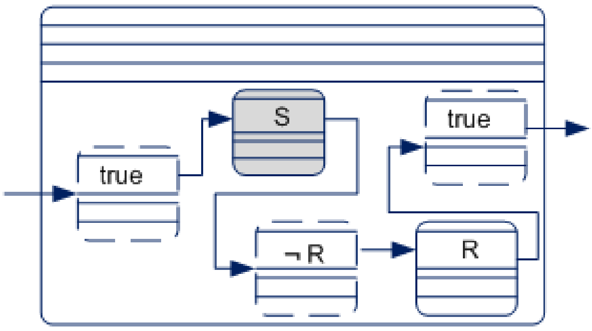
6.3.2.2 Property pattern notations

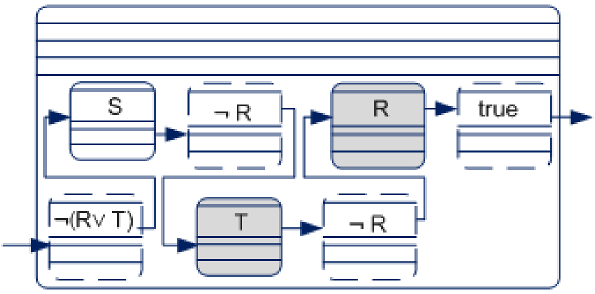
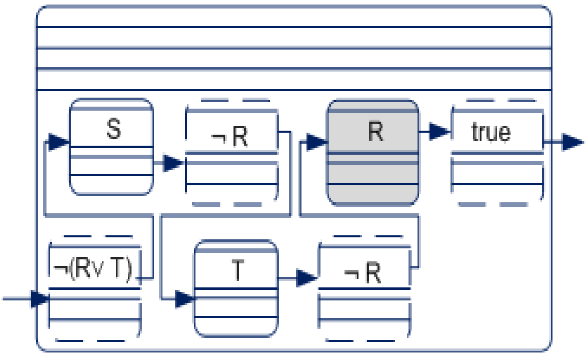
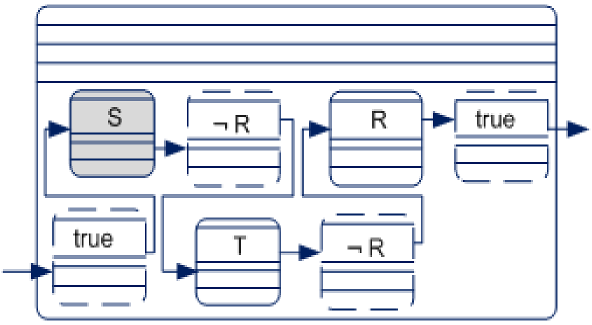
Both scopes and patterns are expressed using mixtures of generic and hypothetical configuration notations and abstract transitions between them. In order to distinguish the two parts in a property specification, we express patterns within composite generic configuration notations. Table 6.3 presents the concrete notations for the property patterns.

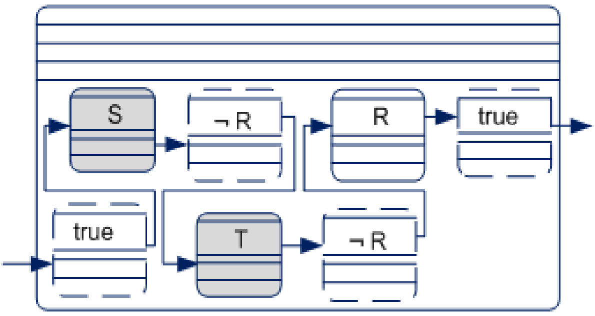
To disambiguate the different elements in the precedence and precedence chain patterns, the preceded states (effects) are represented with shaded generic configuration notations. For the same reason, trigger states are represented with shaded generic configurations in response and response chain patterns. In fact, the shaded elements mark the difference between precedence and response patterns and their respective chains.

Table 6.3 HiLLS notations for property pattern templates

Pattern	Property pattern notations with examples	Description
Existence (eventually)	 **R exists**	Pattern specifies that a transition into a state in which predicate R is satisfied must eventually occur (at least once) within the specified scope. The hypothetical configuration satisfying ¬R indicates that R does not necessarily have to be satisfied by the first state visited within the scope. Similarly, the second hypothetical configuration notation indicates that any predicate may be satisfied subsequent states within the scope.
Universality(always)	 **Always R**	The pattern species that all states visited (from the first to the last) within the associated scope must satisfy predicate R. Hence, there are no hypothetical configuration notations before or after the state of interest.
Absence (never)	 **Never R**	Predicate R must not hold in any of the states visited within the associated scope.

Pattern	Property pattern notations with examples	Description
Bounded existence	 **R exists at most twice**	This requires that a state where property R holds is visited at most two times (i.e., maximum frequency = 2) within the specified scope. The hypothetical state indicates that R may not necessarily hold in the first state. The transitions into R must, however be kept within the limit defined by the maximum frequency.
Precedence	 **S precedes R**	Within the associated scope, a transition to a state satisfying R must be preceded by a transition to a state in which S holds. The hypothetical configuration ¬R preceding S strictly enforces the requirement that R should not precede S while the one between S and R indicates that the latter does not necessarily have to follow the former immediately.
Response	 **R responds to S**	A state satisfying S must be followed by a transition to a state in which R holds within the specified scope. The hypothetical configuration ¬R between S and R indicates that response R does not necessarily have to follow trigger S immediately. To further disambiguate between the notations of *response* and *precedence* patterns, we use a shaded generic configuration for the trigger (S) in the former and for the event that follows the precedent (R) in the latter.

Pattern	Property pattern notations with examples	Description
1-2 Precedence chain	 **S precedes T, R**	Within the associated scope, a transition to a state satisfying T, followed (not necessarily immediately) by a state satisfying R must be preceded by a transition to a state in which S holds. The hypothetical configuration ¬R∨T indicates a state where R or T holds does not precede S. Similarly, the hypothetical configuration ¬R following S requires that R must not hold between S and T while the one following T implies that it is not necessary that R follows T immediately.
2-1 Precedence chain	 **S, T precede R**	This pattern specifies that a sequence of transitions to states satisfying S and T (not necessarily successively) must precede a visit to a state in which R holds within the associated scope.
2-1Response chain	 **T, R respond to S**	This pattern specifies that a sequence of transitions to states satisfying T and R (not necessarily successively) must respond to (i.e., be triggered by) a visit to a state in which S holds within the associated scope.

Pattern	Property pattern notations with examples	Description
1-2 Response chain	 R respond to S, T	Within the associated scope, a transition to a state satisfying S, followed (not necessarily immediately) by a state satisfying T must trigger (be responded to by) a transition to a state in which R holds.

To use the property notations for the specification of temporal requirements, the modeler needs to map each required property to a pair of pattern and scope selected from Table 6.3 and Table 6.2 respectively. Then the dotted section of the scope is replaced with the pattern template to complete a template for the property.

In future work, we intend to explore the provision of HiLLS-based notations for timing requirements also. The next section presents the HiLLS specification of the BVS running example, which includes the system and requirement models (using the property notations provided in this subsection).

6.4 HiLLS SPECIFICATION OF THE BVS

In this section, we demonstrate the application of HiLLS concrete notations for system and requirement modeling by presenting the HiLLS specification of the running example in this thesis: the Beverage Vending System (BVS), the synopsis of which has been provided in Section 3.2.1.

As described in the synopsis, BVS is a composite system with two components: Beverage Vending Machine (BVM) and the user (BVMUser). It also states some temporal properties, which BVM must satisfy. We will present the HiLLS specification in a bottom-up approach. i.e., first, we present the components -BVM and BVMUser - and their requirements (if any); then, we will present the BVS and its relationship with its components as well as the link between the BVM and a model of its required properties.

6.4.1 HiLLS specification of BVM

We present the HiLLS model of the BVM in Figure 6.12. BVM has two input ports and two output ports. The type of output port *outC* is a bag of coins. To describe the system's behavior, BVM has six configurations: *idle*, *charge*, *reject*, *return*, *dispense* and *cancel* each having a unique property specified by the predicate (on the state variables) in its properties compartment.

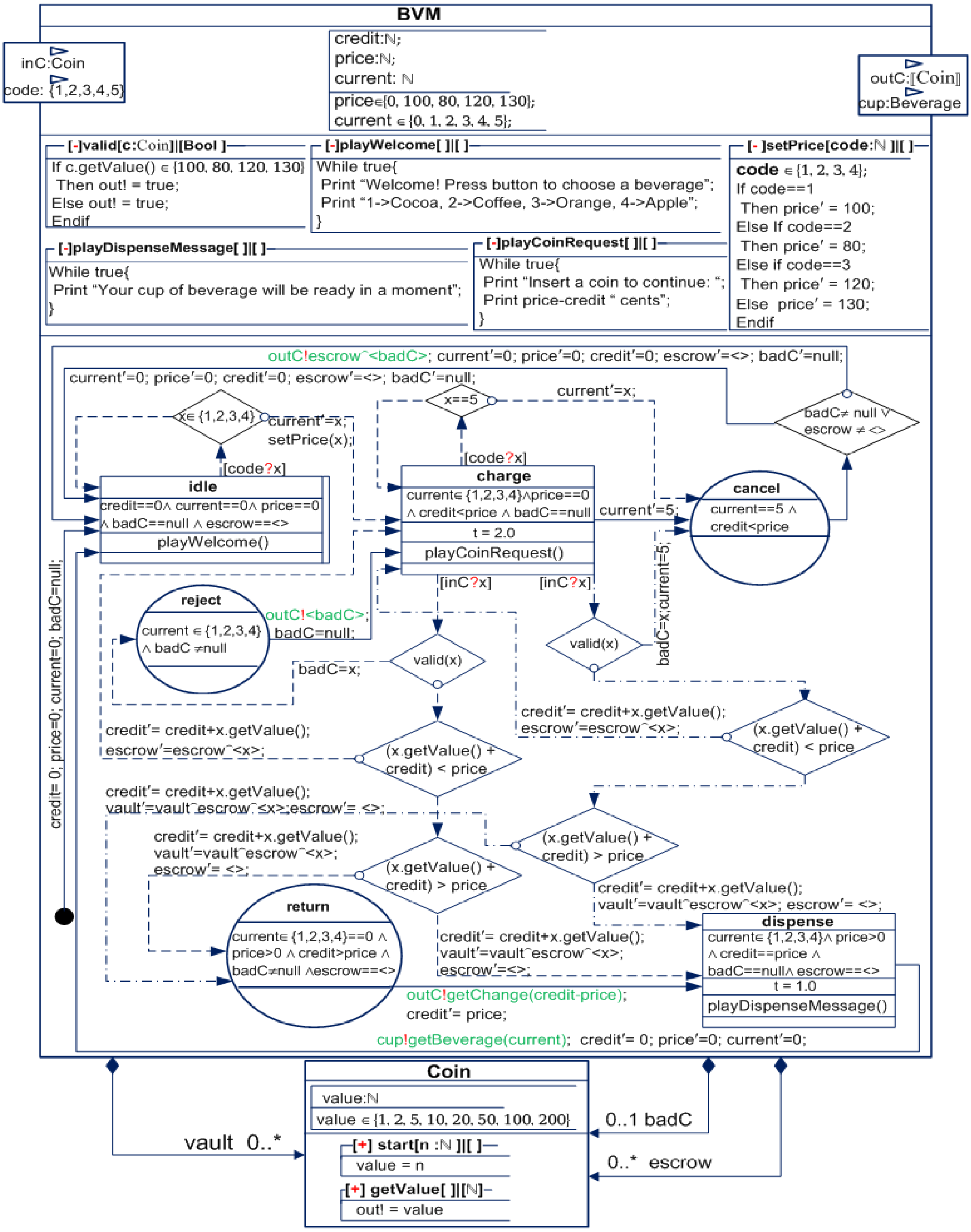


Figure 6.12 HiLLS model of the BVM

For instance, it transits to configuration *idle* whenever predicate $credit == 0 \land current == 0 \land price == 0 \land badC == null \land escrow ==<>$ is true. *Idle* is a passive configuration, *charge* and *dispense* are finite configurations while *reject*, *return* and *cancel* are transient configurations; hence only *charge* and *dispense* need to have their *sojourn times* explicitly defined, others are implicit. i.e., zero for transient configurations and positive infinity for passive configurations. The configuration transition diagram specifies the BVM's behavior as follows:

The system initializes to configuration *idle*. It remains in this *idle* while playing the welcome message as specified in the activities compartment until it receives an input $x \in \{1,2,3,4\}$ on input port *code*; the input triggers an external transition to configuration *charge* while the received x is assigned to state variable *current* and used as argument to the operation that computes the value of variable *price*. Note that the code *x* received indicates an order for a beverage.

Once it assumes configuration *charge*, BVM awaits the receipt of coins within its set of acceptable coins to fund the ongoing transaction while displaying the amount of coins expected in its activities. If does not receive any input within 2 minutes, which is the sojourn time of the configuration, an internal transition to configuration *cancel* is fired by doing the computation $current = 5$, to automatically abort the transaction. An input $x == 5$ on input port *code* also triggers an external transition to *cancel* while assigning *x* to *current*. If a coin *x* is received on input port *inC*, it first checks whether *x* is an acceptable coin or not; if not acceptable, *x* is assigned to variable *badC*, leading to an external transition to configuration *reject*. This leads to an instantaneous output of *badC* on output port *outC* and an internal transition back to *charge*. If *x* is a valid coin, a check is done to see whether its value makes the condition $x.getValue + credit < price$ is true or not. If the condition is true, *x* is not sufficient to pay the price of the selected beverage; hence, the eternal transition targets configuration *charge* to await more coins. If the condition is false, then *x* is either exactly equal to, or greater than the amount required to complete the transaction, so a test for a last condition $x.getValue + credit > price$ is checked. If this condition is false, then *x* is just sufficient for the transaction and the transition targets configuration *dispense*. If condition is true, it implies that the value of *x* is greater than the amount required; hence, the transition terminates on configuration *return*. Whenever it assumes configuration *return*, BVM immediately outputs a bag of coins of value $credit - price$ and fires an internal transition to configuration *dispense*.

A confluent transition may also occur if a coin x is received BVM has been in configuration *charge* for exactly 2 minutes. It first checks whether *x* is valid or not as in the case of an external transition; however, an invalid coin in this case leads to an automatic cancelation of the transaction. If *x* is valid, the confluent transition follows similar paths as the external transition.

BVM spends about 1 minute in configuration *dispense* while the requested beverage is being prepared; the activity of the configuration is to continuously display a message to that the requested beverage is on its way. At the end of the sojourn time, it outputs the beverage on output port *cup* and immediately does an internal transition to configuration *idle*.

The moment BVM enters configuration *cancel*, it immediately checks whether a coin has been stored in *badC* and/or whether *escrow* is not empty. *badC* will be occupied if *cancel* is entered through a confluent transition from *charge*; *escrow* may not be empty if the system has been iterating in configuration *charge* before making a transition to *cancel* either through an internal or confluent transition. If either or both of the two variables is/are occupied, the occupant(s)will be sent out on port *outC* before reinitializing the system to configuration *idle*.

6.4.2 HiLLS Specification of BVMRequirement

In this subsection, we discuss the steps taken to model the three required temporal properties of the BVM. As reminder, we have matched the required temporal properties of BVM to some property patterns in Section 3.2.6.4. We will repeat the matching in this subsection and then demonstrate how to express them with the HiLLS' notations proposed in this chapter for modeling the property patterns and scopes.

6.4.2.1 Temporal property I: BVM must not dispense unless enough coins are inserted to pay for a selected drink

We can rephrase this property to match the precedence chain pattern as:

The selection of a drink, and acquisition of sufficient coins always precede the dispense of selected drink.

This statement matches with the property pattern "S*, T, precede R globally"* where:

S = "a drink is selected", *T*= "sufficient coins have been acquired" and *R* = "selected drink is dispensed". From the BVM model, we know that $current \in \{1, 2, 3, 4\}$ when a drink has been ordered. $credit \geq price$must be true when sufficient coins have been cumulated for a transaction and $current \in \{1, 2, 3, 4\} \wedge price > 0 \wedge credit = price$ must hold when the ordered drink is being dispensed. i.e., $S \coloneqq current \in \{1, 2, 3, 4\}$, $T \coloneqq credit \geq price$ and $R \coloneqq current \in \{1, 2, 3, 4\} \wedge price > 0 \wedge credit = price$.

We can construct the property "S*, T, precede R globally"* by substituting the template of pattern S*, T, precede R* (see Table 6.3) for the dotted line in the template of scope *globally* (see Table 6.2). Figure 6.13 shows the outcome of this substitution.

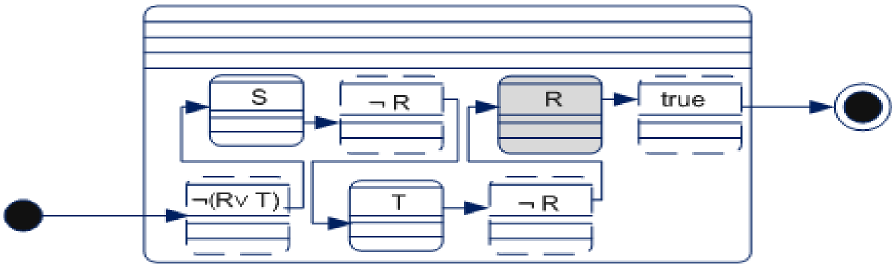


Figure 6.13 HiLLS notation for property "S, T precede R globally"

By substituting the predicates R, S and T in Figure 6.13, we have the HiLLS specification of the property as shown in Figure 6.14.

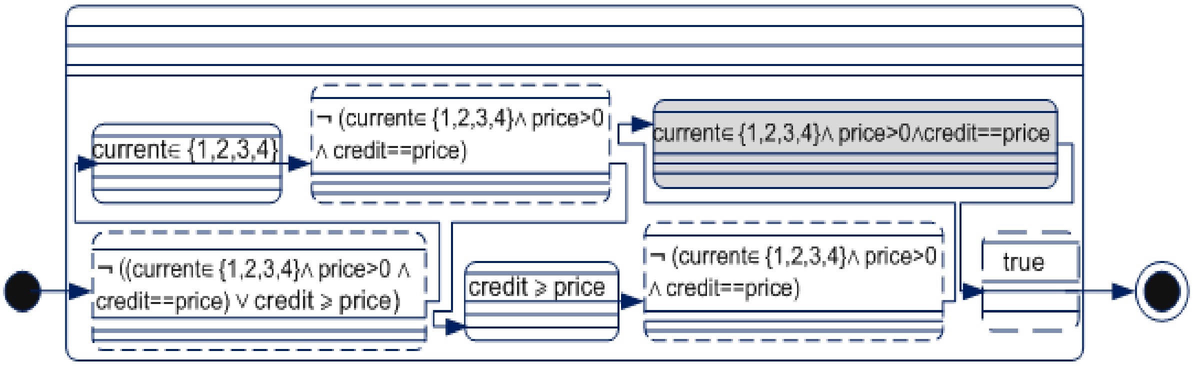

current∈ {1,2,3,4}
¬ (current∈ {1,2,3,4}∧ price>0 ∧ credit==price)
current∈ {1,2,3,4}∧ price>0∧credit==price
¬ ((current∈ {1,2,3,4}∧ price>0 ∧ credit==price) ∨ credit ⩾ price)
credit ⩾ price
¬ (current∈ {1,2,3,4}∧ price>0 ∧ credit==price)
true

Figure 6.14 HiLLS specification of temporal property I of BVM
BVM must not dispense unless enough coins are inserted to pay for a selected drink

6.4.2.2 Temporal property II: BVM should always refund the balance whenever excess coins are inserted

We rephrase this requirement to match the response pattern as:

Refund balance responds to excess payments always

This statement matches with the pattern "*R responds to S globally*" where *R* = "refund balance occurs" and *S* = "excess coins have been inserted".

Excess coins have been inserted when $credit > price$ and refund of balance has occurred when *credit* reduces to the value of *price*, i.e., $credit = price$.

Therefore, $S \coloneqq credit > price$ and $R \coloneqq credit = price$.

Figure 6.15 shows the outcome of substituting the template of pattern *R responds to S* (see Table 6.3) for the dotted line in the template of scope *globally* (see Table 6.2).

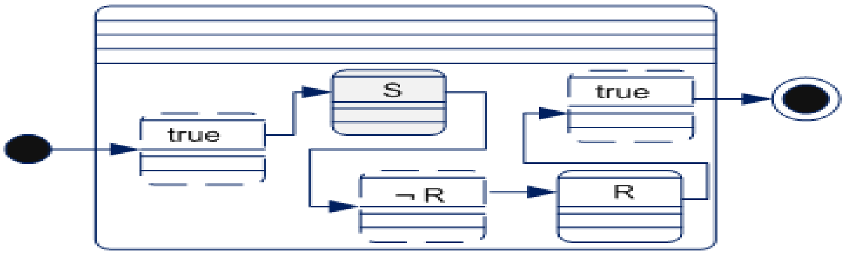


Figure 6.15 HiLLS representation of temporal property "R responds to S globally"

By substituting the predicates R and S in Figure 6.15, we have the HiLLS specification of the property as shown in Figure 6.16.

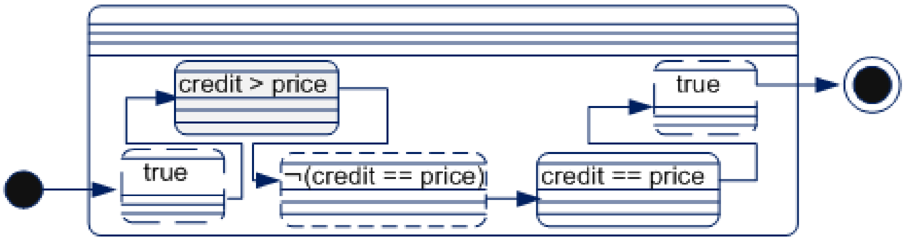


Figure 6.16 HiLLS specification of temporal property II of BVM
BVM should always refund the balance whenever excess coins are inserted

6.4.2.3 Temporal property II: Once the payment for a drink is complete, the transaction cannot be canceled any longer

Again, we can rephrase this property to match with the absence property pattern as follows:

Transaction is canceled is not allowed after sufficient coins have been acquired for the transaction.

This matches with the occurrence pattern *S is absent after R* with *S* = "transaction is canceled" and *R* = "sufficient coins have been acquired for the transaction".

From the system specification, we know that $current\ == \ 5$ is necessary to cancel a transaction and sufficient coins have been acquired when $credit \geq price$. Therefore, $S \coloneqq current\ == \ 5$ and $R \coloneqq credit \geq price$.

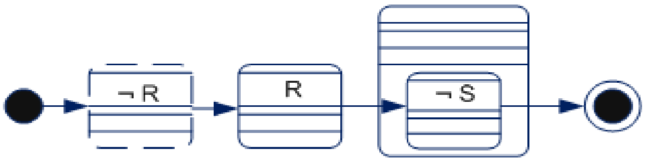


Figure 6.17 HiLLS template for temporal property "S is absent after R"

We substitute the *absence* pattern template in the dotted segment of the *after* scope and we have the template for property *S is absent after R* as shown in Figure 6.17.

We substitute the predicates R and S in Figure 6.17 and the HiLLS specification of the required property is as shown in Figure 6.18

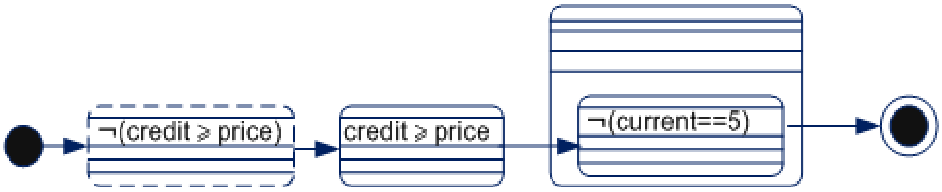


Figure 6.18HiLLS specification of temporal property III of BVM
Once the payment for a drink is completed the transaction cannot be canceled any longer

6.4.2.4 BVM requirements

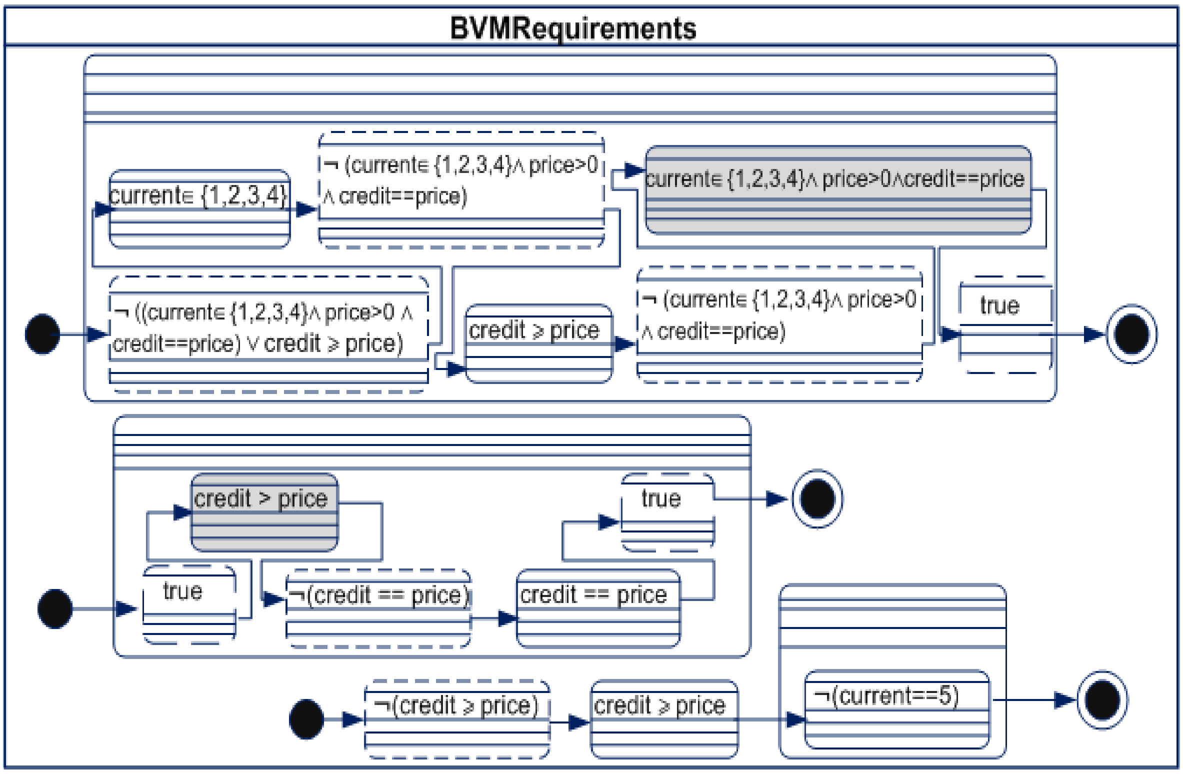

BVMRequirements
current∈{1,2,3,4}
¬ (current∈{1,2,3,4}∧ price>0 ∧ credit==price)
current∈{1,2,3,4}∧ price>0∧credit==price
¬ ((current∈{1,2,3,4}∧ price>0 ∧ credit==price) ∨ credit ⩾ price)
credit ⩾ price
¬ (current∈{1,2,3,4}∧ price>0 ∧ credit==price)
true
credit > price
true
true
¬(credit == price)
credit == price
¬(credit ⩾ price)
credit ⩾ price
¬(current==5)

Figure 6.19 HiLLS model of BVM's requirements

Figure 6.19 shows the HiLLS model of the BVM's requirements showing the specifications of all the three properties in its second compartment.

6.4.3 HiLLS Specification of BVMUser

Figure 6.20 presents the specification of BVMUser. It also uses the same state variables, and for the same purposes, as its DEVS specification in Section 3.2.3.2.

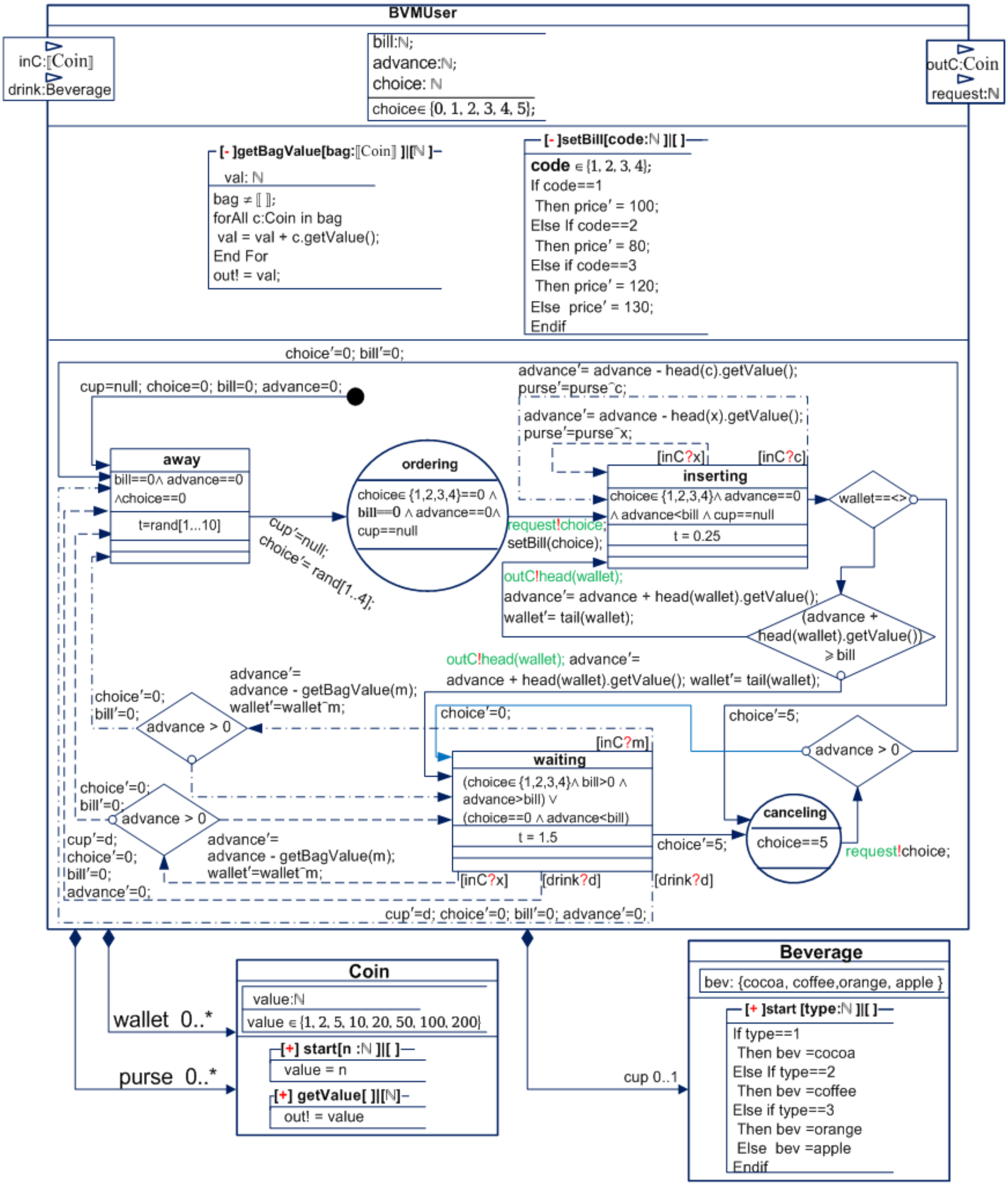


Figure 6.20 HiLLS specification of BVMUser

BVMUser initializes to *away* and remains in the configuration for a random period of between 1 and 10 minutes before doing an internal transition to *ordering*. Before entering *ordering*, a random integer from 1 to 4 is assigned to variable $choice$ and cup is reset to null. Once it assumes configuration *ordering*, BVMUser is ready to place an order for a beverage; it outputs *choice* on port *request*, computes the *bill* and immediately transits to *inserting* to commence the payment.

BVMUser stays in configuration *inserting* for 0.25 minute to get a coin from the wallet. While in this configuration, if a coin x is received on port inC, x is considered to be a coin that has been previously sent out, which is rejected because it is not acceptable for the ongoing transaction; hence, it is kept in the *purse* while its value is deducted from *advance* before going back to *inserting*. If BVM receives a coin *c* on port *inC* at the end of *inserting*, it will be treated the same way, as in the external transition, during the confluent transition back to *inserting*.

At the end of *inserting*, the target of the internal transition depends firstly on the status of the $wallet$. An empty *wallet* implies that BVMUser has run out of coins to complete the transaction; hence, the transition targets configuration *canceling* to order the termination of the transaction in progress. If $wallet$ is not empty, then the final path of the transition depends on the condition $advance + head(wallet).getValue \geq bill$. If this condition is false, it implies that the next coin picked from the *wallet* is not sufficient to complete the transaction; hence, the picked coin is sent out on port $outC$ and its (the coin picked from w*allet*) valued added to *advance* before terminating the transition in *inserting*. If the condition is true, however, it implies that the next coin from *wallet* is sufficient to complete the ongoing transaction, probably with some balance; hence, BVMUser sends out the coin, adds its value to *advance* and transits to configuration *waiting* to await the ordered beverage, and possibly some balance.

BVMUser assumes the *canceling* configuration whenever it is ready to abort an ongoing transition. It immediately sends out the value of variable *choice* on port *request* and the target of the internal transition that immediately follows depend on the status of variable *advance*.$advance == 0$ implies that the net value of coins expended by BVMUser on the transaction is zero; hence, it is reinitialized to configuration *away*. If $advance > 0$ however, the transition targets configuration *waiting* to await the refund of already expended coins.

If BVMUser stays in configuration *waiting* for 1.5 minutes without receiving any input, an internal transition to *canceling* occurs to request the termination of the transaction. If BVMUser receives a bag of coins during or at the end of its sojourn in configuration *waiting*, the same computations will accompany the external or confluent transition that triggered and the targets will be the same under the same conditions. In either case, the cumulative value of all the coins in the bag received will be deducted from *advance* while the coins are deposited in the *wallet*. Then the final path of the transition depends on condition $advance > 0$. $advance > 0$ implies that the coins received is a balance for a completed transaction; hence the system returns to *waiting* in anticipation of the ordered cup of beverage. $advance == 0$ implies that the received coins is a refund for a canceled transaction and BVMUser has nothing else to wait for; hence, it is reinitialized to configuration *away*. Finally, the receipt of a beverage *d* on input port *drink* either during or at the end of *waiting* leads to a transition to *away* while *d* is assigned to variable *cup*.

6.4.4 BVS' Structure and BVM's Requirements

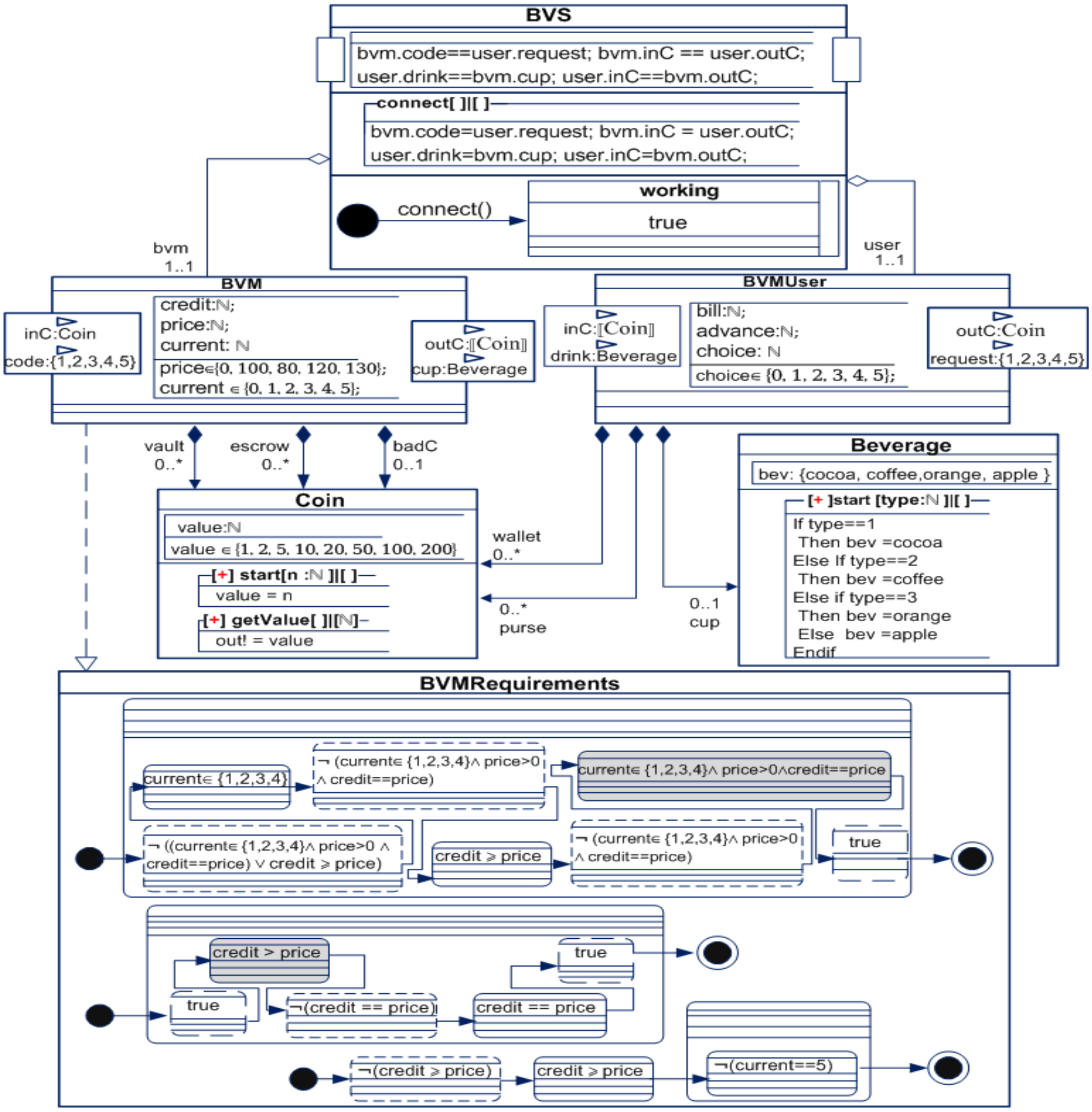


Figure 6.21 HiLLS model of BVS

Figure 6.21 presents the entire HiLLS specification of the case study, showing the BVS, its components -BVM (and its requirements) and BVMUser with their operations and behavior compartments collapsed - and the relationships between them.

As described in the model, BVS (on top of the figure) has two components: *bvm* (an instance of BVM) and *user* (an instance of BVMUser).

BVS is a closed system, hence its input and output interfaces are empty. It has no sate variables but the state schema specifies constraints on the couplings between the ports of its components. For instance, predicate $bvm.code == user.request$ specifies that the value of message on port *code* of *bvm* is always equal to that on port *request* of *user*. Operation *connect*() establishes the port couplings when the system is being initialized into its only state. A coupling is established between two ports by assigning the value of the *sender* to that of *receiver*. For example, $bvm.code = user.request$ assigns the message on *user*'s *request* port to *bvm*'s *code* port.

The *HRequirement* model *BVMRequirements* at the bottom of Figure 6.21 specifies the required temporal properties of BVM. The "*satisfies"* relation between BVM and *BVMRequirements* indicates that the former satisfies the properties in the latter.

6.5 CONCLUSION

In this chapter, we have presented the HiLLS' abstract and concrete syntaxes and an informal mapping between them. We have built the HiLLS' abstract syntax from a disciplined integration of system-theoretic and software engineering concepts to capture, in a considerably generic form, the different concerns of simulation, formal analysis and enactment methodologies for DES in a coherent whole. Specifically, we adopt system-theoretic concepts from DEVS to capture DESs in a considerably generic form, concepts from Object-Z for rigorous refinements of abstract DEVS-based concepts and to make models amenable to formal analysis. We also adopt concepts from a pattern-based classification of Temporal Logic specifications of commonly occurring temporal requirements to express required properties in HiLLS. The chapter presents in details, the steps we have taken to apply metamodel integration techniques for a disciplined integration of the disparate concepts adopted from independent sources.

To build the concrete notations for HiLLS' concepts, we adopt and extend some notations from the UML family of notations to make the language easy to learn and use. In addition to the graphical notations for system modeling, we also propose graphical notations, similar to the graphical elements for describing systems' behaviors, to express the required temporal properties of systems under study. We believe that providing similar sets of notations to describe systems and their requirements is a step towards bridging the chasm between simulation and formal analysis methodologies. It may also stimulate further research into finding common grounds for practitioners of the disparate analysis methodologies.

To demonstrate the use of HiLLS for system and requirement specification, we have presented the HiLLS specification of the running example in this thesis - the Beverage Vending System (BVS) - and its required properties with detailed discussions of how HiLLS constructs express different aspects of the system.

In future work, we intend to explore the specification of timing requirements in addition to temporal properties, with similar notations. There is an ongoing research towards formalizing the HiLLS' concrete syntax and, subsequently, develop a model editor for the language to allow for the synthesis of artifacts for simulation, formal analysis and enactment as envisioned in Chapter 4.

7 HiLLS' SEMANTICS

7.1 INTRODUCTION

We have presented the abstract and concrete syntaxes of HiLLS in the previous chapter. In this chapter, we build on parts of the results we reported in [AMT16, AT16] to present the language's semantics. Recall from our discussion in Section 3.5 that a language's abstract syntax may be mapped to one or more semantics domains; hence, providing different kinds of semantics of models for different purposes and/or audiences. Figure 7.1 presents an overview of the semantics framework of HiLLS covered in this thesis. Using the translational semantics technique, the HiLLS' abstract syntax is mapped to four semantics domains: DEVS for the purpose of simulation, Z and Temporal Logic for the purpose of formal (logical) analysis, and the DEVS-based enactment framework presented in Chapter 5 for the purpose of enactment. Its compatibility with the underlying formalisms of the three purposes - simulation, formal analysis and enactment- is, in fact, what makes HiLLS fit in the kernel of the integrative MDSE framework (SimStudio II) presented in Chapter4.

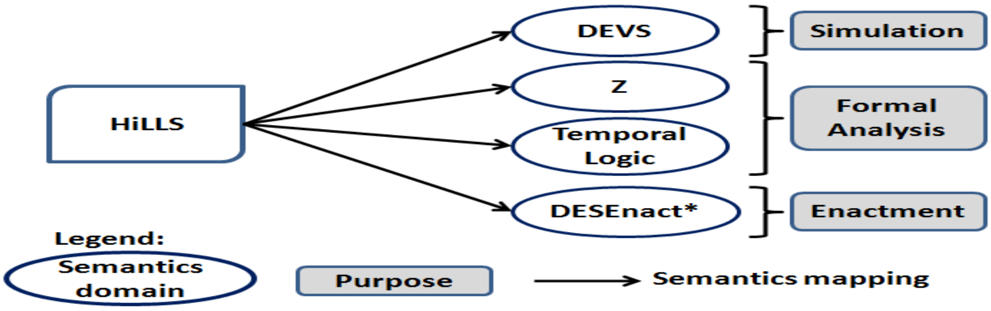


Figure 7.1 Semantics framework of HiLLS
*DESEnact refers to the DEVS-based enactment framework presented in Chapter 5

As explained previously in Chapter 5, we have chosen DEVS as the semantics domain to provide simulation capability for HiLLS because of the universality of the former to express most kinds of DES [Van00] and the availability of tools implementing its simulation protocols. Examples of the many, and evolving, published DEVS-based M&S tools or environments include DEVSJava [SZ98, Mat03], DEVS-Suite [KSE09], MS4Me [ZS13, SZC+13], CD++ [WCD01, Wai02, BWC13], PowerDEVS [KLP03, BK11], LSIS-DME [HZ07], DEVSimPy [CSP+11, CS15], and so on. Z and TL have also been chosen, for the same reason, as the semantics domains for logical analysis. Particularly, the Community Z Tools (CZT) [MU05, MFM+05] links Z with a host of evolving tools for formal analysis.

The rest of the chapter is structured to present the different branches of HiLLS' semantics in separate sections. Sections7.2, 07.3 and 7.4 present the simulation semantics, logical (formal analysis) semantics and execution (enactment) semantics respectively. In each of the sections, we discuss (informally) how the different models of the BVS (the running example) presented in Chapters 3 and 5 can be derived from its HiLLS specification. Finally, we conclude the chapter in Section 7.5.

7.2 SIMULATION SEMANTICS

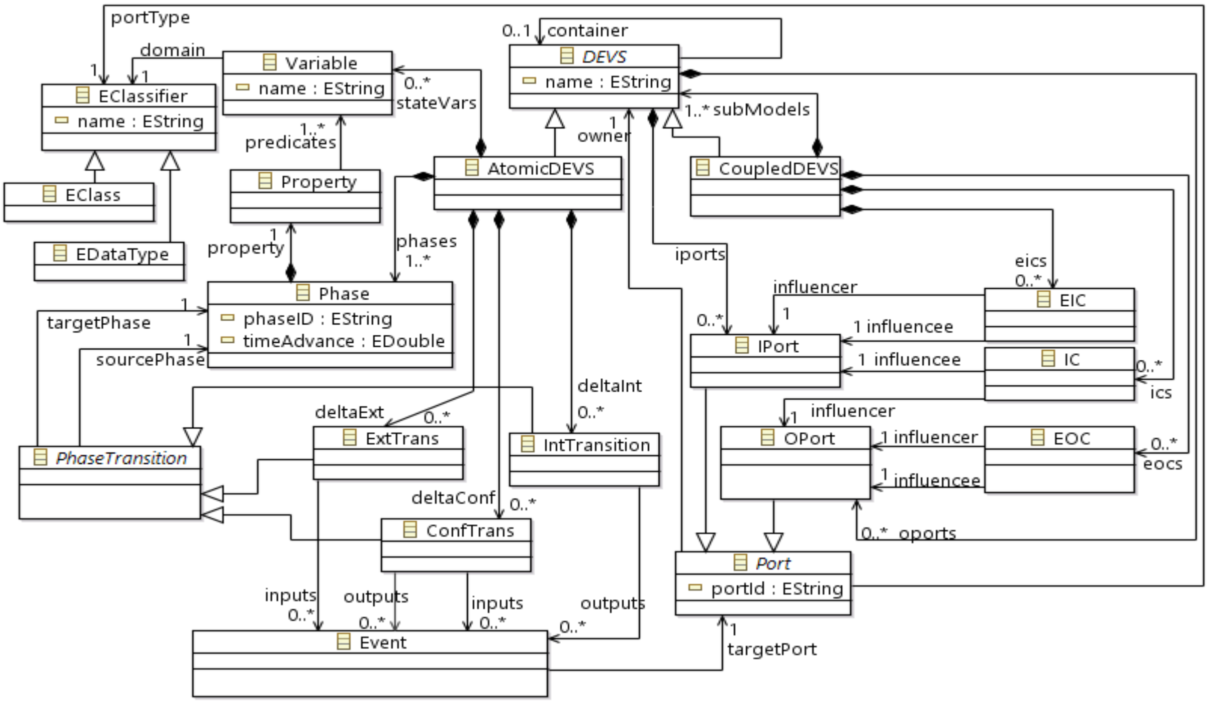
In this section, we present a DEVS metamodel and show the concepts it describes may be derived from a given HiLLS model. Then, we demonstrate with a discussion of the correspondences between the HiLLS model of the BVS and its DEVS model.

7.2.1 DEVS Metamodel

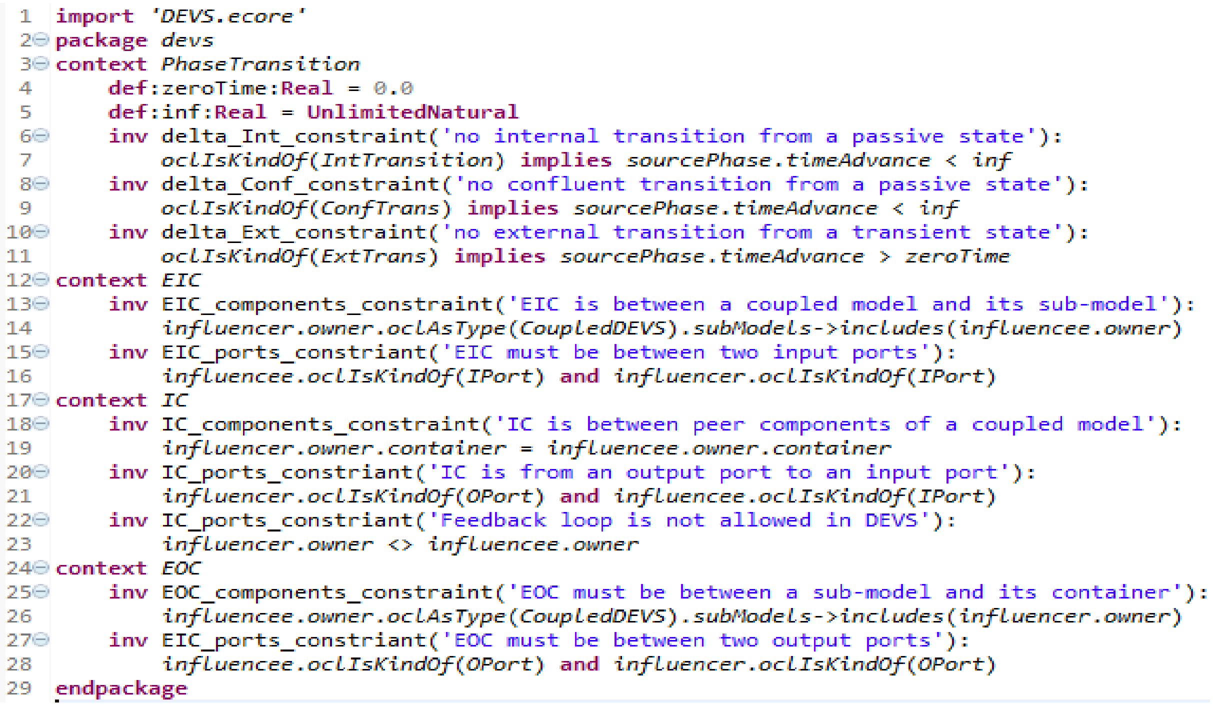
Though DEVS is considered universal for modeling DESs, has many areas of application and enjoys a plethora of supporting tools, there is not yet a standard DEVS metamodel, which is generally agreed upon by the community to drive MDSE practices with the formalism. Hence, everybody uses the mathematical specification of the language as a metametamodel to build a DEVS metamodel that is most suitable for the intended use without violating the language's constraints. Figure 1.1Figure 7.2presents a simple DEVS metamodel, which we have defined in conformity to the mathematical specification, as presented in Section 3.2.2, and reported in [AMT16].

As shown by the two sub-types of the abstract *DEVS* class in Figure 7.2(a), DEVS describes a DES as either an *AtomicDEVS* or *CoupledDEVS*. A DEVS model may have zero or more input ports, *iports*, and/or zero or more output ports, *oports*; a port is defined by a name, *portid*, and a type, *portType*, which may be a Class or primitive Data Type. An *AtomicDEVS* defines state variables, *stateVars* and a finite set of *phases* where a *Phase* is an abstraction of a unique combination of values of (or predicate on) the state variables. Technically, the phases constitute disjointed subsets of the state space. A typical example of *phase* variable is the variable ϕ in our DEVS specification of the BVS in Section 3.2.3. A phase is characterized by a *timeAdvance*.

An *AtomicDEVS* may also define sets *deltaInt*, *deltaExt* and *deltaConf* of internal, external and confluent phase transitions, respectively. Each *IntTransition* and *ConfTransition* may be accompanied by a bag of *outputs* events while every *ExtTransition* and *ConfTransition* is triggered by a bag of *intputs* events. Additional information on phase transitions is provided in the OCL (Object Constraint Language) constraints in Figure 7.2(b). The constraints define some restrictions on exceptional cases in which each of the transitions may not occur. A *CoupledDEVS* defines a set, A *subModels*, of at least one component(s) of the A *container* model. It also defines the sets *eics*, *ics* and *eocs* of couplings between the components of the model. Figure 7.2(b) also provides additional information and restrictions on the three kinds of couplings in accordance to the DEVS formalism.



(a) DEVS Metamodel



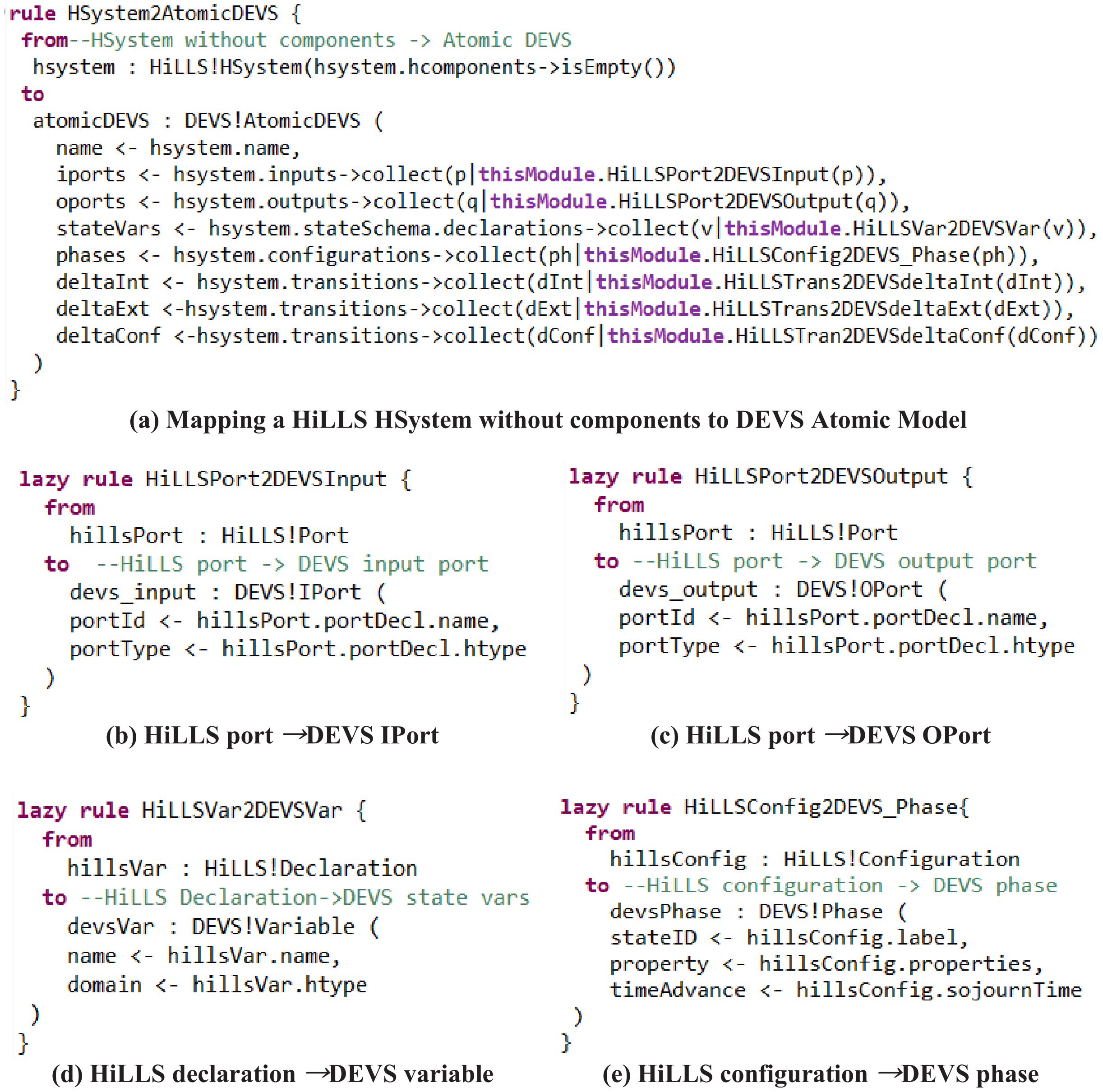
```
import 'DEVS.ecore'
package devs
context PhaseTransition
    def:zeroTime:Real = 0.0
    def:inf:Real = UnlimitedNatural
    inv delta_Int_constraint('no internal transition from a passive state'):
        oclIsKindOf(IntTransition) implies sourcePhase.timeAdvance < inf
    inv delta_Conf_constraint('no confluent transition from a passive state'):
        oclIsKindOf(ConfTrans) implies sourcePhase.timeAdvance < inf
    inv delta_Ext_constraint('no external transition from a transient state'):
        oclIsKindOf(ExtTrans) implies sourcePhase.timeAdvance > zeroTime
context EIC
    inv EIC_components_constraint('EIC is between a coupled model and its sub-model'):
        influencer.owner.oclAsType(CoupledDEVS).subModels->includes(influencee.owner)
    inv EIC_ports_constraint('EIC must be between two input ports'):
        influencee.oclIsKindOf(IPort) and influencer.oclIsKindOf(IPort)
context IC
    inv IC_components_constraint('IC is between peer components of a coupled model'):
        influencer.owner.container = influencee.owner.container
    inv IC_ports_constraint('IC is from an output port to an input port'):
        influencer.oclIsKindOf(OPort) and influencee.oclIsKindOf(IPort)
    inv IC_ports_constraint('Feedback loop is not allowed in DEVS'):
        influencer.owner <> influencee.owner
context EOC
    inv EOC_components_constraint('EOC must be between a sub-model and its container'):
        influencee.owner.oclAsType(CoupledDEVS).subModels->includes(influencer.owner)
    inv EIC_ports_constraint('EOC must be between two output ports'):
        influencee.oclIsKindOf(OPort) and influencer.oclIsKindOf(OPort)
endpackage
```

(b) DEVS Static constraints

Figure 7.2 DEVS metamodel and static constraints [AMT16]

7.2.2 HiLLS to DEVS Mapping

Using model transformation technique, we show the correspondences between the concepts and relationships described in the metamodels of HiLLS (see Figure 6.8 and Figure 6.9) and DEVS. The HiLLS-to-DEVS mapping rules, specified in ATLAS Transformation Language (ATL) [JAB+06], have been reported in [AMT16].



```
rule HSystem2AtomicDEVS {
 from--HSystem without components -> Atomic DEVS
  hsystem : HiLLS!HSystem(hsystem.hcomponents->isEmpty())
 to
  atomicDEVS : DEVS!AtomicDEVS (
    name <- hsystem.name,
    iports <- hsystem.inputs->collect(p|thisModule.HiLLSPort2DEVSInput(p)),
    oports <- hsystem.outputs->collect(q|thisModule.HiLLSPort2DEVSOutput(q)),
    stateVars <- hsystem.stateSchema.declarations->collect(v|thisModule.HiLLSVar2DEVSVar(v)),
    phases <- hsystem.configurations->collect(ph|thisModule.HiLLSConfig2DEVS_Phase(ph)),
    deltaInt <- hsystem.transitions->collect(dInt|thisModule.HiLLSTrans2DEVSdeltaInt(dInt)),
    deltaExt <-hsystem.transitions->collect(dExt|thisModule.HiLLSTrans2DEVSdeltaExt(dExt)),
    deltaConf <-hsystem.transitions->collect(dConf|thisModule.HiLLSTran2DEVSdeltaConf(dConf))
  )
}
```

(a) Mapping a HiLLS HSystem without components to DEVS Atomic Model

```
lazy rule HiLLSPort2DEVSInput {
  from
    hillsPort : HiLLS!Port
  to  --HiLLS port -> DEVS input port
    devs_input : DEVS!IPort (
    portId <- hillsPort.portDecl.name,
    portType <- hillsPort.portDecl.htype
  )
}
```

(b) HiLLS port →DEVS IPort

```
lazy rule HiLLSPort2DEVSOutput {
  from
    hillsPort : HiLLS!Port
  to --HiLLS port -> DEVS output port
    devs_output : DEVS!OPort (
    portId <- hillsPort.portDecl.name,
    portType <- hillsPort.portDecl.htype
  )
}
```

(c) HiLLS port →DEVS OPort

```
lazy rule HiLLSVar2DEVSVar {
  from
    hillsVar : HiLLS!Declaration
  to --HiLLS Declaration->DEVS state vars
    devsVar : DEVS!Variable (
    name <- hillsVar.name,
    domain <- hillsVar.htype
 )
}
```

(d) HiLLS declaration →DEVS variable

```
lazy rule HiLLSConfig2DEVS_Phase{
  from
    hillsConfig : HiLLS!Configuration
  to --HiLLS configuration -> DEVS phase
    devsPhase : DEVS!Phase (
    stateID <- hillsConfig.label,
    property <- hillsConfig.properties,
    timeAdvance <- hillsConfig.sojournTime
  )
}
```

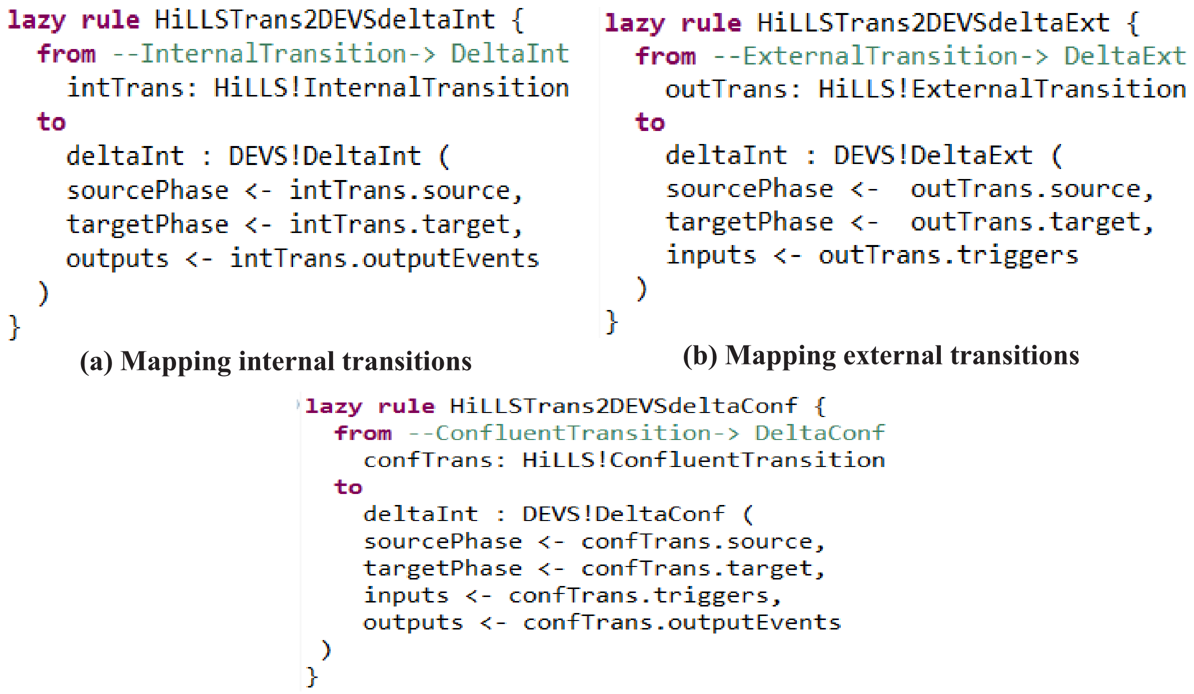
(e) HiLLS configuration →DEVS phase

Figure 7.3Mapping rules of HiLLS concepts to Atomic DEVS concepts [AMT16]

Figure 7.3(a) shows the semantics mapping rules to obtain an *AtomicDEVS* from a given *HSystem* with an empty *hComponents* (set of components). The elements of *AtomicDEVS* as described in the DEVS metamodel are shown on the left-hand side of the rule with the corresponding *HSystem* elements on the right-hand side.

The ―lazy rules” in Figure 7.3(b) and Figure 7.3(c) provide the rules mapping individual HiLLS input and output ports to DEVS input and output ports with HiLLS port declaration name and type mapping to DEVS port id and type, respectively. These ―lazy” rules are invoked from the *AtomicDEVS* and *CoupledDEVS* rules to obtain their input and output ports. In Figure 7.3(d) and Figure 7.3(e), we show the mapping rules to obtain DEVS state variables and phases from HiLLS state variables and configurations, respectively.

Similarly, individual HiLLS' *InternalTransition*, *ExternalTransition* and *ConfluentTransition* are mapped to DEVS' *DeltaInt*, *DeltaExt* and *DeltaConf* respectively by the rules in Figure7.4. It is important to state here that the imperative HiLLS computations that accompany the transitions cannot be explicitly accounted for in this declarative mapping. However, we can use a Model-to-Text transformation technique, with one of the DEVS-based M&S tools as target, to generate codes with such details for the target platform.



```
lazy rule HiLLSTrans2DEVSdeltaInt {
  from --InternalTransition-> DeltaInt
    intTrans: HiLLS!InternalTransition
  to
    deltaInt : DEVS!DeltaInt (
    sourcePhase <- intTrans.source,
    targetPhase <- intTrans.target,
    outputs <- intTrans.outputEvents
  )
}
```

(a) Mapping internal transitions

```
lazy rule HiLLSTrans2DEVSdeltaExt {
  from --ExternalTransition-> DeltaExt
    outTrans: HiLLS!ExternalTransition
  to
    deltaInt : DEVS!DeltaExt (
    sourcePhase <-  outTrans.source,
    targetPhase <-  outTrans.target,
    inputs <- outTrans.triggers
  )
}
```

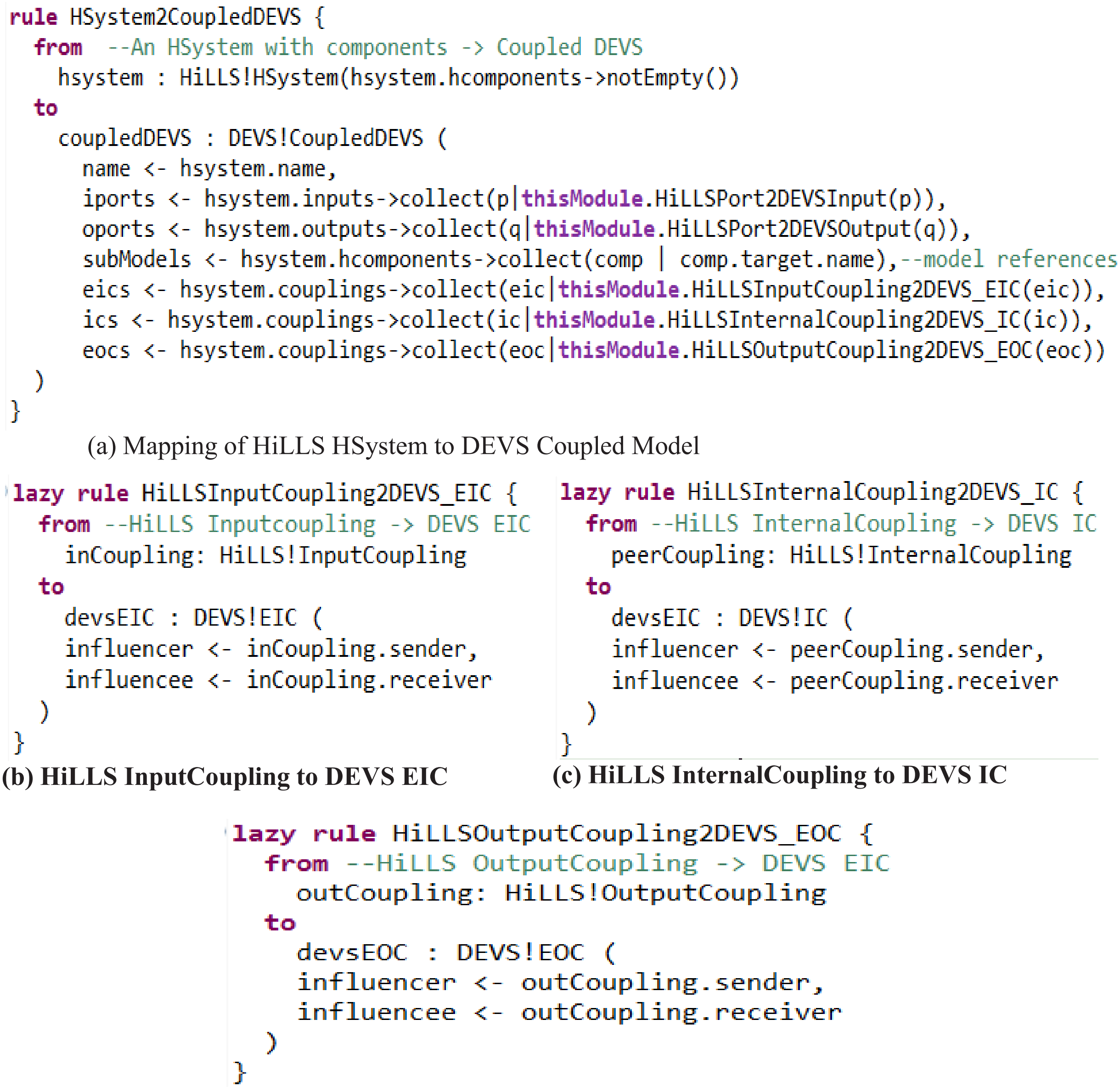
(b) Mapping external transitions

```
lazy rule HiLLSTrans2DEVSdeltaConf {
  from --ConfluentTransition-> DeltaConf
    confTrans: HiLLS!ConfluentTransition
  to
    deltaInt : DEVS!DeltaConf (
    sourcePhase <- confTrans.source,
    targetPhase <- confTrans.target,
    inputs <- confTrans.triggers,
    outputs <- confTrans.outputEvents
  )
}
```

(c) Mapping confluent transitions

Figure7.4Mapping HiLLS configuration transitions to DEVS phase transitions [AMT16]

In Figure 7.5, we show the correspondences between a*HSystem* with a nonempty *hComponents* and *CoupledDEVS* concepts. While Figure 7.5(a) provides the rules to obtain the different sets, Figure 7.5(b - d) show the rules for obtaining individual EIC, IC and EOC, respectively. The rules for obtaining individual input and output ports have been presented previously in Figure 7.3(b) and Figure 7.3(c).



```
rule HSystem2CoupledDEVS {
  from  --An HSystem with components -> Coupled DEVS
    hsystem : HiLLS!HSystem(hsystem.hcomponents->notEmpty())
  to
    coupledDEVS : DEVS!CoupledDEVS (
      name <- hsystem.name,
      iports <- hsystem.inputs->collect(p|thisModule.HiLLSPort2DEVSInput(p)),
      oports <- hsystem.outputs->collect(q|thisModule.HiLLSPort2DEVSOutput(q)),
      subModels <- hsystem.hcomponents->collect(comp | comp.target.name),--model references
      eics <- hsystem.couplings->collect(eic|thisModule.HiLLSInputCoupling2DEVS_EIC(eic)),
      ics <- hsystem.couplings->collect(ic|thisModule.HiLLSInternalCoupling2DEVS_IC(ic)),
      eocs <- hsystem.couplings->collect(eoc|thisModule.HiLLSOutputCoupling2DEVS_EOC(eoc))
  )
}
```

(a) Mapping of HiLLS HSystem to DEVS Coupled Model

```
lazy rule HiLLSInputCoupling2DEVS_EIC {
  from --HiLLS Inputcoupling -> DEVS EIC
    inCoupling: HiLLS!InputCoupling
  to
    devsEIC : DEVS!EIC (
    influencer <- inCoupling.sender,
    influencee <- inCoupling.receiver
  )
}
```

(b) HiLLS InputCoupling to DEVS EIC

```
lazy rule HiLLSInternalCoupling2DEVS_IC {
  from --HiLLS InternalCoupling -> DEVS IC
    peerCoupling: HiLLS!InternalCoupling
  to
    devsEIC : DEVS!IC (
    influencer <- peerCoupling.sender,
    influencee <- peerCoupling.receiver
  )
}
```

(c) HiLLS InternalCoupling to DEVS IC

```
lazy rule HiLLSOutputCoupling2DEVS_EOC {
  from --HiLLS OutputCoupling -> DEVS EIC
    outCoupling: HiLLS!OutputCoupling
  to
    devsEOC : DEVS!EOC (
    influencer <- outCoupling.sender,
    influencee <- outCoupling.receiver
  )
}
```

(d) HiLLS OutputCoupling to DEVS EOC

Figure 7.5Mapping rules of HiLLS concepts to Coupled DEVS concepts part [AMT16]

7.2.3 Correspondences between the HiLLS and DEVS Specifications of the BVS

As a proof of concept, we place the HiLLS and DEVS specifications of the BVS case study to illustrate how the different elements of a DEVS specification can be derived from a given HiLLS model.

We begin with the models of BVM in Figure 7.6below; we use arrows to link elements of the HiLLS model (on the left) with their corresponding constructs in the DEVS specification (on the right). The input (X) and output (Y) sets of the DEVS specification can be derived from the input and output interfaces respectively of the HiLLS model.

The light blue arrows show that DEVS' state set (S) is derived from the state schema and configurations of the HiLLS model. The declarations in the state schema, the containment references to HClasses (if any) and the labels of the configurations produce the variables in S while its (S's) predicates are derived from the conjunction of the predicates in the HiLLS state schema and the properties of the configurations. DEVS' *ta*() function is derived by scanning the sojourn times of all configurations. In this case, only *charge* and *dispense* have explicitly defined *sojournTime*, all other configurations have implicit values of *sojournTime*.

The red arrows link the HiLLS transitions to their corresponding specifications in the DEVS model; we use solid, dashed and dotted-dashed arrows for internal, external and confluent transitions respectively. For instance, the internal transition between configurations *reject* (source) and *charge* (target) produces the equation $charge\colon \langle badC = null\rangle, if\ \phi = reject$ in the δ_{int} function in the DEVS specification.

Each of the four paths of the external transition triggered by [inC?x] from configuration *charge* constitutes an equation in the δ_{ext} function of the DEVS specification. For example, the path that targets configuration *reject* produces the equation $reject\colon \langle badC\ = in\rangle,\ if\ \phi = charge \wedge p = inC \wedge v(in) = 0$ in the function. In this case, the components of predicate $\phi = charge \wedge p = inC \wedge v(in) = 0$ are derived as follows: $\phi = charge$ is equivalent to the *source* of the transition, $p = inC$ is he associated port of the trigger and $v(in) = 0$ is equivalent to the test condition *valid*(x) in the HiLLS transition. The equivalent constructs of the confluent transitions can be read in similar manners with reference to the δ_{conf} function in the DEVS specification.

Finally, we can extract the output operations associated with the configuration transitions to build the DEVS λ function. In each case, the *source* configuration of the associated transition is read along with the output operation. For instance, the output operation **outC!<badC>;** preceding the internal transition from *reject* to *charge* yields the equation $outC = \{badC\}$, $if\ \phi = reject$ in the DEVS λ function.

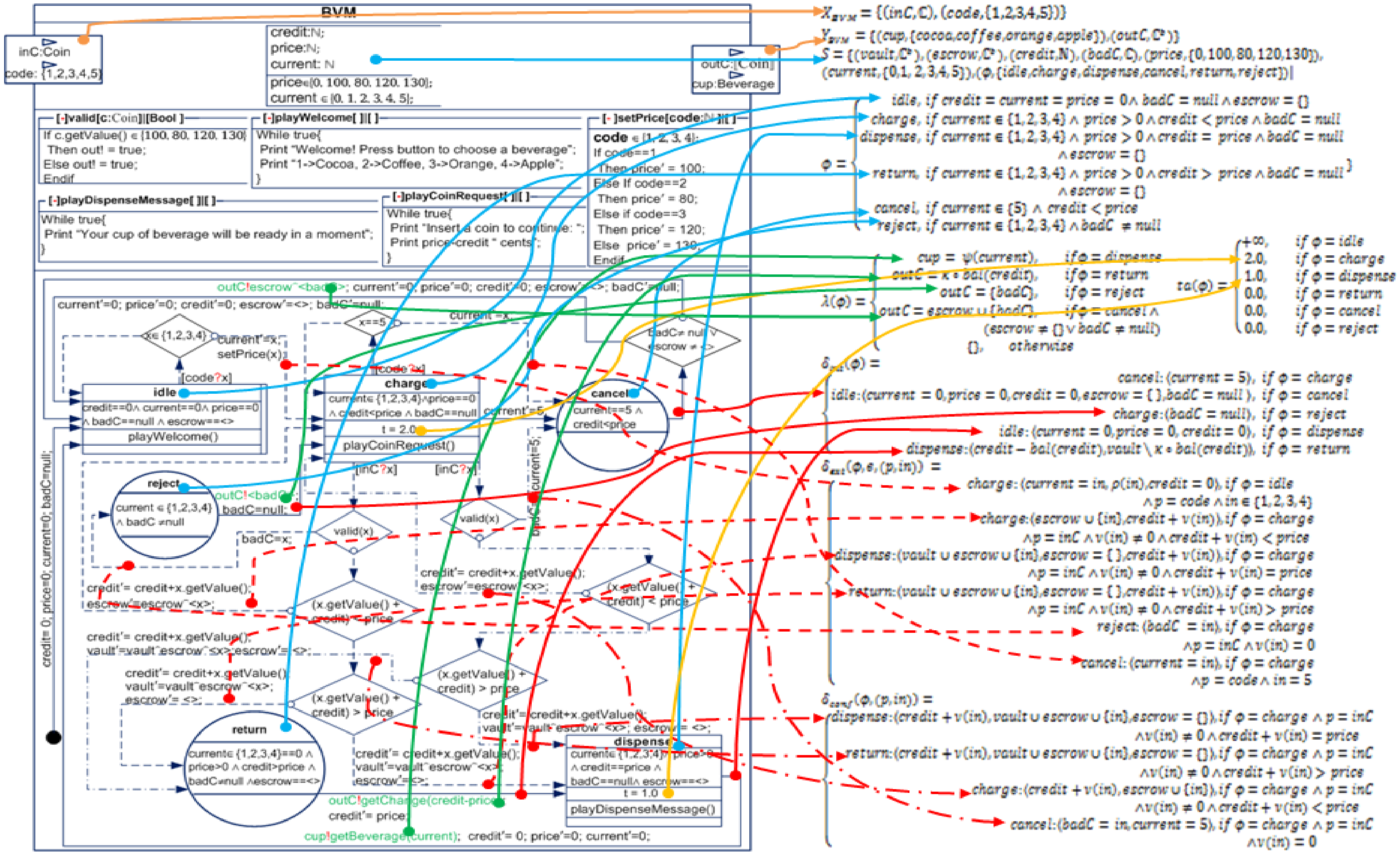
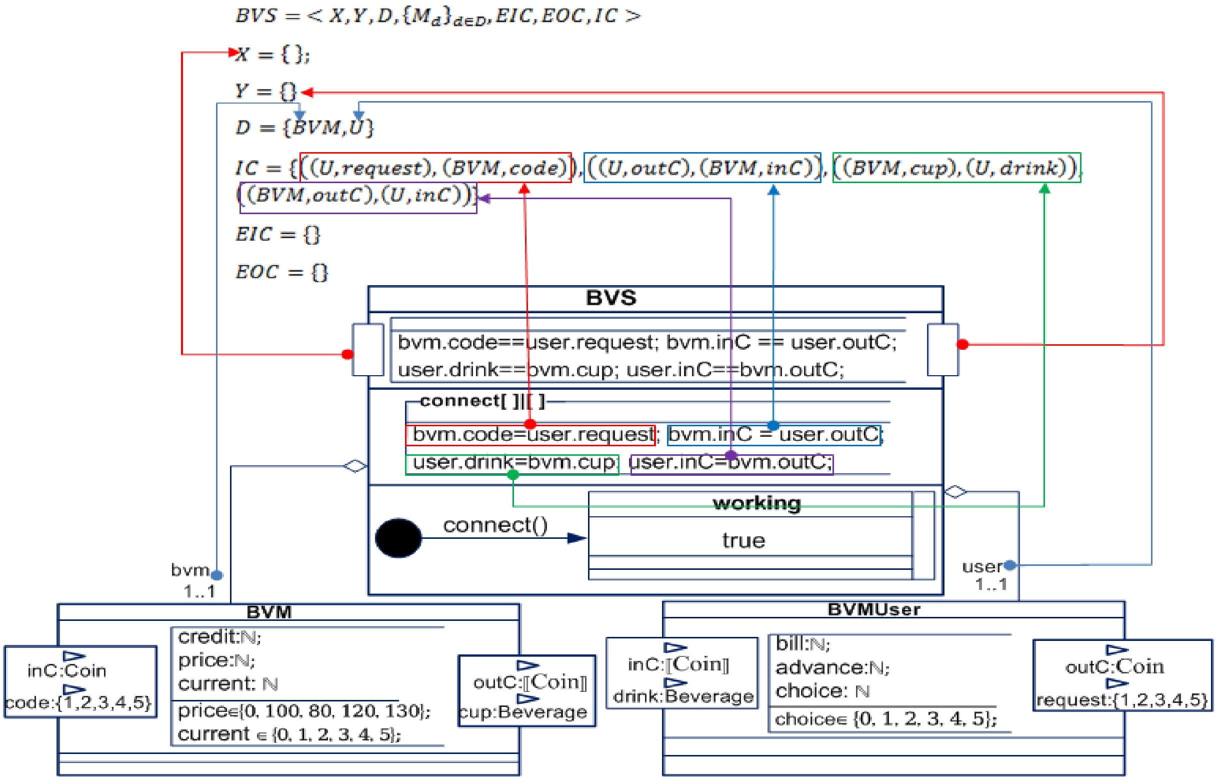

BVM
inC:Coin
code: {1,2,3,4,5}
credit:N;
price:N;
current: N
price∈{0, 100, 80, 120, 130};
current ∈ {0, 1, 2, 3, 4, 5};
outC:[Coin]
cup:Beverage
[-]valid[c:Coin][Bool]
If c.getValue() ∈ {100, 80, 120, 130}
Then out! = true;
Else out! = true;
Endif
[-]playWelcome[ ][ ]
While true{
Print "Welcome! Press button to choose a beverage";
Print "1->Cocoa, 2->Coffee, 3->Orange, 4->Apple";
}
[-]setPrice[code:N][ ]
code ∈ {1, 2, 3, 4};
If code==1
Then price' = 100;
Else If code==2
Then price' = 80;
Else if code==3
Then price' = 120;
Else price' = 130;
Endif
[-]playDispenseMessage[ ][ ]
While true{
Print "Your cup of beverage will be ready in a moment";
}
[-]playCoinRequest[ ][ ]
While true{
Print "Insert a coin to continue: ";
Print price-credit " cents";
}
idle
credit==0∧ current==0∧ price==0 ∧ badC==null ∧ escrow==<>
playWelcome()
charge
current∈ {1,2,3,4}∧price==0 ∧ credit<price ∧ badC==null
t = 2.0
playCoinRequest()
cancel
current==5 ∧ credit<price
reject
current ∈ {1,2,3,4} ∧ badC ≠null
return
current∈ {1,2,3,4}==0 ∧ price>0 ∧ credit>price ∧ badC≠null ∧escrow==<>
dispense
current∈ {1,2,3,4}∧price>0 ∧ credit==price ∧ badC==null∧ escrow==<>
t = 1.0
playDispenseMessage()
credit= 0; price=0; current=0; badC=null;
outC!escrow^<badC>; current'=0; price'=0; credit'=0; escrow'=<>; badC'=null;
current'=0; price'=0; credit'=0; escrow'=<>; badC'=null;
x∈ {1,2,3,4}
current'=x;
setPrice(x);
[code?x]
x==5
current'=x;
[code?x]
current'=5;
badC≠ null ∨ escrow ≠ <>
outC!<badC>;
badC=null;
[inC?x]
valid(x)
badC=x;
credit'= credit+x.getValue();
escrow'=escrow^<x>;
(x.getValue() + credit) &lt; price
credit'= credit+x.getValue();
vault'=vault^escrow^<x>;escrow'= <>;
(x.getValue() + credit) > price
(x.getValue() + credit) = price
outC!getCharge(credit-price);
credit'= price;
cup!getBeverage(current); credit'= 0; price'=0; current'=0;
$X_{BVM} = \{(inC,\mathbb{C}),(code,\{1,2,3,4,5\})\}$
$Y_{BVM} = \{(cup,\{cocoa,coffee,orange,apple\}),(outC,\mathbb{C}^*)\}$
$S = \{(vault,\mathbb{C}^*),(escrow,\mathbb{C}^*),(credit,\mathbb{N}),(badC,\mathbb{C}),(price,\{0,100,80,120,130\}),(current,\{0,1,2,3,4,5\}),(\phi,\{idle,charge,dispense,cancel,return,reject\})\}$
$\phi =$
$idle$, if $credit = current = price = 0 \wedge badC = null \wedge escrow = \{\}$
$charge$, if $current \in \{1,2,3,4\} \wedge price > 0 \wedge credit < price \wedge badC = null$
$dispense$, if $current \in \{1,2,3,4\} \wedge price > 0 \wedge credit = price \wedge badC = null \wedge escrow = \{\}$
$return$, if $current \in \{1,2,3,4\} \wedge price > 0 \wedge credit > price \wedge badC = null \wedge escrow = \{\}$
$cancel$, if $current \in \{5\} \wedge credit < price$
$reject$, if $current \in \{1,2,3,4\} \wedge badC \neq null$
$\lambda(\phi) =$
$cup = \psi(current)$, if $\phi = dispense$
$outC = x \bullet bal(credit)$, if $\phi = return$
$outC = \{badC\}$, if $\phi = reject$
$outC = escrow \cup \{badC\}$, if $\phi = cancel \wedge (escrow \neq \{\} \vee badC \neq null)$
$\{\}$, otherwise
$ta(\phi) =$
$+\infty$, if $\phi = idle$
2.0, if $\phi = charge$
1.0, if $\phi = dispense$
0.0, if $\phi = return$
0.0, if $\phi = cancel$
0.0, if $\phi = reject$
$\delta_{int}(\phi) =$
$cancel: (current = 5)$, if $\phi = charge$
$idle: (current = 0, price = 0, credit = 0, escrow = \{\}, badC = null)$, if $\phi = cancel$
$charge: (badC = null)$, if $\phi = reject$
$idle: (current = 0, price = 0, credit = 0)$, if $\phi = dispense$
$dispense: (credit - bal(credit), vault \setminus x \bullet bal(credit))$, if $\phi = return$
$\delta_{ext}(\phi, e, (p, in)) =$
$charge: (current = in, p(in), credit = 0)$, if $\phi = idle \wedge p = code \wedge in \in \{1,2,3,4\}$
$charge: (escrow \cup \{in\}, credit + v(in))$, if $\phi = charge \wedge p = inC \wedge v(in) \neq 0 \wedge credit + v(in) < price$
$dispense: (vault \cup escrow \cup \{in\}, escrow = \{\}, credit + v(in))$, if $\phi = charge \wedge p = inC \wedge v(in) \neq 0 \wedge credit + v(in) = price$
$return: (vault \cup escrow \cup \{in\}, escrow = \{\}, credit + v(in))$, if $\phi = charge \wedge p = inC \wedge v(in) \neq 0 \wedge credit + v(in) > price$
$reject: (badC = in)$, if $\phi = charge \wedge p = inC \wedge v(in) = 0$
$cancel: (current = in)$, if $\phi = charge \wedge p = code \wedge in = 5$
$\delta_{conf}(\phi, (p, in)) =$
$dispense: (credit + v(in), vault \cup escrow \cup \{in\}, escrow = \{\})$, if $\phi = charge \wedge p = inC \wedge v(in) \neq 0 \wedge credit + v(in) = price$
$return: (credit + v(in), vault \cup escrow \cup \{in\}, escrow = \{\})$, if $\phi = charge \wedge p = inC \wedge v(in) \neq 0 \wedge credit + v(in) > price$
$charge: (credit + v(in), escrow \cup \{in\})$, if $\phi = charge \wedge p = inC \wedge v(in) \neq 0 \wedge credit + v(in) < price$
$cancel: (badC = in, current = 5)$, if $\phi = charge \wedge p = inC \wedge v(in) = 0$

Figure 7.6 Correspondences between HiLLS and DEVS models of BVM

The correspondences described in Figure 7.6above typify the synthesis of a DEVS atomic model from a HiLLS' HSystem that has no components. Figure 7.7 illustrates how a coupled DEVS model can be derived from a composite HiLLS HSystem (one with hComponent references). The DEVS specification of BVS is shown on top of the diagram with the HiLLS model at the bottom. The empty input and output interfaces of the HiLLS model correspond to the empty sets X and Y respectively.

We can derive the set *D* in the DEVS specification from the *hComponent* references *bvm* and *user* of BVM in the HiLLS model. Finally, the DEVS coupling relations can be obtained from the coupling expressions defined in the *connect* operation. For example, the expression $bvm.code = user.request$ translates into the pair $((U, request), (BVM, code))$ in the DEVS IC relations. Note that *U* is the name of the DEVS model of the BVM's user in the DEVS specification presented in Section 3.2.3.2.



$BVS = < X, Y, D, \{M_d\}_{d \in D}, EIC, EOC, IC >$
$X = \{\};$
$Y = \{\}$
$D = \{BVM, U\}$
$IC = \{((U, request), (BVM, code)), ((U, outC), (BVM, inC)), ((BVM, cup), (U, drink)), ((BVM, outC), (U, inC))\}$
$EIC = \{\}$
$EOC = \{\}$
BVS
bvm.code==user.request; bvm.inC == user.outC;
user.drink==bvm.cup; user.inC==bvm.outC;
connect[ ]|[ ]
bvm.code=user.request; bvm.inC = user.outC;
user.drink=bvm.cup; user.inC=bvm.outC;
connect()
working
true
bvm
1..1
user
1..1
BVM
inC:Coin
code:{1,2,3,4,5}
credit:ℕ;
price:ℕ;
current: ℕ
price∈{0, 100, 80, 120, 130};
current ∈ {0, 1, 2, 3, 4, 5};
outC:⟦Coin⟧
cup:Beverage
BVMUser
inC:⟦Coin⟧
drink:Beverage
bill:ℕ;
advance:ℕ;
choice: ℕ
choice∈ {0, 1, 2, 3, 4, 5};
outC:Coin
request:{1,2,3,4,5}

Figure 7.7Correspondences between HiLLS and DEVS models of BVS

7.3 LOGICAL SEMANTICS

In this section, we present the mapping rules to translate a HiLLS system models to Z specification for formal analysis. To begin with, we present a metamodel that describes the essential elements of the Z formalism, which are most required for this translation.

7.3.1 Z Metamodel

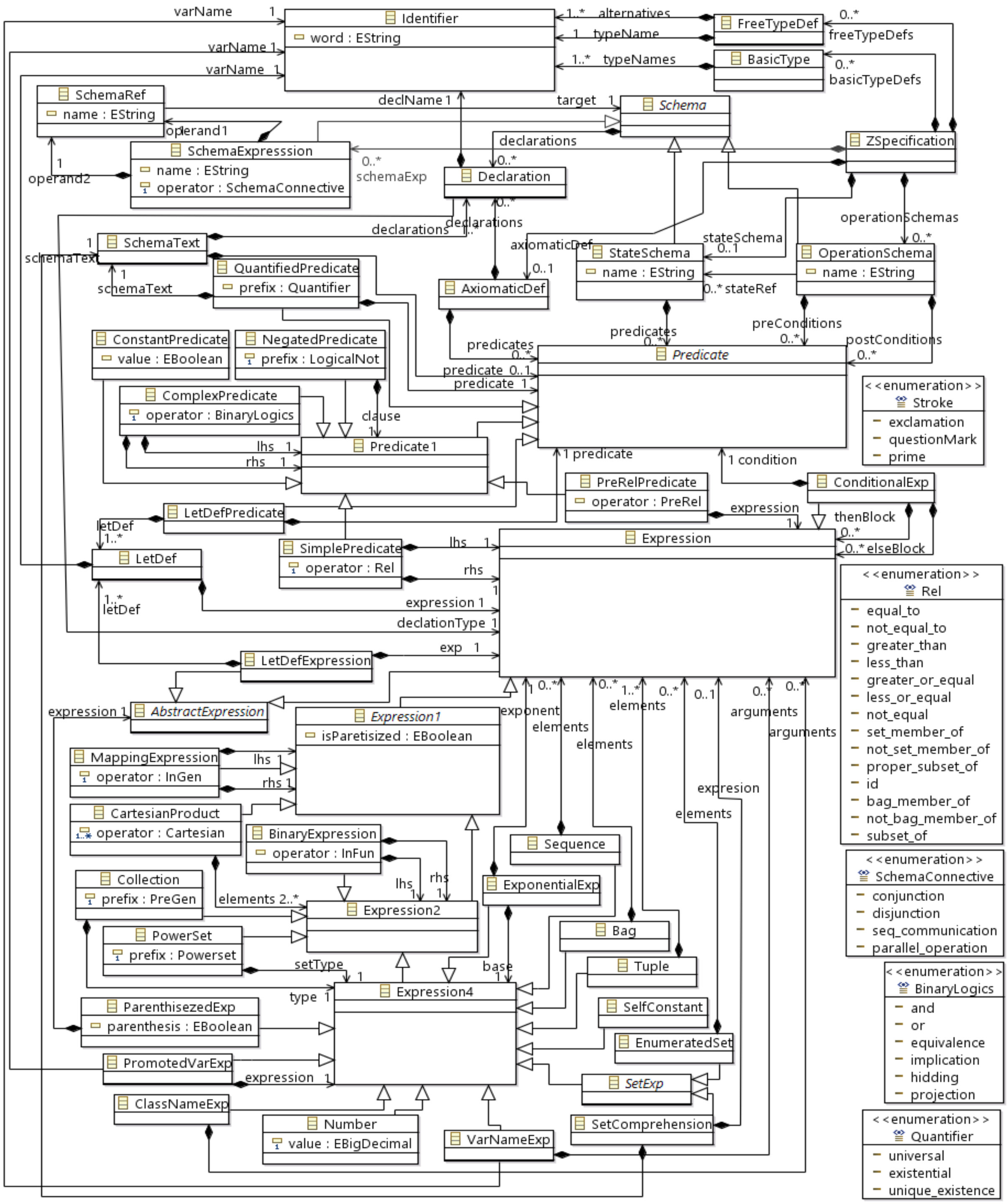


Figure 7.8 Simplified Z metamodel

Figure 7.8presents a simplified metamodel extracted from the grammar-based specification of Z's syntax in [Spi92]. It is simplified in the sense that it does not capture the entire language's vocabulary; rather, it describes the Z concepts that are essential for the presentation of the translation of HiLLS to Z.

As described previously in Section3.2.4, a Z specification is made up of *paragraphs*. A paragraph can be a *basic type definition*, a *free type definition*, an *axiomatic definition*, a *stateschema*, an *operationschema*. In addition, using *schema calculus*, we can combine schemas for hierarchical construction of complex schemas from simpler ones. These minimal concepts are described in Figure 7.8.

Essentially, a state schema consists of zero or more declarations and zero or more predicates that define constraints on the declared variables. Each declaration has a name, declName, which is an identifier, and a type, *declarationType*, which is defined by an expression. *declName* consists of a unique *word* and an optional *decoration*. A decoration is a special character at the end of a variable name to provide certain information about the purpose of the variable or its status at different moments. There are three kinds of decoration as described in enumeration *Stroke*: the question (?) indicates that a variable serves as an input in an operation schema, exclamation (!) denotes an output variable while the prime ($'$) decoration denotes the post-execution value of the variable. An axiomatic definition is similar to the state schema except that it does not have a name.

In an operation schema, each predicate serves one of two purposes: *pre-condition* and *post-condition*. A pre-condition specifies the constraints that must be satisfied by certain variables before the operation can be executed while a post-condition specifies a constraint that must be satisfied after the execution.

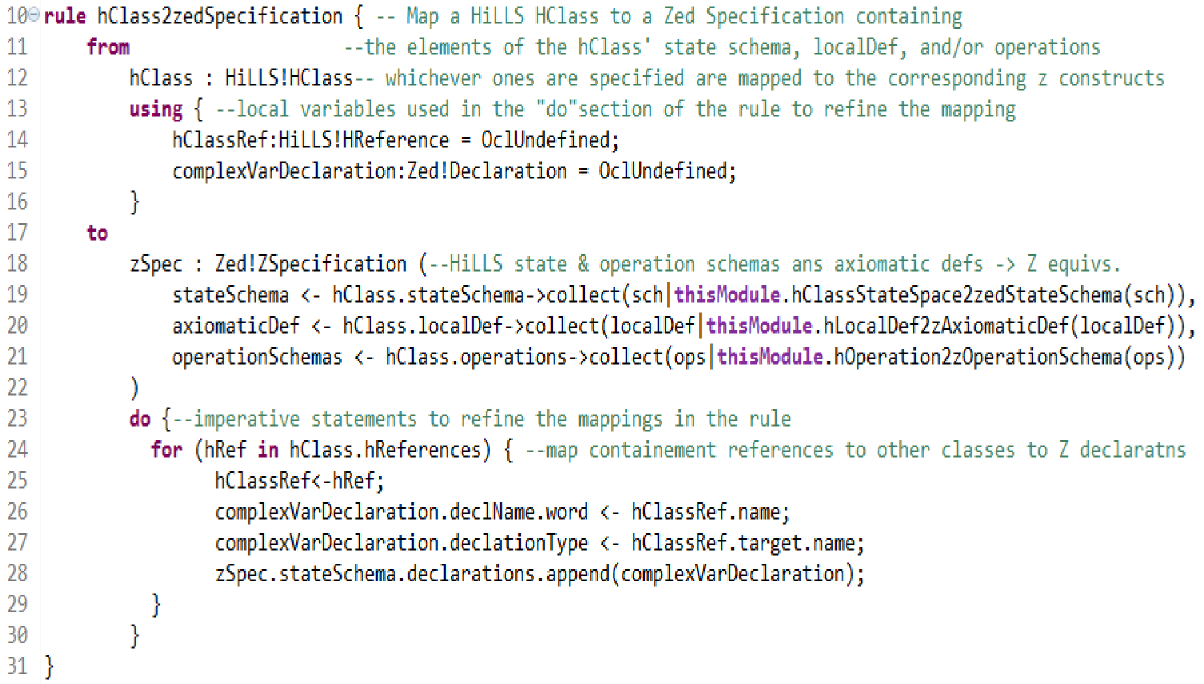
7.3.2 HiLLS to Z Mapping

In this subsection, we present the mapping rules to map HiLLS concepts to Z concepts. Like the HiLLS-to-DEVS mapping rules presented in the previous section, the HiLLS-to-Z mapping rules are specified using the ATL.

7.3.2.1 Mapping HClass to Z specification

Figure 7.9 presents the top-level transformation rule to translate an instance *hClass* of HClass in HiLLS to a Z specification. In the optional "using" section, we define two local variables, which are used in subsequent sections of the rule. The "to" section specifies that *hClass* is maps to an instance *zSpec* of *ZSpecification* in the Zed. The *stateSchema*, *localDef* and *operations* of *hClass* map respectively to a *stateSchema*, an *axiomaticDef* and *operations* in *zSpec*.

Each of the mappings is achieved by calling the rules that map the individual elements; the called rules will be presented subsequently.



```
10 rule hClass2zedSpecification { -- Map a HiLLS HClass to a Zed Specification containing
11     from                    --the elements of the hClass' state schema, localDef, and/or operations
12         hClass : HiLLS!HClass-- whichever ones are specified are mapped to the corresponding z constructs
13         using { --local variables used in the "do"section of the rule to refine the mapping
14             hClassRef:HiLLS!HReference = OclUndefined;
15             complexVarDeclaration:Zed!Declaration = OclUndefined;
16         }
17     to
18         zSpec : Zed!ZSpecification (--HiLLS state & operation schemas ans axiomatic defs -> Z equivs.
19             stateSchema <- hClass.stateSchema->collect(sch|thisModule.hClassStateSpace2zedStateSchema(sch)),
20             axiomaticDef <- hClass.localDef->collect(localDef|thisModule.hLocalDef2zAxiomaticDef(localDef)),
21             operationSchemas <- hClass.operations->collect(ops|thisModule.hOperation2zOperationSchema(ops))
22         )
23         do {--imperative statements to refine the mappings in the rule
24           for (hRef in hClass.hReferences) { --map containement references to other classes to Z declaratns
25                 hClassRef<-hRef;
26                 complexVarDeclaration.declName.word <- hClassRef.name;
27                 complexVarDeclaration.declationType <- hClassRef.target.name;
28                 zSpec.stateSchema.declarations.append(complexVarDeclaration);
29           }
30         }
31 }
```

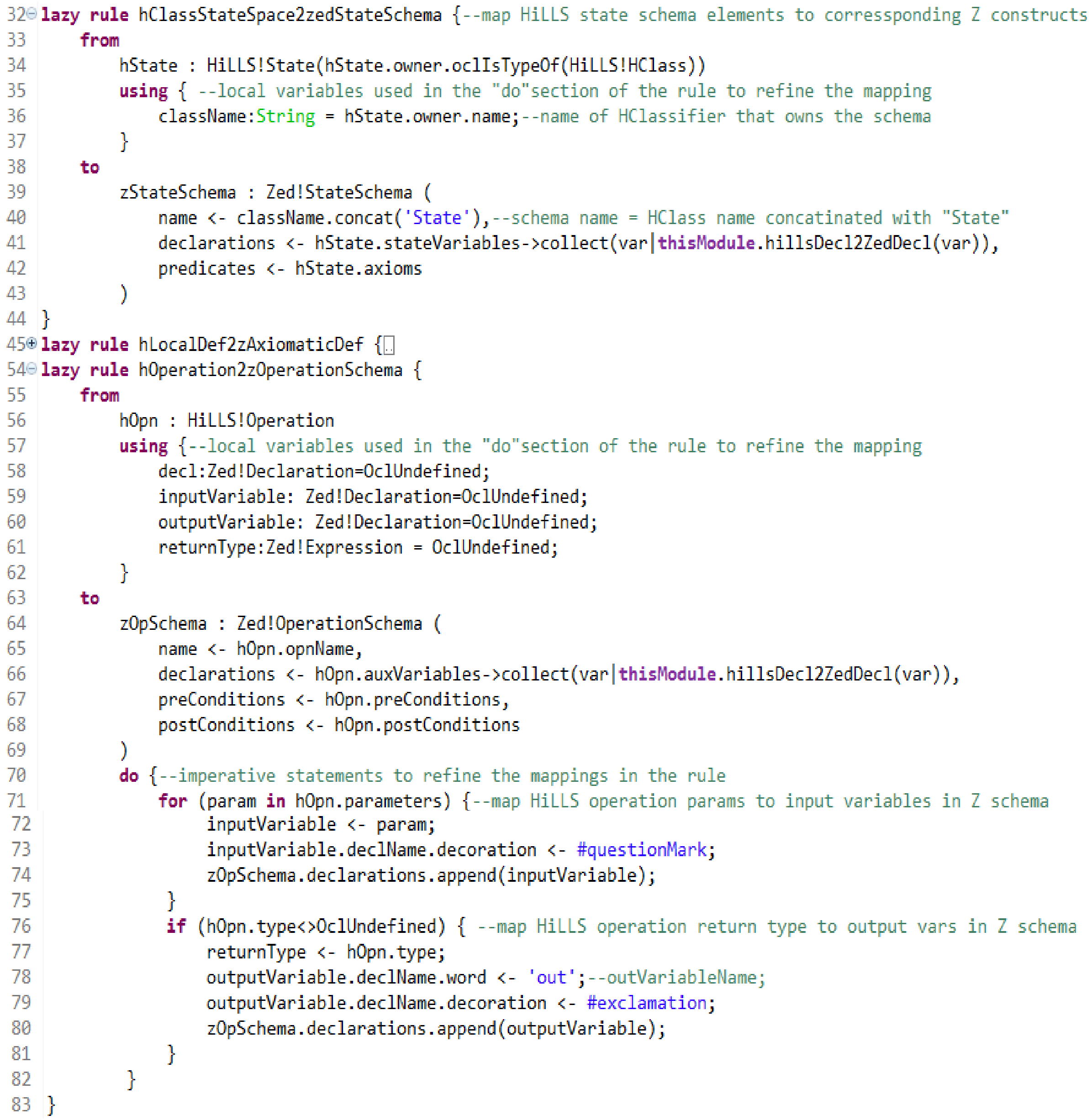
Figure 7.9 HiLLS HClass to Z specification

In the "do" section, we specify some imperative statements to refine the translation specified in the "to" section. Particularly, each composition reference from *hClass* to another HClass is translated into a Z declaration with the reference's name and the name of the referenced HClass mapping to the Z declaration's name and type respectively (lines 24-27). Then, in line 28, we add the new declaration to the list of declarations in *zSpec*'s state schema.

7.3.2.2 Mapping HClass'state schemas and operations to Z schemas

Figure 7.10 (lines 32-44) presents the mapping rules to translate the state schema *hState* of a given HClass to a Z state schema. Recall that in the HiLLS metamodel, the concept of state schema is shared between HClass and HSystem through HClassifier. The precondition "*hState.owner.oclIsTypeOf(HiLLS!HClass)*" in line 34 specifies that this rule applies only to a HiLLS state schema that is owned by an HClass.

The local variable *className* declared in the "using" section holds the name of the HClass that owns *hState*. This is used in line 40 to define the name of the *zStateSchema* created from *hState*. In a direct translation, all declarations and predicates in the source are mapped, respectively, to declarations and predicates in the target. Each declaration mapping calls the rule that constructs the target declaration. Expectedly, a similar rule should exist for the predicates; this requires the details of the structures of all predicates. It will be specified in our future work on the mapping. We present the mapping of an operation, *hOpn*, in HiLLS to a Z operation schema, *zOpSchema*, in Figure 7.10 (lines 54-83).

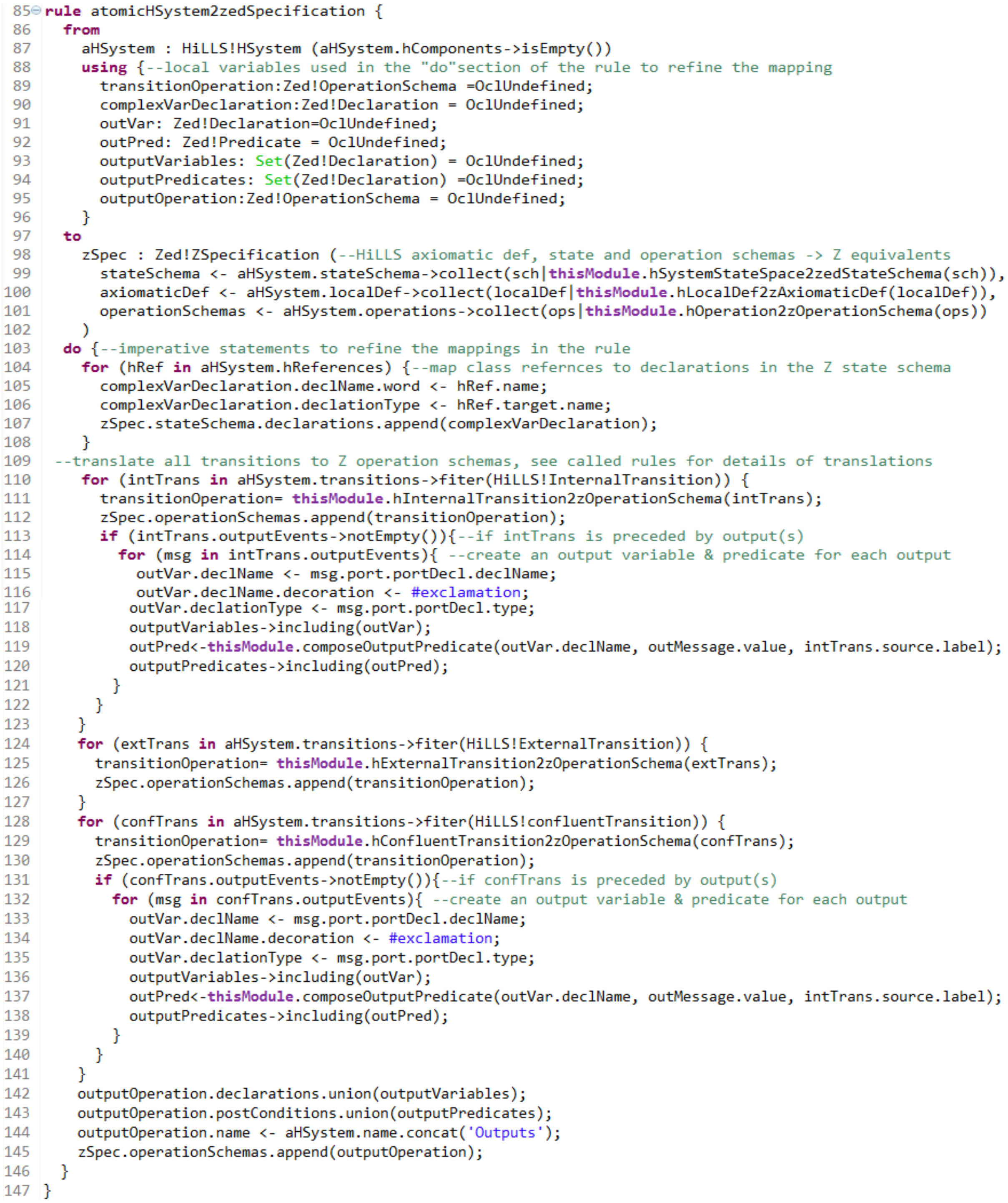


```
32 lazy rule hClassStateSpace2zedStateSchema {--map HiLLS state schema elements to corressponding Z constructs
33     from
34         hState : HiLLS!State(hState.owner.oclIsTypeOf(HiLLS!HClass))
35         using { --local variables used in the "do"section of the rule to refine the mapping
36             className:String = hState.owner.name;--name of HClassifier that owns the schema
37         }
38     to
39         zStateSchema : Zed!StateSchema (
40             name <- className.concat('State'),--schema name = HClass name concatinated with "State"
41             declarations <- hState.stateVariables->collect(var|thisModule.hillsDecl2ZedDecl(var)),
42             predicates <- hState.axioms
43         )
44 }
45 lazy rule hLocalDef2zAxiomaticDef {
54 lazy rule hOperation2zOperationSchema {
55     from
56         hOpn : HiLLS!Operation
57         using {--local variables used in the "do"section of the rule to refine the mapping
58             decl:Zed!Declaration=OclUndefined;
59             inputVariable: Zed!Declaration=OclUndefined;
60             outputVariable: Zed!Declaration=OclUndefined;
61             returnType:Zed!Expression = OclUndefined;
62         }
63     to
64         zOpSchema : Zed!OperationSchema (
65             name <- hOpn.opnName,
66             declarations <- hOpn.auxVariables->collect(var|thisModule.hillsDecl2ZedDecl(var)),
67             preConditions <- hOpn.preConditions,
68             postConditions <- hOpn.postConditions
69         )
70         do {--imperative statements to refine the mappings in the rule
71             for (param in hOpn.parameters) {--map HiLLS operation params to input variables in Z schema
72                 inputVariable <- param;
73                 inputVariable.declName.decoration <- #questionMark;
74                 zOpSchema.declarations.append(inputVariable);
75              }
76              if (hOpn.type<>OclUndefined) { --map HiLLS operation return type to output vars in Z schema
77                 returnType <- hOpn.type;
78                 outputVariable.declName.word <- 'out';--outVariableName;
79                 outputVariable.declName.decoration <- #exclamation;
80                 zOpSchema.declarations.append(outputVariable);
81              }
82          }
83 }
```

Figure 7.10 Mapping rules for translating HiLLS HClass' state schema to Z state schema and HiLLS operation to Z operation schema

As declared in the "to" section, the name, variables, pre-conditions and post-conditions in the source are translated respectively to the same concepts in the target model. The target model so generated in "to" section is further refined in the "do" section. In lines 71-75, each parameter (if any) of *hOpn* translates to an input variable (with decoration "?") in the target model. Similarly, if *hOpn* defines a return type, it is translated (lines 76-82) an output variable (with decoration "!") in the target model.

7.3.2.3 *Mapping HSystem to Z specification*



```
rule atomicHSystem2zedSpecification {
  from
    aHSystem : HiLLS!HSystem (aHSystem.hComponents->isEmpty())
    using {--local variables used in the "do"section of the rule to refine the mapping
      transitionOperation:Zed!OperationSchema =OclUndefined;
      complexVarDeclaration:Zed!Declaration = OclUndefined;
      outVar: Zed!Declaration=OclUndefined;
      outPred: Zed!Predicate = OclUndefined;
      outputVariables: Set(Zed!Declaration) = OclUndefined;
      outputPredicates: Set(Zed!Declaration) =OclUndefined;
      outputOperation:Zed!OperationSchema = OclUndefined;
    }
  to
    zSpec : Zed!ZSpecification (--HiLLS axiomatic def, state and operation schemas -> Z equivalents
      stateSchema <- aHSystem.stateSchema->collect(sch|thisModule.hSystemStateSpace2zedStateSchema(sch)),
      axiomaticDef <- aHSystem.localDef->collect(localDef|thisModule.hLocalDef2zAxiomaticDef(localDef)),
      operationSchemas <- aHSystem.operations->collect(ops|thisModule.hOperation2zOperationSchema(ops))
    )
  do {--imperative statements to refine the mappings in the rule
    for (hRef in aHSystem.hReferences) {--map class refernces to declarations in the Z state schema
      complexVarDeclaration.declName.word <- hRef.name;
      complexVarDeclaration.declationType <- hRef.target.name;
      zSpec.stateSchema.declarations.append(complexVarDeclaration);
    }
 --translate all transitions to Z operation schemas, see called rules for details of translations
    for (intTrans in aHSystem.transitions->fiter(HiLLS!InternalTransition)) {
      transitionOperation= thisModule.hInternalTransition2zOperationSchema(intTrans);
      zSpec.operationSchemas.append(transitionOperation);
      if (intTrans.outputEvents->notEmpty()){--if intTrans is preceded by output(s)
        for (msg in intTrans.outputEvents){ --create an output variable & predicate for each output
          outVar.declName <- msg.port.portDecl.declName;
          outVar.declName.decoration <- #exclamation;
         outVar.declationType <- msg.port.portDecl.type;
         outputVariables->including(outVar);
         outPred<-thisModule.composeOutputPredicate(outVar.declName, outMessage.value, intTrans.source.label);
         outputPredicates->including(outPred);
        }
      }
    }
    for (extTrans in aHSystem.transitions->fiter(HiLLS!ExternalTransition)) {
      transitionOperation= thisModule.hExternalTransition2zOperationSchema(extTrans);
      zSpec.operationSchemas.append(transitionOperation);
    }
    for (confTrans in aHSystem.transitions->fiter(HiLLS!confluentTransition)) {
      transitionOperation= thisModule.hConfluentTransition2zOperationSchema(confTrans);
      zSpec.operationSchemas.append(transitionOperation);
      if (confTrans.outputEvents->notEmpty()){--if confTrans is preceded by output(s)
        for (msg in confTrans.outputEvents){ --create an output variable & predicate for each output
         outVar.declName <- msg.port.portDecl.declName;
         outVar.declName.decoration <- #exclamation;
         outVar.declationType <- msg.port.portDecl.type;
         outputVariables->including(outVar);
         outPred<-thisModule.composeOutputPredicate(outVar.declName, outMessage.value, intTrans.source.label);
         outputPredicates->including(outPred);
        }
      }
    }
    outputOperation.declarations.union(outputVariables);
    outputOperation.postConditions.union(outputPredicates);
    outputOperation.name <- aHSystem.name.concat('Outputs');
    zSpec.operationSchemas.append(outputOperation);
  }
}
```

Figure 7.11 Mapping rule for translating HiLLS atomic HSystem to Z specification

Figure 7.11 presents the mapping rule between an atomic HSystem *aHSystem* and its equivalent Z specification *zSpec*. Like in the case of HClass, the state space, local definitions and operations of *aHSystem* translate into Z state schema, axiomatic definition and operation schemas respectively in *zSpec*. While the mapping rules for local definitions and operations are the same for HClass and HSystem, the mapping rule, *hSystemStateSpace2zStateSchema* (line 99), for HSystem's state space is different from that of HClass. This rule will be presented in the next sub subsection.

The "do" section specifies the imperative statements to extract the data required for the synthesis of equivalent Z artifacts for three categories of HiLLS constructs as follows:

A. Lines 104-108 specify the translation of hreferences originating from *aHSystem* into Z declarations and add them (the synthesized declarations) to the declaration part of the state schema created for *zSpec*.

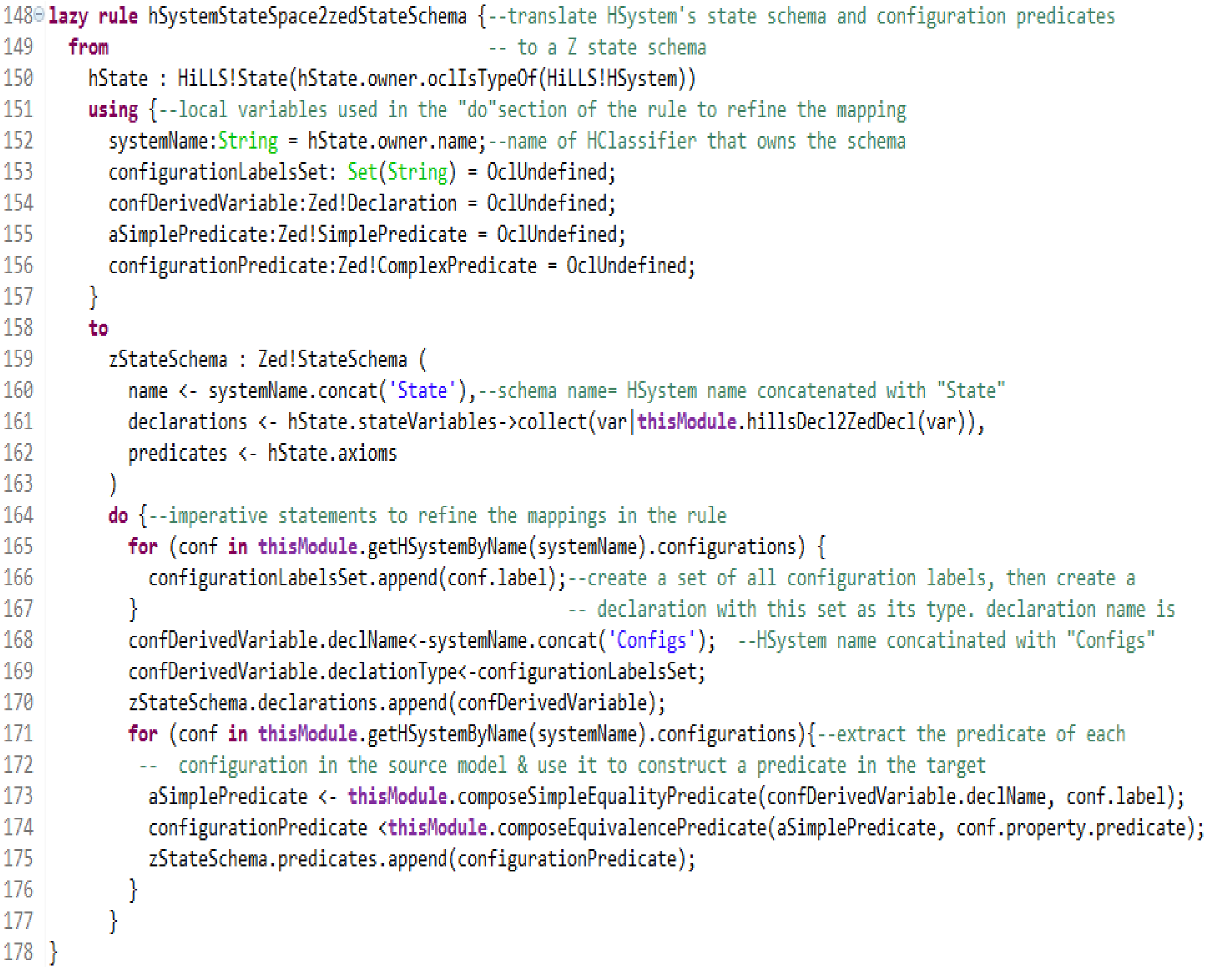
B. All internal, external and confluent configuration transitions in *aHSystem* are translated into Z operation schemas and added to the set of operation schemas in *zSpecs* within line groups 110-112, 124-127 and 128-130 respectively. In each case, a transition-to-operation schema mapping rule, e.g., *hInternalTransition2zOperationSchema*() in line 111, is called to do the translation. We will discuss these mapping rules in subsequent sub subsections.

C. In the process of translating internal and confluent configuration transitions to their equivalent operation schemas, if any of them is accompanied by an *output* specification (see lines 113-122 for internal transitions and 131-140 for confluent transitions), then an output variable and an output predicate are created and stored in sets *outputVariables* and *outputPredicates* respectively. Sets *outputVariables* and *outputPredicates* have been declared as local variables in the "using" section. Before exiting the "do" section, the two sets are used to build an operation schema, which is added to the set of operation schemas in *zSpec* as specified in lines 142-145.

7.3.2.4 Mapping HSystem's state space to Z schemas

We present, in Figure 7.12, the mapping rule to translate the state space, *hState*, of a HiLLS HSystem to a state schema, *zStateSchema*, in Z. As the "to" section (lines 158-163) describes, the name attribute of the HSystem that owns *hState* is extracted and concatenated with string "State" to provide a name for the target *zStateSchema*. As usual, the variable declarations and predicates in *hState* translate directly to declarations and predicates in *zStateSchema.*

However, the state space of a HSystem is jointly specified by its state schema and configuration diagrams. Hence, there is need to capture the state information specified by the configuration diagrams in the target *zStateSchema*; this aspect is specified in the "do" section of the rules (lines 164-177).



```
148 lazy rule hSystemStateSpace2zedStateSchema {--translate HSystem's state schema and configuration predicates
149   from                                      -- to a Z state schema
150     hState : HiLLS!State(hState.owner.oclIsTypeOf(HiLLS!HSystem))
151     using {--local variables used in the "do"section of the rule to refine the mapping
152       systemName:String = hState.owner.name;--name of HClassifier that owns the schema
153       configurationLabelsSet: Set(String) = OclUndefined;
154       confDerivedVariable:Zed!Declaration = OclUndefined;
155       aSimplePredicate:Zed!SimplePredicate = OclUndefined;
156       configurationPredicate:Zed!ComplexPredicate = OclUndefined;
157     }
158     to
159       zStateSchema : Zed!StateSchema (
160         name <- systemName.concat('State'),--schema name= HSystem name concatenated with "State"
161         declarations <- hState.stateVariables->collect(var|thisModule.hillsDecl2ZedDecl(var)),
162         predicates <- hState.axioms
163       )
164       do {--imperative statements to refine the mappings in the rule
165         for (conf in thisModule.getHSystemByName(systemName).configurations) {
166           configurationLabelsSet.append(conf.label);--create a set of all configuration labels, then create a
167         }                                         -- declaration with this set as its type. declaration name is
168         confDerivedVariable.declName<-systemName.concat('Configs');  --HSystem name concatinated with "Configs"
169         confDerivedVariable.declationType<-configurationLabelsSet;
170         zStateSchema.declarations.append(confDerivedVariable);
171         for (conf in thisModule.getHSystemByName(systemName).configurations){--extract the predicate of each
172          --  configuration in the source model & use it to construct a predicate in the target
173           aSimplePredicate <- thisModule.composeSimpleEqualityPredicate(confDerivedVariable.declName, conf.label);
174           configurationPredicate <thisModule.composeEquivalencePredicate(aSimplePredicate, conf.property.predicate);
175           zStateSchema.predicates.append(configurationPredicate);
176         }
177       }
178 }
```

Figure 7.12 Mapping rule to translate the state space of a HiLLS HSytem to a Z state aschema

First, the labels of all configurations are extracted into a set *configurationLabelSet* (lines 165-167), then a Z declaration is created with this set as its type and the HSystem's name concatenated with string "Configs" as the declaration name. This new declaration is added to the declaration part of *zStateSchema* (see lines 168-170). Finally, in line 171-176, the property of each of the configurations is extracted and used to construct a predicate that is added to the predicate part of *zStateSchema*.

To illustrate the meaning of this rule, we present the translation of the state space of the BVM's HiLLS model to its Z state schema (see Section 3.2.4.4) in Figure 7.13.

In Figure 7.13, we use arrows to match corresponding elements between the source and target model elements when the mapping rule presented in Figure 7.12 is applied to the BVM's HiLLS model.

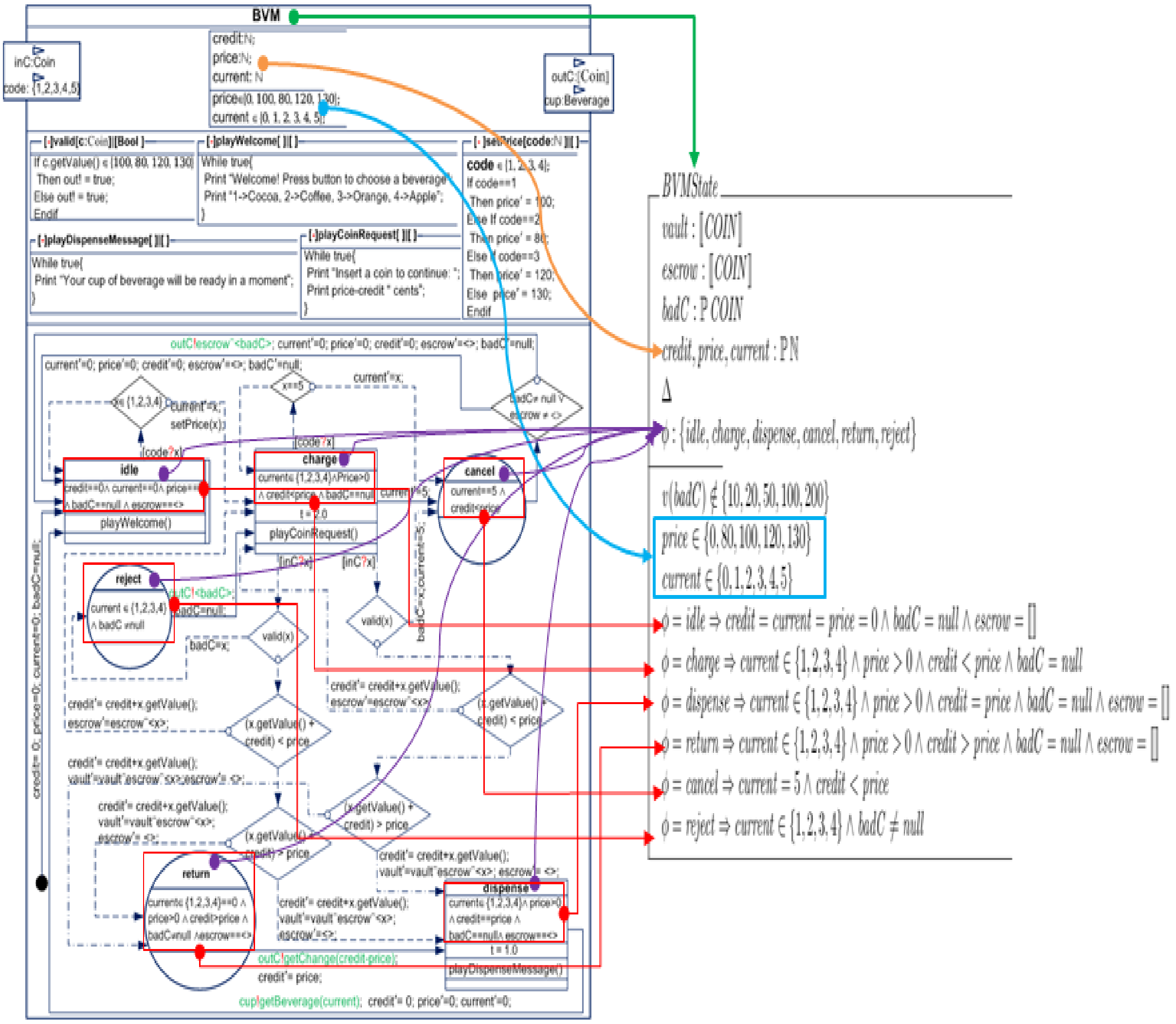
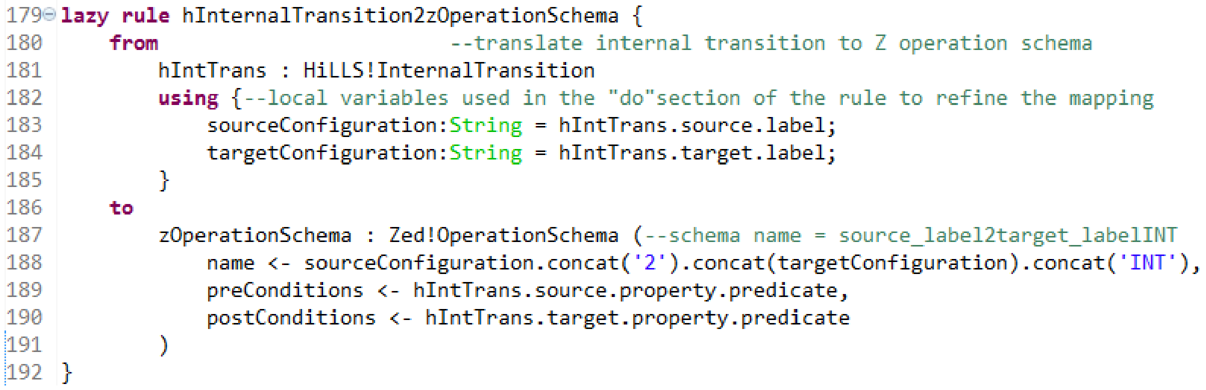

BVM
inC:Coin
code: {1,2,3,4,5}
credit:N;
price:N;
current: N
price∈{0, 100, 80, 120, 130};
current ∈{0, 1, 2, 3, 4, 5};
outC:[Coin]
cup:Beverage
[-]valid[c:Coin][Bool]
If c.getValue() ∈ {100, 80, 120, 130}
Then out! = true;
Else out! = true;
Endif
[-]playWelcome[ ][ ]
While true{
Print "Welcome! Press button to choose a beverage";
Print "1->Cocoa, 2->Coffee, 3->Orange, 4->Apple";
}
[-]setPrice[code:N][ ]
code ∈ {1, 2, 3, 4};
If code==1
Then price' = 100;
Else If code==2
Then price' = 80;
Else If code==3
Then price' = 120;
Else price' = 130;
Endif
[-]playDispenseMessage[ ][ ]
While true{
Print "Your cup of beverage will be ready in a moment";
}
[-]playCoinRequest[ ][ ]
While true{
Print "Insert a coin to continue: ";
Print price-credit " cents";
}
outC!escrow⌢<badC>; current'=0; price'=0; credit'=0; escrow'=<>; badC'=null;
current'=0; price'=0; credit'=0; escrow'=<>; badC'=null;
x∈{1,2,3,4}
current'=x;
setPrice(x);
x==5
current'=x;
badC≠ null ∨ escrow ≠ <>
[code?x]
[code?x]
idle
credit==0∧ current==0∧ price==0
∧ badC==null ∧ escrow==<>
playWelcome()
charge
current∈{1,2,3,4}∧Price>0
∧ credit<price ∧ badC==null
t = 2.0
playCoinRequest()
current'=5;
cancel
current==5 ∧
credit<price
[inC?x]
[inC?x]
reject
current ∈ {1,2,3,4}
∧ badC ≠null
outC!<badC>;
badC=null;
valid(x)
badC=x;
valid(x)
badC'=x;current'=5;
credit'= credit+x.getValue();
escrow'=escrow⌢<x>;
(x.getValue() + credit) < price
credit'= credit+x.getValue();
escrow'=escrow⌢<x>;
(x.getValue() + credit) < price
credit'= credit+x.getValue();
vault'=vault⌢escrow⌢<x>;escrow'= <>;
credit'= credit+x.getValue();
vault'=vault⌢escrow⌢<x>;
escrow'= <>;
(x.getValue() + credit) > price
(x.getValue() + credit) > price
credit'= credit+x.getValue();
vault'=vault⌢escrow⌢<x>; escrow'= <>;
return
current∈{1,2,3,4}==0 ∧
price>0 ∧ credit>price ∧
badC≠null ∧escrow==<>
credit'= credit+x.getValue();
vault'=vault⌢escrow⌢<x>;
escrow'=<>;
dispense
current∈{1,2,3,4}∧ price>0
∧ credit==price ∧
badC==null∧ escrow==<>
t = 1.0
playDispenseMessage()
outC!getChange(credit-price);
credit'= price;
cup!getBeverage(current); credit'= 0; price'=0; current'=0;
credit= 0; price=0; current=0; badC=null;
BVMState
vault : [COIN]
escrow : [COIN]
badC : P COIN
credit, price, current : P N
Δ
ϕ : {idle, charge, dispense, cancel, return, reject}
v(badC) ∉ {10, 20, 50, 100, 200}
price ∈ {0, 80, 100, 120, 130}
current ∈ {0, 1, 2, 3, 4, 5}
ϕ = idle ⇒ credit = current = price = 0 ∧ badC = null ∧ escrow = []
ϕ = charge ⇒ current ∈ {1, 2, 3, 4} ∧ price > 0 ∧ credit < price ∧ badC = null
ϕ = dispense ⇒ current ∈ {1, 2, 3, 4} ∧ price > 0 ∧ credit = price ∧ badC = null ∧ escrow = []
ϕ = return ⇒ current ∈ {1, 2, 3, 4} ∧ price > 0 ∧ credit > price ∧ badC = null ∧ escrow = []
ϕ = cancel ⇒ current = 5 ∧ credit < price
ϕ = reject ⇒ current ∈ {1, 2, 3, 4} ∧ badC ≠ null

Figure 7.13BVM example of translation of HSystem's state space to Z state schema

The green arrow on the top of the figure indicates that the name of the Z state schema produced is obtained from the name of the HSystem whose state space is been mapped. The orange and sky blue arrows match the declarations and predicates in the BVM's state schema with declarations and predicates in the BVMState. Declarations *vault*, *escrow* and *badC* in BVMState are derived from the references from BVM to the corresponding HClasses as described in the previous chapter.

The purple arrows indicate that the type $\{idle, charge, dispense, cancel, return, reject\}$of declaration ϕin BVMState match with the set of labels of all configurations specified in BVM. If this were generated automatically with the rule, the declaration name would be "BVMConfigs". Finally, the red arrows match the configuration labels and their properties with the corresponding predicates in BVMState.

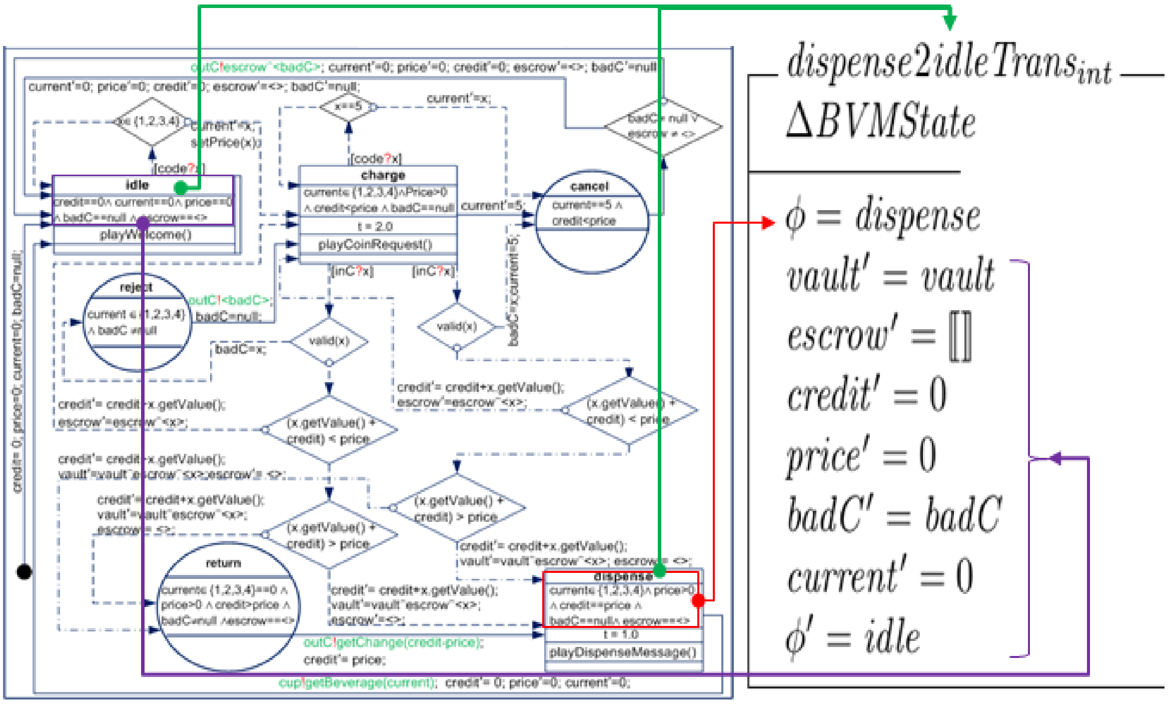
7.3.2.5 Mapping HSystem's internal configuration transitions to Z operation schemas



```
179 lazy rule hInternalTransition2zOperationSchema {
180     from                            --translate internal transition to Z operation schema
181         hIntTrans : HiLLS!InternalTransition
182         using {--local variables used in the "do"section of the rule to refine the mapping
183             sourceConfiguration:String = hIntTrans.source.label;
184             targetConfiguration:String = hIntTrans.target.label;
185         }
186     to
187         zOperationSchema : Zed!OperationSchema (--schema name = source_label2target_labelINT
188             name <- sourceConfiguration.concat('2').concat(targetConfiguration).concat('INT'),
189             preConditions <- hIntTrans.source.property.predicate,
190             postConditions <- hIntTrans.target.property.predicate
191         )
192 }
```

Figure 7.14Mapping rule to translate HiLLS internal transition to Z schema

We present, in Figure 7.14, the mapping rules to translate a given HiLLS internal configuration transition *hIntTrans* to a Z operation schema, *zOperationSchema*. Looking into the "to" section of the rule (lines 188-190); *zOperationSchema* gets its name from the concatenation of the labels of the source and target configurations of *hIntTrans* in the format: *sourceLabel2targetLabelINT*. The properties of the source and target configurations of *hIntTrans* translate to the pre- and post-conditions respectively of *zOperationSchema*.

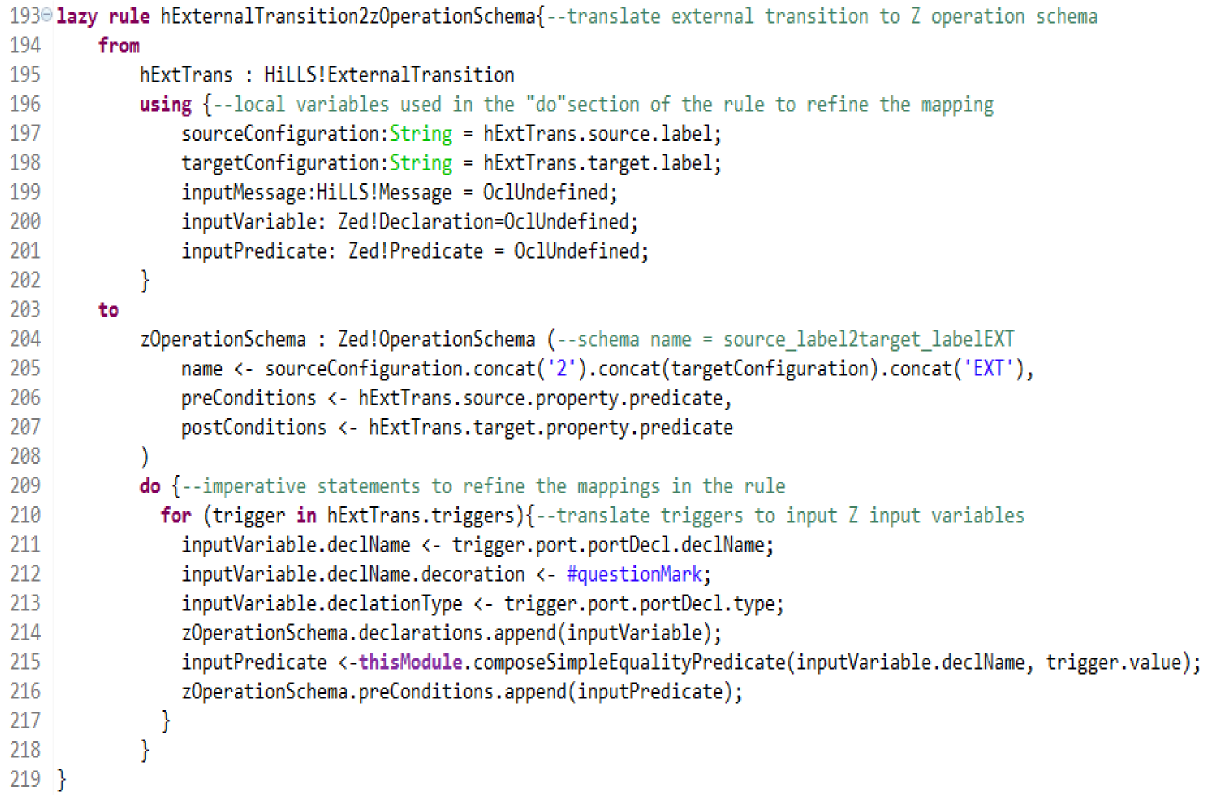


outC!escrow⁀<badC>; current'=0; price'=0; credit'=0; escrow'=<>; badC'=null
current'=0; price'=0; credit'=0; escrow'=<>; badC'=null;
x∈ {1,2,3,4}
current'=x;
setPrice(x);
x==5
current'=x;
badC ≠ null ∨ escrow ≠ <>
[code?x]
idle
credit==0∧ current==0∧ price==0 ∧ badC==null ∧ escrow==<>
playWelcome()
[code?x]
charge
current∈ {1,2,3,4}∧Price>0 ∧ credit<price ∧ badC==null
t = 2.0
playCoinRequest()
current'=5;
cancel
current==5 ∧ credit<price
[inC?x]
[inC?x]
reject
current ∉ {1,2,3,4} ∧ badC ≠null
outC!<badC>;
badC=null;
valid(x)
badC=x;
valid(x)
badC=x;current=5;
credit'= credit+x.getValue(); escrow'=escrow⁀<x>;
(x.getValue() + credit) < price
credit'= credit+x.getValue(); escrow'=escrow⁀<x>;
(x.getValue() + credit) < price
credit'= credit+x.getValue(); vault'=vault⁀escrow⁀<x>;escrow'= <>;
credit'= credit+x.getValue(); vault'=vault⁀escrow⁀<x>; escrow'= <>;
(x.getValue() + credit) > price
(x.getValue() + credit) > price
credit'= credit+x.getValue(); vault'=vault⁀escrow⁀<x>; escrow'= <>;
credit= 0; price=0; current=0; badC=null;
return
current∈ {1,2,3,4}==0 ∧ price>0 ∧ credit>price ∧ badC≠null ∧escrow==<>
credit'= credit+x.getValue(); vault'=vault⁀escrow⁀<x>; escrow'=<>;
dispense
current∈ {1,2,3,4}∧ price>0 ∧ credit==price ∧ badC==null∧ escrow==<>
t = 1.0
playDispenseMessage()
outC!getChange(credit-price);
credit'= price;
cup!getBeverage(current); credit'= 0; price'=0; current'=0;
dispense2idleTrans_int
ΔBVMState
φ = dispense
vault' = vault
escrow' = []
credit' = 0
price' = 0
badC' = badC
current' = 0
φ' = idle

Figure 7.15 BVM example of the translation of HiLLS internal transition to Z

As an example to illustrate the mapping rules in Figure 7.14, Figure 7.15 presents the correspondences between the HiLLS and Z specifications of the internal transition from *dispense* to *idle* in the BVM. The concatenation of the labels of the source and target configurations in the format prescribed in the rule gives the name of the operation schema on the right of the figure. The precondition of the schema is the same as the property of configuration *dispense*, which is the source and the post condition is the property of the target configuration *idle*.

7.3.2.6 Mapping HSystem's external configuration transitions to Z operation schemas



```
193 lazy rule hExternalTransition2zOperationSchema{--translate external transition to Z operation schema
194     from
195         hExtTrans : HiLLS!ExternalTransition
196         using {--local variables used in the "do"section of the rule to refine the mapping
197             sourceConfiguration:String = hExtTrans.source.label;
198             targetConfiguration:String = hExtTrans.target.label;
199             inputMessage:HiLLS!Message = OclUndefined;
200             inputVariable: Zed!Declaration=OclUndefined;
201             inputPredicate: Zed!Predicate = OclUndefined;
202         }
203     to
204         zOperationSchema : Zed!OperationSchema (--schema name = source_label2target_labelEXT
205             name <- sourceConfiguration.concat('2').concat(targetConfiguration).concat('EXT'),
206             preConditions <- hExtTrans.source.property.predicate,
207             postConditions <- hExtTrans.target.property.predicate
208         )
209         do {--imperative statements to refine the mappings in the rule
210           for (trigger in hExtTrans.triggers){--translate triggers to input Z input variables
211             inputVariable.declName <- trigger.port.portDecl.declName;
212             inputVariable.declName.decoration <- #questionMark;
213             inputVariable.declationType <- trigger.port.portDecl.type;
214             zOperationSchema.declarations.append(inputVariable);
215             inputPredicate <-thisModule.composeSimpleEqualityPredicate(inputVariable.declName, trigger.value);
216             zOperationSchema.preConditions.append(inputPredicate);
217           }
218         }
219 }
```

Figure 7.16Mapping rule to translate HiLLS external transition to Z schema

Figure 7.16 presents the rule to translate a HiLLS external transition *hExtTrans* to a Z operation schema *zOperationSchema*. The "to" section of the rule is similar to that of the internal transition. The present rule, however, specifies a "do" section (lines 209-218) in which the *triggers* of the transition are extracted to refine the synthesized *zOperationSchema*.

For each trigger of *hExtTrans*, the name and type of the associated input port are extracted to declare an input variable, which is added to the set of declarations of *zOperationSchema*(lines 211-214). The input variable so created, together with the received message, is also used to compose a predicate, which is added to the set of preconditions in the predicate part of *zOperationSchema* (lines 215-216).

We illustrate the application of this rule, in Figure 7.17, by using it to match the HiLLS and Z specifications of the external transition from *charge* to *cancel* in the BVM. The trigger **[code?x]** in the HiLLS specification translates into the input variable *code*?*CODE* in the Z specification. The input variable "*code*?", together with the received value (5) constitutes the precondition *code?=5* in addition to the precondition synthesized from the property of the source configuration.

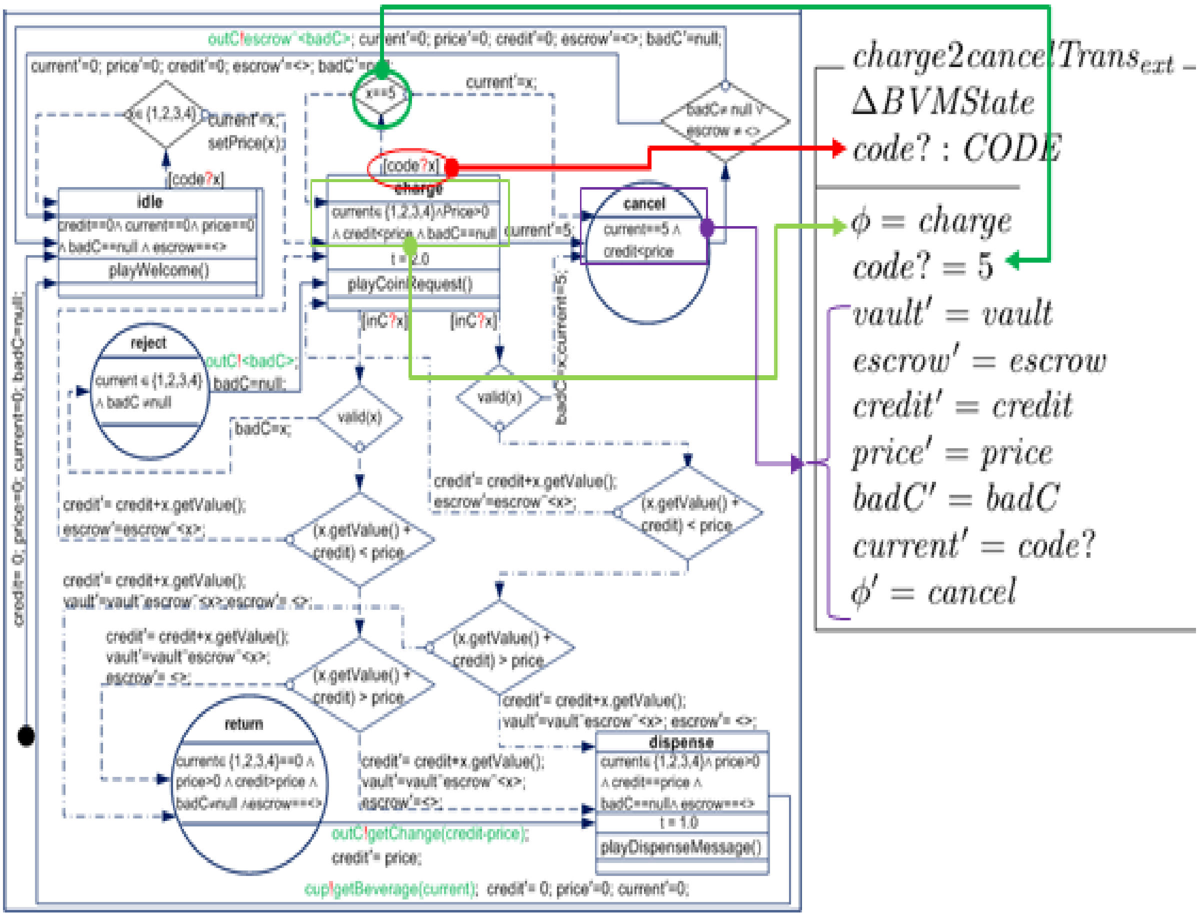

outC!escrow⌢<badC>; current'=0; price'=0; credit'=0; escrow'=<>; badC'=null;
current'=0; price'=0; credit'=0; escrow'=<>; badC'=null;
x∈{1,2,3,4}
current'=x;
setPrice(x);
x==5
current'=x;
badC≠null ∨ escrow ≠ <>
[code?x]
[code?x]
idle
credit==0∧ current==0∧ price==0 ∧ badC==null ∧ escrow==<>
playWelcome()
charge
current∈{1,2,3,4}∧Price>0 ∧ credit<price ∧ badC==null
t = 2.0
playCoinRequest()
current'=5;
cancel
current==5 ∧ credit<price
[inC?x]
[inC?x]
badC'=x;current'=5;
reject
current ∉ {1,2,3,4} ∧ badC ≠null
outC!<badC>;
badC=null;
valid(x)
valid(x)
badC=x;
credit'= credit+x.getValue();
escrow'=escrow⌢<x>;
(x.getValue() + credit) < price
credit'= credit+x.getValue();
escrow'=escrow⌢<x>;
(x.getValue() + credit) < price
credit'= credit+x.getValue();
vault'=vault⌢escrow⌢<x>;escrow'= <>;
credit'= credit+x.getValue();
vault'=vault⌢escrow⌢<x>;
escrow'= <>;
(x.getValue() + credit) > price
(x.getValue() + credit) > price
credit'= credit+x.getValue();
vault'=vault⌢escrow⌢<x>; escrow'= <>;
return
current∉{1,2,3,4}==0 ∧ price>0 ∧ credit>price ∧ badC≠null ∧escrow==<>
credit'= credit+x.getValue();
vault'=vault⌢escrow⌢<x>;
escrow'=<>;
dispense
current∈{1,2,3,4}∧ price>0 ∧ credit==price ∧ badC==null∧ escrow==<>
t = 1.0
playDispenseMessage()
outC!getChange(credit-price);
credit'= price;
cup!getBeverage(current); credit'= 0; price'=0; current'=0;
credit'= 0; price'=0; current'=0; badC'=null;
$charge2cancelTrans_{ext}$
$\Delta BVMState$
$code? : CODE$
$\phi = charge$
$code? = 5$
$vault' = vault$
$escrow' = escrow$
$credit' = credit$
$price' = price$
$badC' = badC$
$current' = code?$
$\phi' = cancel$

Figure 7.17 BVM example of the translation of HiLLS external transition to Z

7.3.3 HiLLS Requirement to Temporal Logic

In the integration of the different concepts in the HiLLS metamodel in the previous chapter, the concepts for expressing requirements were incorporated into the metamodel by interfacing with the metamodel of the TL patterns as a whole. Technically, HiLLS adopts the constructs of the pattern as is and provides HiLLS-based graphical notations to express them in a more user-friendly format. Hence, there is a bijective mapping between the requirement constructs in HiLLS and the metamodel of TL patterns presented in Section 6.2.3.

Consequently, we expect that given any tool for expressing TL constructs, we can define the templates for the various patterns in the tool and automatically synthesize the required parameters for such templates from a given HiLLS-based requirement specification. We intend to explore this aspect in our future work.

7.4 ENACTMENT (EXECUTION) SEMANTICS

In the section, we present the mapping of the concepts described in the metamodel HiLLS to the enactment framework presented in Chapter 5. The framework provides Java-based templates to create models of coupled and atomic systems for enactment in the framework. Therefore, the semantics mapping from HiLLS to the framework is a model-to-text (M2T) transformation to generate codes based on the framework's templates from a given HiLLS model. We get the actual enactment semantics of the HiLLS model when the generated codes are executed.

We use Acceleo Model Transformation Language (MTL) to write the code generators with the HiLLS metamodel as source and the enactment model templates as the targets. Acceleo MTL[15] is an Eclipse-based code generator, which implements the OMG's MOF M2T specification[16].

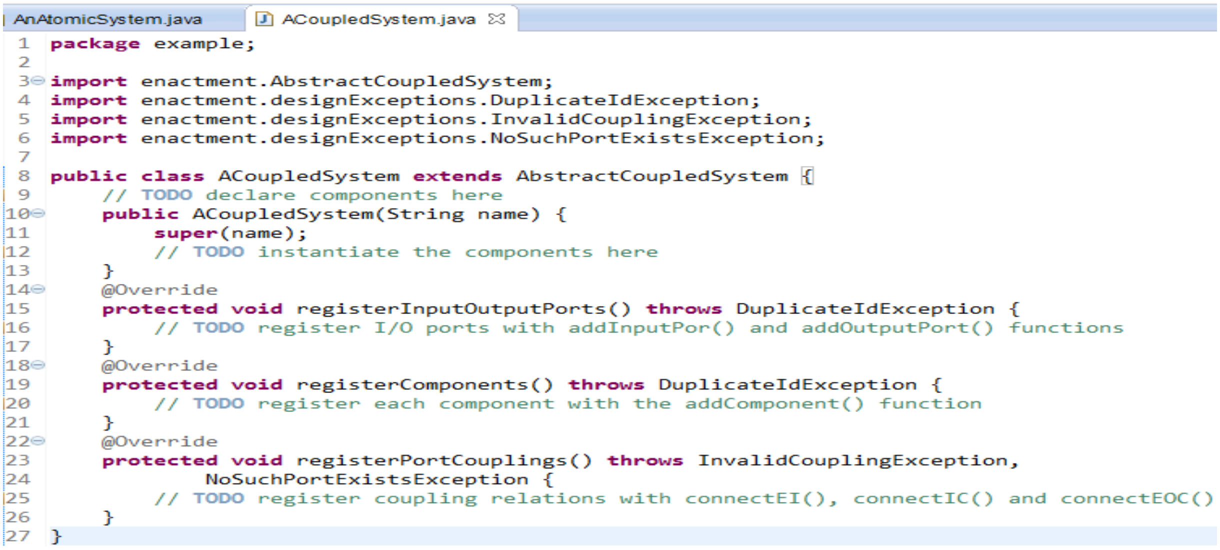
In the rest of this section, we present each of the two template models and follow it with the code generator, which generates the codes for its unimplemented methods as well as Java methods for user-defined operations in the HiLLS model. As proofs of concept, we will illustrate the generators of essential model components with correspondences between the HiLLS model of the BVS and its enactment model. We also present a code generator to generate conventional Java classes for HiLLS HClass models.

7.4.1 Enactment Semantics of HiLLS Composite HSystem

The enactment semantics of a HiLLS' composite HSystem is given by the execution of an equivalent coupled enactment model in the enactment framework. Hence, we specify a translational semantics to generate the enactment code from the HiLLS model. To begin with, we present, in Figure 7.18, the template for creating coupled DES models for execution in the enactment framework. Thus, we have to define a code generator that generates, from a given coupled HSystem, the appropriate codes for the unimplemented methods in the template. To facilitate the reader's understanding, we present the code generator in three fragments to generate codes for different elements of the template in Figure 7.18.

[15] https://wiki.eclipse.org/Acceleo

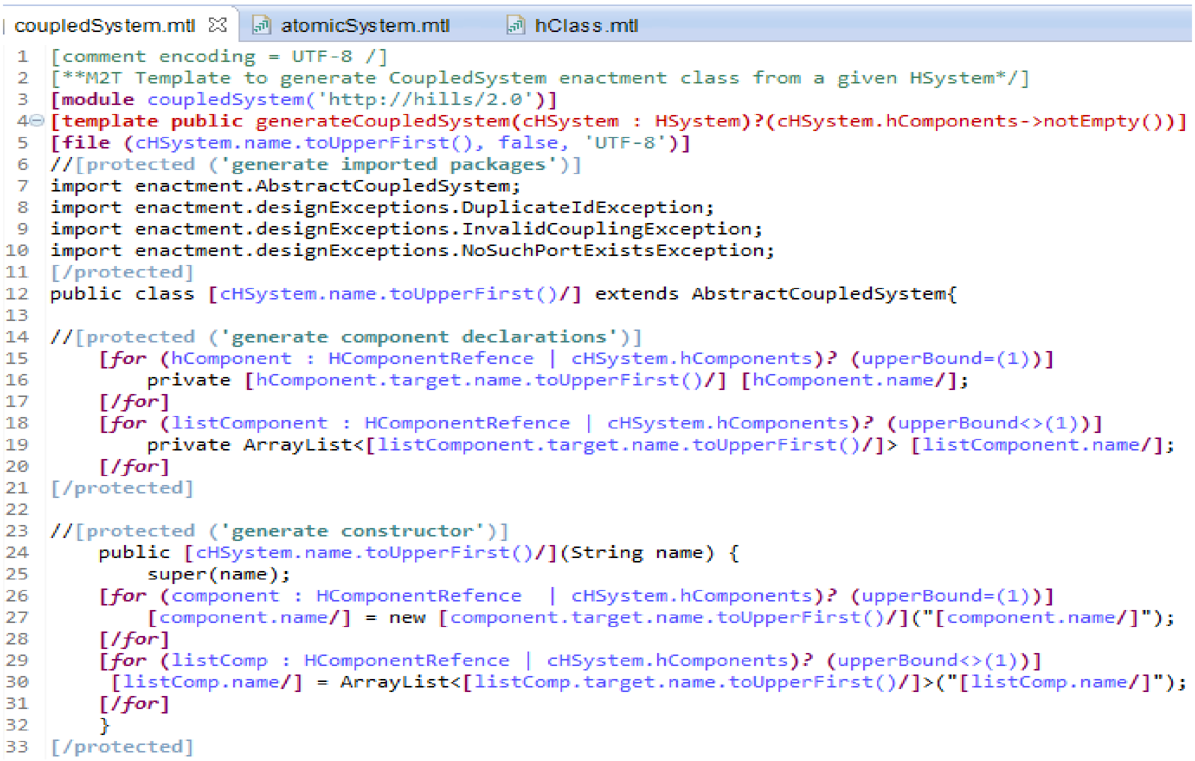
[16] http://www.omg.org/spec/MOFM2T/1.0/



```java
package example;

import enactment.AbstractCoupledSystem;
import enactment.designExceptions.DuplicateIdException;
import enactment.designExceptions.InvalidCouplingException;
import enactment.designExceptions.NoSuchPortExistsException;

public class ACoupledSystem extends AbstractCoupledSystem {
    // TODO declare components here
    public ACoupledSystem(String name) {
        super(name);
        // TODO instantiate the components here
    }
    @Override
    protected void registerInputOutputPorts() throws DuplicateIdException {
        // TODO register I/O ports with addInputPor() and addOutputPort() functions
    }
    @Override
    protected void registerComponents() throws DuplicateIdException {
        // TODO register each component with the addComponent() function
    }
    @Override
    protected void registerPortCouplings() throws InvalidCouplingException,
            NoSuchPortExistsException {
        // TODO register coupling relations with connectEI(), connectIC() and connectEOC()
    }
}
```

Figure 7.18Template for creating coupled DES models for enactment

7.4.1.1 Code generator fragment for required packages,componentsand constructor



```
[comment encoding = UTF-8 /]
[**M2T Template to generate CoupledSystem enactment class from a given HSystem*/]
[module coupledSystem('http://hills/2.0')]
[template public generateCoupledSystem(cHSystem : HSystem)?(cHSystem.hComponents->notEmpty())]
[file (cHSystem.name.toUpperFirst(), false, 'UTF-8')]
//[protected ('generate imported packages')]
import enactment.AbstractCoupledSystem;
import enactment.designExceptions.DuplicateIdException;
import enactment.designExceptions.InvalidCouplingException;
import enactment.designExceptions.NoSuchPortExistsException;
[/protected]
public class [cHSystem.name.toUpperFirst()/] extends AbstractCoupledSystem{

//[protected ('generate component declarations')]
    [for (hComponent : HComponentRefence | cHSystem.hComponents)? (upperBound=(1))]
        private [hComponent.target.name.toUpperFirst()/] [hComponent.name/];
    [/for]
    [for (listComponent : HComponentRefence | cHSystem.hComponents)? (upperBound<>(1))]
        private ArrayList<[listComponent.target.name.toUpperFirst()/]> [listComponent.name/];
    [/for]
[/protected]

//[protected ('generate constructor')]
    public [cHSystem.name.toUpperFirst()/](String name) {
        super(name);
    [for (component : HComponentRefence  | cHSystem.hComponents)? (upperBound=(1))]
        [component.name/] = new [component.target.name.toUpperFirst()/]("[component.name/]");
    [/for]
    [for (listComp : HComponentRefence | cHSystem.hComponents)? (upperBound<>(1))]
     [listComp.name/] = ArrayList<[listComp.target.name.toUpperFirst()/]>("[listComp.name/]");
    [/for]
    }
[/protected]
```

Figure 7.19 Code generator for coupled system class, its ports and components

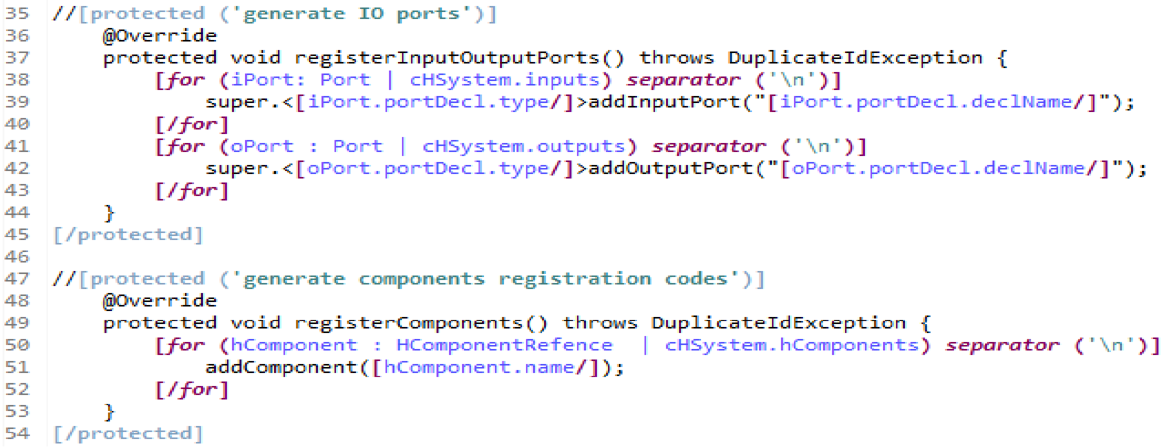
Figure 7.19 presents the fragment of the code generator that generates the codes for the class template in Figure 7.18 from the beginning to the constructor.

Line 4 (in Figure 7.19) specifies that this code generator is applicable only to a HiLLS HSystem, *cHSystem*, which has some *hComponentReference*s to other HSystems. Lines 7-11 generate the code to import the default-required packages specified in the target template. The generation of the class begins in line 12; the class's name is generated from the name attribute of *cHSystem* based on Java naming convention.

In lines 14-21, we generate the declarations of components of the system as private attributes. Recall from the HiLLS metamodel in Section 6.2 that an *HComponentReferece* has *name*, *lowerBound* and *upperBound* attributes and a reference, *target*, to an HSystem. In lines 14-21 of the code generator segment above, an *hComponentReference* with *upperBound* equal to 1 is declared as a complex attribute with name extracted from the name of *hComponentReference* and type extracted from the name of the HSystem referenced by the *target*reference. If *upperBound* is greater than 1, however, we generate an array list instead.

Lines 23-33 generate the class constructor. We generate the codes to instantiate, within the constructor, all components declared in the previous lines.

7.4.1.2 Code generator fragment for ports and components registrations



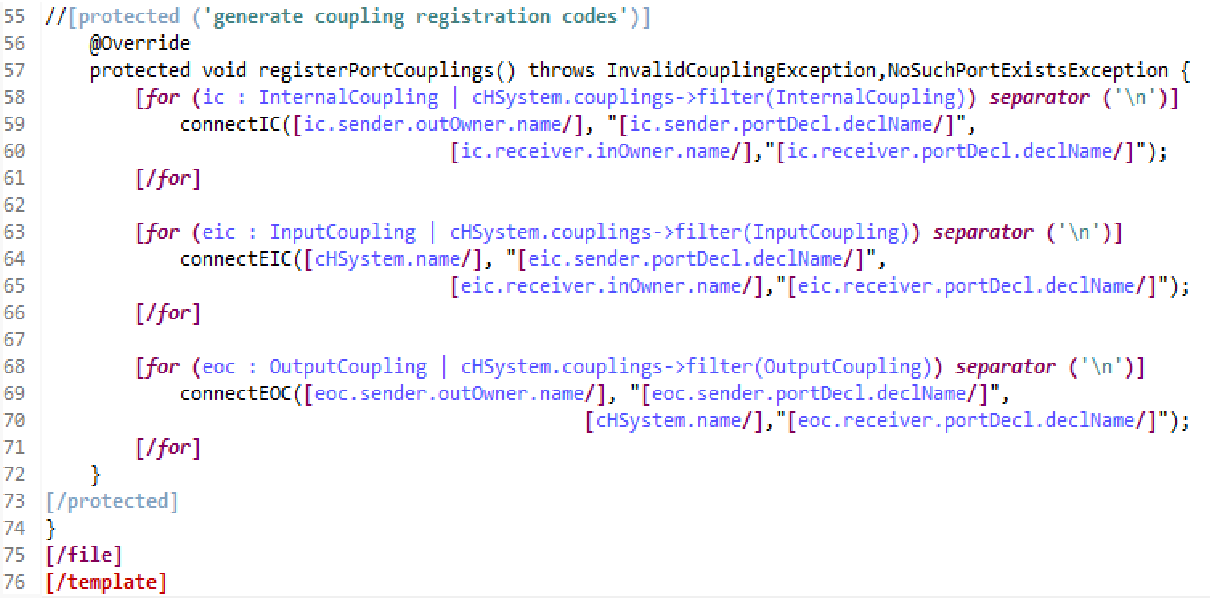
```
35 //[protected ('generate IO ports')]
36     @Override
37     protected void registerInputOutputPorts() throws DuplicateIdException {
38         [for (iPort: Port | cHSystem.inputs) separator ('\n')]
39             super.<[iPort.portDecl.type/]>addInputPort("[iPort.portDecl.declName/]");
40         [/for]
41         [for (oPort : Port | cHSystem.outputs) separator ('\n')]
42             super.<[oPort.portDecl.type/]>addOutputPort("[oPort.portDecl.declName/]");
43         [/for]
44     }
45 [/protected]
46 
47 //[protected ('generate components registration codes')]
48     @Override
49     protected void registerComponents() throws DuplicateIdException {
50         [for (hComponent : HComponentRefence  | cHSystem.hComponents) separator ('\n')]
51             addComponent([hComponent.name/]);
52         [/for]
53     }
54 [/protected]
```

Figure 7.20 Code generator for ports and components registrations

Figure 7.20 presents the segments of the code generator that generate the code to register input and output ports and components of the coupled system. Lines 35-45 generate the method *registerInputOutputPorts*(). To register any port in this method, the port's name and type are required. In the HiLLS metamodel, a port refers to a declaration, which has a name and type. The code generator extracts these informations from the input model to implement the method.

Similarly, lines 47-54 generates method *registerComponents*() and its implementation.

7.4.1.3 Code generator fragment for coupling registrations



```
//[protected ('generate coupling registration codes')]
    @Override
    protected void registerPortCouplings() throws InvalidCouplingException,NoSuchPortExistsException {
        [for (ic : InternalCoupling | cHSystem.couplings->filter(InternalCoupling)) separator ('\n')]
            connectIC([ic.sender.outOwner.name/], "[ic.sender.portDecl.declName/]",
                                    [ic.receiver.inOwner.name/],"[ic.receiver.portDecl.declName/]");
        [/for]

        [for (eic : InputCoupling | cHSystem.couplings->filter(InputCoupling)) separator ('\n')]
            connectEIC([cHSystem.name/], "[eic.sender.portDecl.declName/]",
                                    [eic.receiver.inOwner.name/],"[eic.receiver.portDecl.declName/]");
        [/for]

        [for (eoc : OutputCoupling | cHSystem.couplings->filter(OutputCoupling)) separator ('\n')]
            connectEOC([eoc.sender.outOwner.name/], "[eoc.sender.portDecl.declName/]",
                                                [cHSystem.name/],"[eoc.receiver.portDecl.declName/]");
        [/for]
    }
[/protected]
}
[/file]
[/template]
```

Figure 7.21 Code generator for coupling registrations

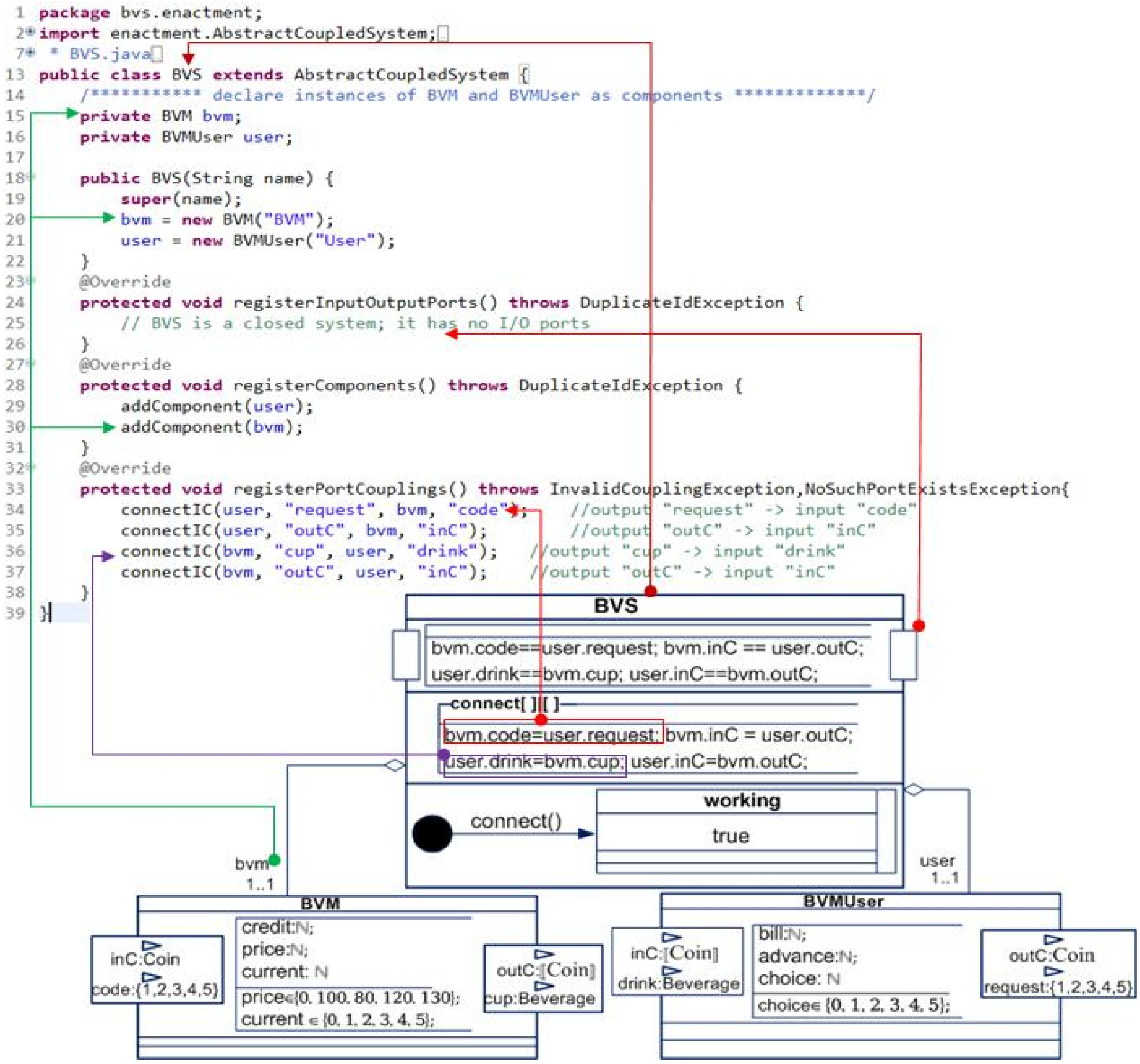
Finally, Figure 7.21 presents the segment of the code generator that generate the method *registerPortCouplings*() and its implementation. Each statement in the method is a call to one of methods *connectIC*, *connectEIC* and *connectEOC* each of which requires four parameters in the order: *sending system's name, sending port's name, receiving system's name, receiving port's name*. Each of these parameters is extracted from the input HiLLS model as specified in the generator template.

7.4.1.4 Relations between the elements of the HiLLS and enactment models of the BVS

As proof of concept, we present, in Figure 7.22, the correspondences between the HiLLS model of the BVS and its enactment model (see Section 5.5.1.3) to demonstrate that the latter can be fully generated from the former.

The brick-red arrow shows a correspondence between the names of the two models. The green arrows indicate that the *hComponentReferencebvm* in the HiLLS model is linked to the declaration, instantiation and registration of component *bvm* in the enactment code. The same also applies to component *user*.

The red arrowfrom the empty output interface of the HiLLS BVS to the *registerInputOutputPorts*() function of the enactment code indicates that both models have no input or output ports. Finally, the red and purple arrows pointing to the function *registerportCouplings*() show that each of the parameters of the connect methods has a corresponding construct in the HiLLS model.



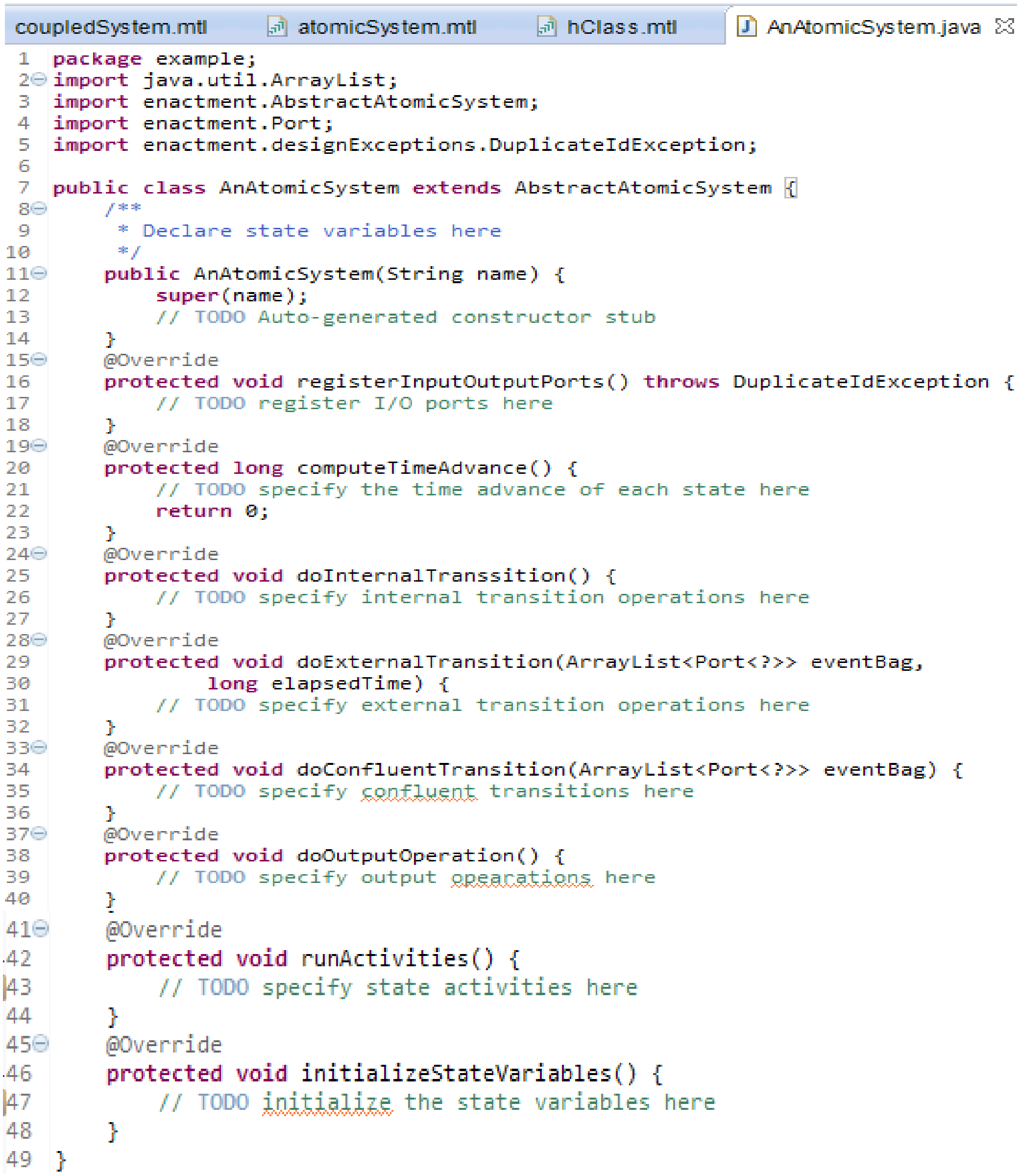
```java
package bvs.enactment;
import enactment.AbstractCoupledSystem;
 * BVS.java
public class BVS extends AbstractCoupledSystem {
    /*********** declare instances of BVM and BVMUser as components *************/
    private BVM bvm;
    private BVMUser user;

    public BVS(String name) {
        super(name);
        bvm = new BVM("BVM");
        user = new BVMUser("User");
    }
    @Override
    protected void registerInputOutputPorts() throws DuplicateIdException {
        // BVS is a closed system; it has no I/O ports
    }
    @Override
    protected void registerComponents() throws DuplicateIdException {
        addComponent(user);
        addComponent(bvm);
    }
    @Override
    protected void registerPortCouplings() throws InvalidCouplingException,NoSuchPortExistsException{
        connectIC(user, "request", bvm, "code");      //output "request" -> input "code"
        connectIC(user, "outC", bvm, "inC");          //output "outC" -> input "inC"
        connectIC(bvm, "cup", user, "drink");      //output "cup" -> input "drink"
        connectIC(bvm, "outC", user, "inC");       //output "outC" -> input "inC"
    }
}
```

BVS
bvm.code==user.request; bvm.inC == user.outC;
user.drink==bvm.cup; user.inC==bvm.outC;
connect[ ][ ]
bvm.code=user.request; bvm.inC = user.outC;
user.drink=bvm.cup; user.inC=bvm.outC;
working
connect()
true
bvm
1..1
user
1..1
BVM
inC:Coin
code:{1,2,3,4,5}
credit:N;
price:N;
current: N
price∈{0, 100, 80, 120, 130};
current ∈ {0, 1, 2, 3, 4, 5};
outC:[Coin]
cup:Beverage
BVMUser
inC:[Coin]
drink:Beverage
bill:N;
advance:N;
choice: N
choice∈ {0, 1, 2, 3, 4, 5};
outC:Coin
request:{1,2,3,4,5}

Figure 7.22 Correspondences between HiLLS and enactment models of the BVS

7.4.2 Enactment Semantics of HiLLS Atomic HSystem

The equivalent atomic system model, based on the enactment framework, when executed, provides the enactment semantics of a given atomic HSystem. As a reminder, we first present, in Figure 7.23, the *AtomicSystem*template, which defines the structure of an atomic system model. Note that in addition to the methods inherited from the framework, the user may define some special-purpose methods, which are called from the implementations of the inherited methods to perform some specific functions.



```java
package example;
import java.util.ArrayList;
import enactment.AbstractAtomicSystem;
import enactment.Port;
import enactment.designExceptions.DuplicateIdException;

public class AnAtomicSystem extends AbstractAtomicSystem {
    /**
     * Declare state variables here
     */
    public AnAtomicSystem(String name) {
        super(name);
        // TODO Auto-generated constructor stub
    }
    @Override
    protected void registerInputOutputPorts() throws DuplicateIdException {
        // TODO register I/O ports here
    }
    @Override
    protected long computeTimeAdvance() {
        // TODO specify the time advance of each state here
        return 0;
    }
    @Override
    protected void doInternalTranssition() {
        // TODO specify internal transition operations here
    }
    @Override
    protected void doExternalTransition(ArrayList<Port<?>> eventBag,
            long elapsedTime) {
        // TODO specify external transition operations here
    }
    @Override
    protected void doConfluentTransition(ArrayList<Port<?>> eventBag) {
        // TODO specify confluent transitions here
    }
    @Override
    protected void doOutputOperation() {
        // TODO specify output opearcations here
    }
    @Override
    protected void runActivities() {
        // TODO specify state activities here
    }
    @Override
    protected void initializeStateVariables() {
        // TODO initialize the state variables here
    }
}
```

Figure 7.23 Template for creating atomic system models for enactment

Therefore, we define the enactment semantics of an atomic HSystem by generating from it, the implementations of the unimplemented methods in this template. We present the code generator in fragments to generate different parts of the class to facilitate the reader's understanding.

We will use the correspondences between different parts of the HiLLS model of the BVM and its enactment code for illustrations in this section; thus, we re-present the HiLLS model in Figure 7.24 to optimize our numerous references to it.

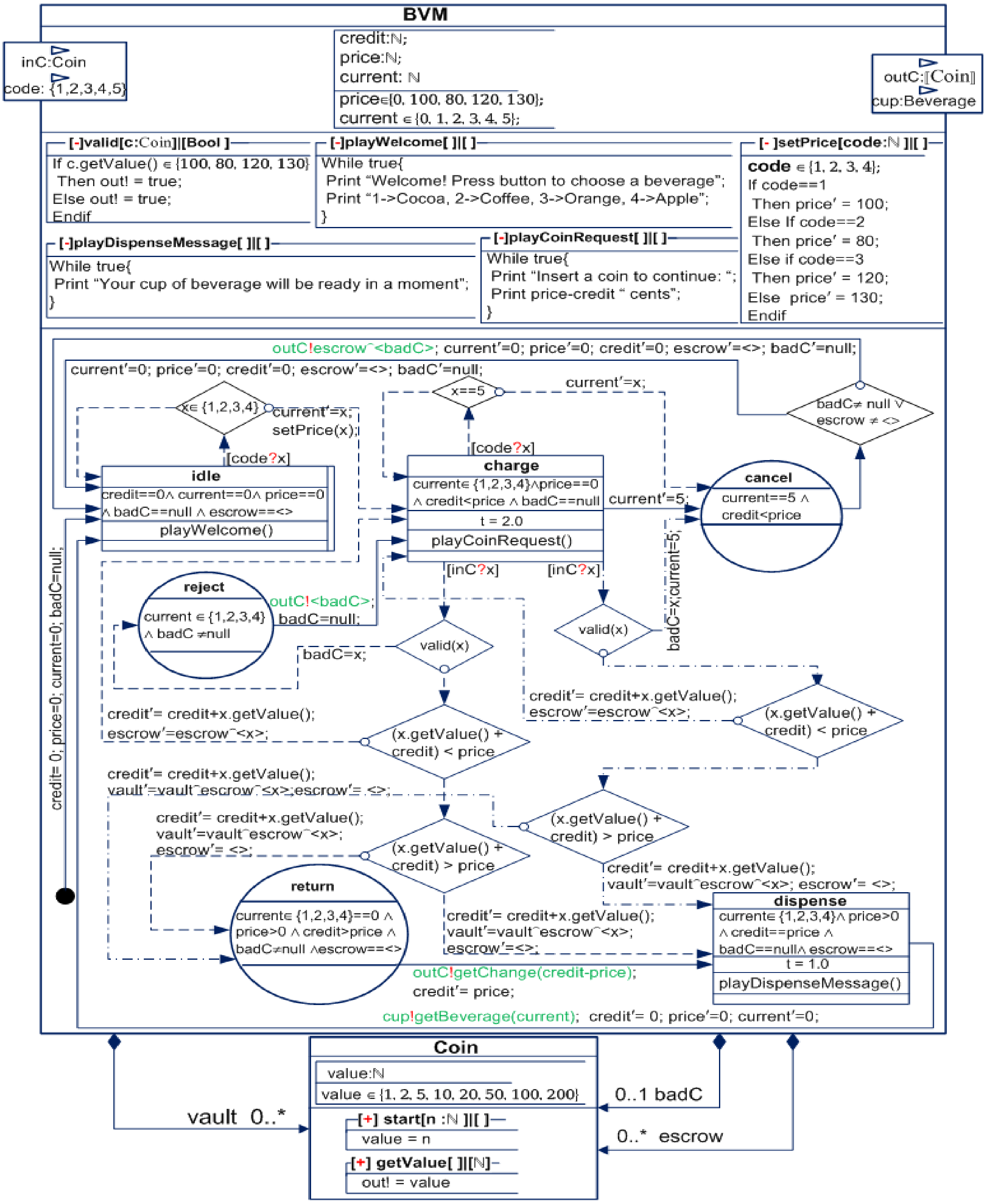
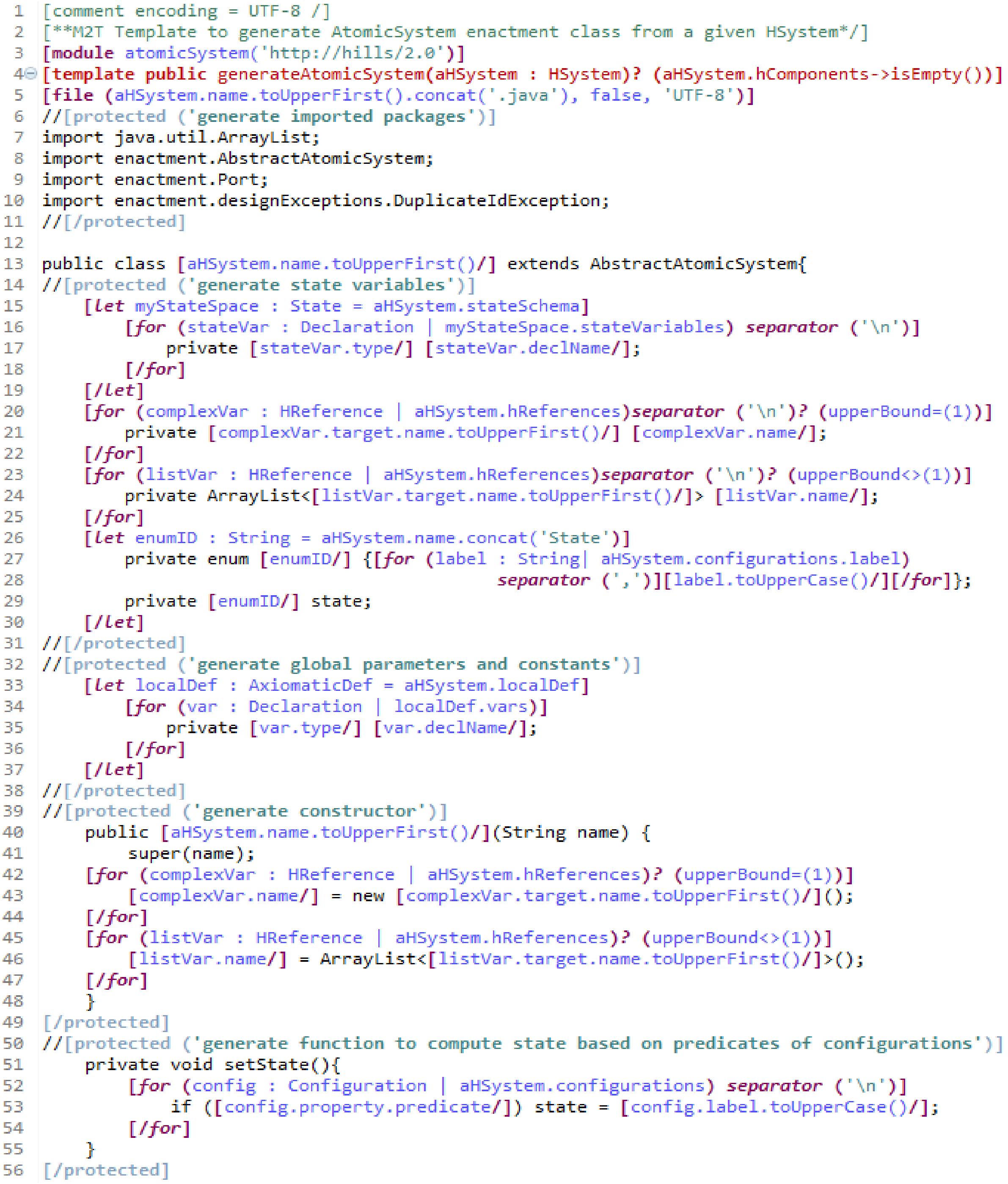

BVM
inC:Coin
code: {1,2,3,4,5}
credit:ℕ;
price:ℕ;
current: ℕ
price∈{0, 100, 80, 120, 130};
current ∈{0, 1, 2, 3, 4, 5};
outC:[Coin]
cup:Beverage
[-]valid[c:Coin][Bool]
If c.getValue() ∈ {100, 80, 120, 130}
Then out! = true;
Else out! = true;
Endif
[-]playWelcome[ ][ ]
While true{
Print "Welcome! Press button to choose a beverage";
Print "1->Cocoa, 2->Coffee, 3->Orange, 4->Apple";
}
[-]setPrice[code:ℕ][ ]
code ∈ {1, 2, 3, 4};
If code==1
Then price′ = 100;
Else If code==2
Then price′ = 80;
Else if code==3
Then price′ = 120;
Else price′ = 130;
Endif
[-]playDispenseMessage[ ][ ]
While true{
Print "Your cup of beverage will be ready in a moment";
}
[-]playCoinRequest[ ][ ]
While true{
Print "Insert a coin to continue: ";
Print price-credit " cents";
}
outC!escrow⁀<badC>; current′=0; price′=0; credit′=0; escrow′=<>; badC′=null;
current′=0; price′=0; credit′=0; escrow′=<>; badC′=null;
x∈ {1,2,3,4}
current′=x;
setPrice(x);
[code?x]
x==5
current′=x;
[code?x]
badC≠ null ∨ escrow ≠ <>
idle
credit==0∧ current==0∧ price==0 ∧ badC==null ∧ escrow==<>
playWelcome()
charge
current∈ {1,2,3,4}∧price==0 ∧ credit<price ∧ badC==null
t = 2.0
playCoinRequest()
current′=5;
cancel
current==5 ∧ credit<price
[inC?x]
[inC?x]
reject
current ∈ {1,2,3,4} ∧ badC ≠null
outC!<badC>;
badC=null;
badC=x;
valid(x)
valid(x)
badC=x;current=5;
credit= 0; price=0; current=0; badC=null;
credit′= credit+x.getValue();
escrow′=escrow⁀<x>;
(x.getValue() + credit) < price
credit′= credit+x.getValue();
escrow′=escrow⁀<x>;
(x.getValue() + credit) < price
credit′= credit+x.getValue();
vault′=vault⁀escrow⁀<x>;escrow′= <>;
credit′= credit+x.getValue();
vault′=vault⁀escrow⁀<x>;
escrow′= <>;
(x.getValue() + credit) > price
(x.getValue() + credit) > price
credit′= credit+x.getValue();
vault′=vault⁀escrow⁀<x>; escrow′= <>;
return
current∈ {1,2,3,4}==0 ∧ price>0 ∧ credit>price ∧ badC≠null ∧escrow==<>
credit′= credit+x.getValue();
vault′=vault⁀escrow⁀<x>;
escrow′=<>;
dispense
current∈ {1,2,3,4}∧ price>0 ∧ credit==price ∧ badC==null∧ escrow==<>
t = 1.0
playDispenseMessage()
outC!getChange(credit-price);
credit′= price;
cup!getBeverage(current); credit′= 0; price′=0; current′=0;
Coin
value:ℕ
value ∈ {1, 2, 5, 10, 20, 50, 100, 200}
[+] start[n :ℕ][ ]
value = n
[+] getValue[ ][ℕ]
out! = value
vault 0..*
0..1 badC
0..* escrow

Figure 7.24HiLLS specification of the BVM

7.4.2.1 *Code generator fragment for an atomic system class, its state space and required packages*



```
[comment encoding = UTF-8 /]
[**M2T Template to generate AtomicSystem enactment class from a given HSystem*/]
[module atomicSystem('http://hills/2.0')]
[template public generateAtomicSystem(aHSystem : HSystem)? (aHSystem.hComponents->isEmpty())]
[file (aHSystem.name.toUpperFirst().concat('.java'), false, 'UTF-8')]
//[protected ('generate imported packages')]
import java.util.ArrayList;
import enactment.AbstractAtomicSystem;
import enactment.Port;
import enactment.designExceptions.DuplicateIdException;
//[/protected]

public class [aHSystem.name.toUpperFirst()/] extends AbstractAtomicSystem{
//[protected ('generate state variables')]
    [let myStateSpace : State = aHSystem.stateSchema]
        [for (stateVar : Declaration | myStateSpace.stateVariables) separator ('\n')]
            private [stateVar.type/] [stateVar.declName/];
        [/for]
    [/let]
    [for (complexVar : HReference | aHSystem.hReferences)separator ('\n')? (upperBound=(1))]
        private [complexVar.target.name.toUpperFirst()/] [complexVar.name/];
    [/for]
    [for (listVar : HReference | aHSystem.hReferences)separator ('\n')? (upperBound<>(1))]
        private ArrayList<[listVar.target.name.toUpperFirst()/]> [listVar.name/];
    [/for]
    [let enumID : String = aHSystem.name.concat('State')]
        private enum [enumID/] {[for (label : String| aHSystem.configurations.label)
                                            separator (',')][label.toUpperCase()/][/for]};
        private [enumID/] state;
    [/let]
//[/protected]
//[protected ('generate global parameters and constants')]
    [let localDef : AxiomaticDef = aHSystem.localDef]
        [for (var : Declaration | localDef.vars)]
            private [var.type/] [var.declName/];
        [/for]
    [/let]
//[/protected]
//[protected ('generate constructor')]
    public [aHSystem.name.toUpperFirst()/](String name) {
        super(name);
    [for (complexVar : HReference | aHSystem.hReferences)? (upperBound=(1))]
        [complexVar.name/] = new [complexVar.target.name.toUpperFirst()/]();
    [/for]
    [for (listVar : HReference | aHSystem.hReferences)? (upperBound<>(1))]
        [listVar.name/] = ArrayList<[listVar.target.name.toUpperFirst()/]>();
    [/for]
    }
[/protected]
//[protected ('generate function to compute state based on predicates of configurations')]
    private void setState(){
        [for (config : Configuration | aHSystem.configurations) separator ('\n')]
            if ([config.property.predicate/]) state = [config.label.toUpperCase()/];
        [/for]
    }
[/protected]
```

Figure 7.25 Code generator segment for an atomic system and its state space

Figure 7.25above presents the segment of the generator that creates a Java class (and class the file) for an atomic HSystem and generates the enactment codes for its state space.

As specified in line 4, this code generator template is applicable to a HSystem *aHSystem* that has no components; i.e., an atomic HSystem. First, in line 5, it extracts the name *aHSystem* and uses it to name a new Java file based on Java file naming convention.

Lines 6-11 generate the default-required packages as suggested by the template in Figure 7.23. The *AtomicSystem* class is generated in line 13; the class' name is derived from the name of *aHSystem* based on Java naming convention.

The *let* blocks in lines 15-25 map all variables declared in the *state schema* of *aHSystem* and all its *hR*eferences to private attributes of the class. The *let* block in lines 26-30 maps the labels of all configurations defined in *aHSystem* into an enumeration, which is used as the type of another state variable named "state". In lines 33-37, all declarations in the *localDef* (axiomatic schema) of *aHSystem* are mapped to private attributes.

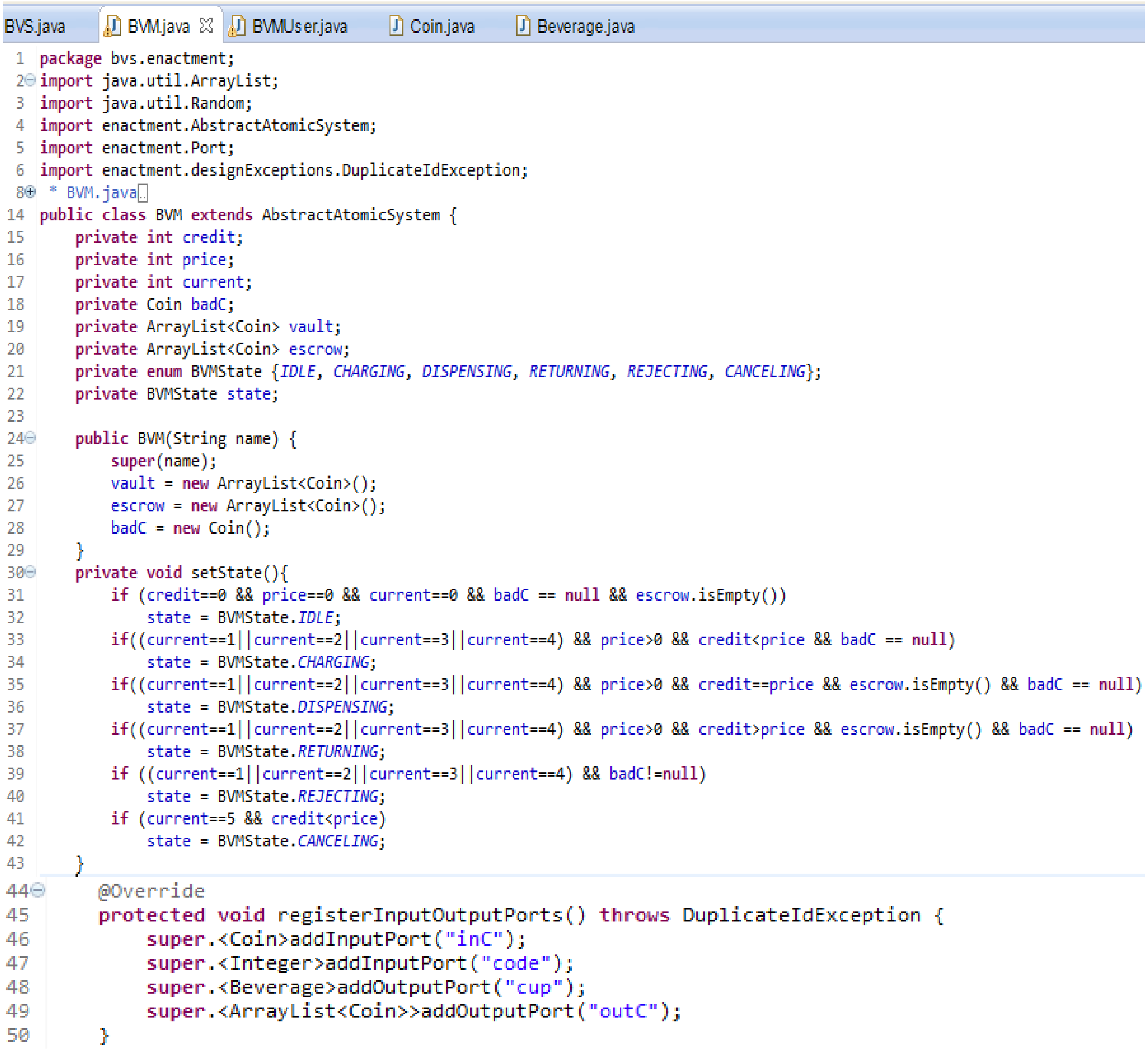
The constructor of the class is generated in lines 40-48; all non-primitive attribute declarations generated in the previous lines are instantiated within the constructor block.

We generate user-defined method *setState*() in lines 51-55, especially to complete the definition of the state space of *aHSystem*. Within *setState*, we map the *property* of every *configuration* defined in *aHSystem* to the condition of an "if statement", and in the conditional statement, we assign the configuration's *label* to the variable "state" generated previously in line 29.

To illustrate the expected result of applying this generator segment to the HiLLS model of BVM (see Figure 7.24.), we present the corresponding segment of the enactment code to discuss the relationships between their elements. We can see that the variables *credit*, *price* and *current* declared in the former's state schema as well as its *hRefrencesbadC*, *escrow* and *vault* translate into private attributes in the latter. According to the code generation rules in line 26-30 (Figure 7.25), the labels of all configurations defined in the HiLLS model translate into an enum type, *BVMState*, in the enactment code and this type is used to declare an attribute *state*.

Following the code generation rules in lines 39-49, the constructor of the enactment model class creates instances of all non-primitive attributes generated from the *hReference*s of the HiLLS model.

Finally, in accordance to code generation rules in lines 50-56, we see that for each configuration in the HiLLS model, there is an "if statement" in the *setState*() function of the enactment code, which has the configuration's property as *condition* while its *label* is assigned to attribute *state*.



```
1  package bvs.enactment;
2  import java.util.ArrayList;
3  import java.util.Random;
4  import enactment.AbstractAtomicSystem;
5  import enactment.Port;
6  import enactment.designExceptions.DuplicateIdException;
8   * BVM.java
14 public class BVM extends AbstractAtomicSystem {
15     private int credit;
16     private int price;
17     private int current;
18     private Coin badC;
19     private ArrayList<Coin> vault;
20     private ArrayList<Coin> escrow;
21     private enum BVMState {IDLE, CHARGING, DISPENSING, RETURNING, REJECTING, CANCELING};
22     private BVMState state;
23
24     public BVM(String name) {
25         super(name);
26         vault = new ArrayList<Coin>();
27         escrow = new ArrayList<Coin>();
28         badC = new Coin();
29     }
30     private void setState(){
31         if (credit==0 && price==0 && current==0 && badC == null && escrow.isEmpty())
32             state = BVMState.IDLE;
33         if((current==1||current==2||current==3||current==4) && price>0 && credit<price && badC == null)
34             state = BVMState.CHARGING;
35         if((current==1||current==2||current==3||current==4) && price>0 && credit==price && escrow.isEmpty() && badC == null)
36             state = BVMState.DISPENSING;
37         if((current==1||current==2||current==3||current==4) && price>0 && credit>price && escrow.isEmpty() && badC == null)
38             state = BVMState.RETURNING;
39         if ((current==1||current==2||current==3||current==4) && badC!=null)
40             state = BVMState.REJECTING;
41         if (current==5 && credit<price)
42             state = BVMState.CANCELING;
43     }
44     @Override
45     protected void registerInputOutputPorts() throws DuplicateIdException {
46         super.<Coin>addInputPort("inC");
47         super.<Integer>addInputPort("code");
48         super.<Beverage>addOutputPort("cup");
49         super.<ArrayList<Coin>>addOutputPort("outC");
50     }
```

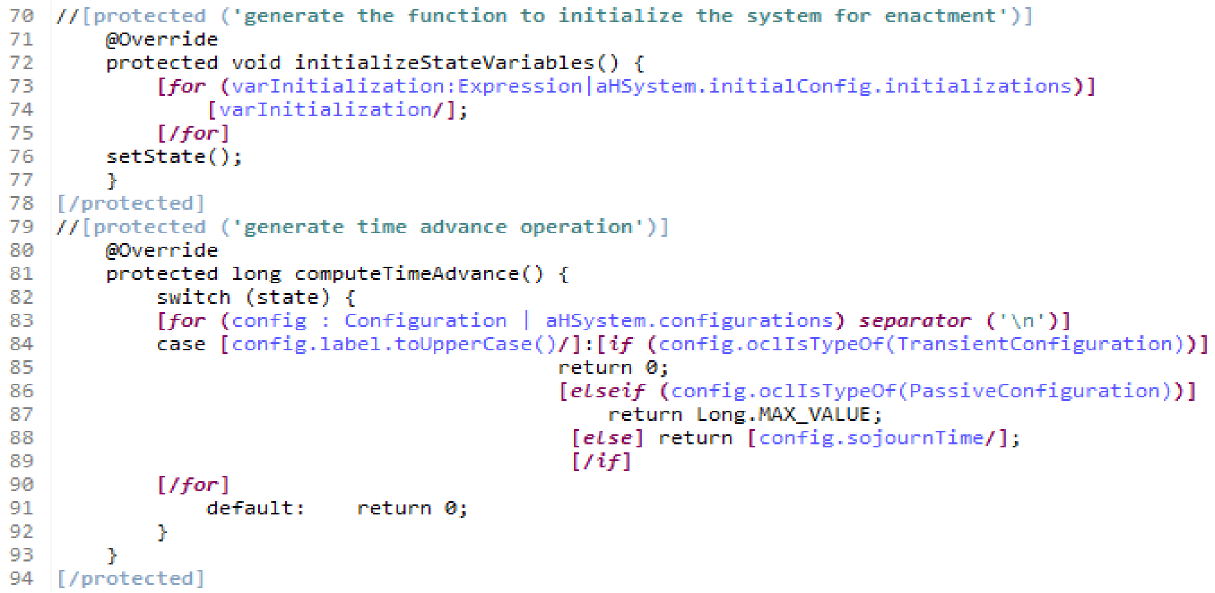
Figure 7.26 A sample state space and port registration code of an atomic system model for enactment

7.4.2.2 Code generator fragment for port registration in an atomic system

The generator fragment that generates the code for the implementation of method *registerInputOutputPorts*() in atomic systems is the same as described previously for coupled system in Section 7.4.1.2; we only present an example of its result in this section.

In accordance to the code generator rules, Figure 7.25above (lines 44-50) presents an example of the correspondences between the port specifications in the HiLLS model and port registration in the enactment model. For each of input ports *inC* and *code* specified in the HiLLS model, its name and type provides the parameter for a call to the *addInputPort*() method in the enactment code. Similarly, each of HiLLS' output port *cup* and *outC* provides the parameters required to invoke the addOutputPort() method.

7.4.2.3 Code generator fragment for time advance and system initialization in an atomic system



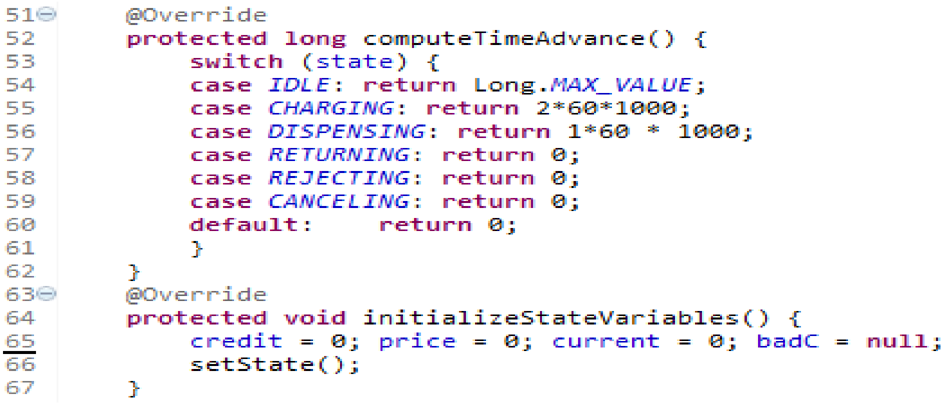
```
70 //[protected ('generate the function to initialize the system for enactment')]
71     @Override
72     protected void initializeStateVariables() {
73         [for (varInitialization:Expression|aHSystem.initialConfig.initializations)]
74             [varInitialization/];
75         [/for]
76     setState();
77     }
78 [/protected]
79 //[protected ('generate time advance operation')]
80     @Override
81     protected long computeTimeAdvance() {
82         switch (state) {
83         [for (config : Configuration | aHSystem.configurations) separator ('\n')]
84         case [config.label.toUpperCase()/]:[if (config.oclIsTypeOf(TransientConfiguration))]
85                                         return 0;
86                                         [elseif (config.oclIsTypeOf(PassiveConfiguration))]
87                                             return Long.MAX_VALUE;
88                                          [else] return [config.sojournTime/];
89                                          [/if]
90         [/for]
91             default:    return 0;
92         }
93     }
94 [/protected]
```

Figure 7.27 Code generator fragment for initialization and time advance in atomic system

Figure 7.27 presents the segments of the code generator that generate the implementations codes for methods *initializeStateVariables*() and *computeTimeAdvance*() from a given atomic HSystem. Lines 70-78 specify the implantation of the former; each of the expressions specified on the transition from the initial state notation to the starting configuration in the HiLLS model translates into a statement in the method. A statement to invoke the method *setStat*() is included at the end of the method.

To implement the method *computeTimeAdvance*() (lines 79-94), we generate a switch statement with class attribute *state* as its case. Within the switch, we iterate over all the configurations defined in the HiLLS model and extract their labels and *sojourn Tim*es to build each case of the switch statement. Only finite configurations are queried for their *sojournTime* expressions; zero and Long.Max_VALUE are generated for instances of transient and passive configurations respectively.

We illustrate the expected result of applying these rules to the HiLLS model of BVM in Figure 7.28.



```java
@Override
protected long computeTimeAdvance() {
    switch (state) {
    case IDLE: return Long.MAX_VALUE;
    case CHARGING: return 2*60*1000;
    case DISPENSING: return 1*60 * 1000;
    case RETURNING: return 0;
    case REJECTING: return 0;
    case CANCELING: return 0;
    default:     return 0;
    }
}
@Override
protected void initializeStateVariables() {
    credit = 0; price = 0; current = 0; badC = null;
    setState();
}
```

Figure 7.28 Sample enactment code for time advance and initial state specifications

For the *computeTimeAdvance*() method, note that *returning*, *rejecting* and *canceling* are transient while *idle* is a passive configuration. For *charging* and *dispensing*, the specified sojourn times are 2 and 1 minutes respectively.

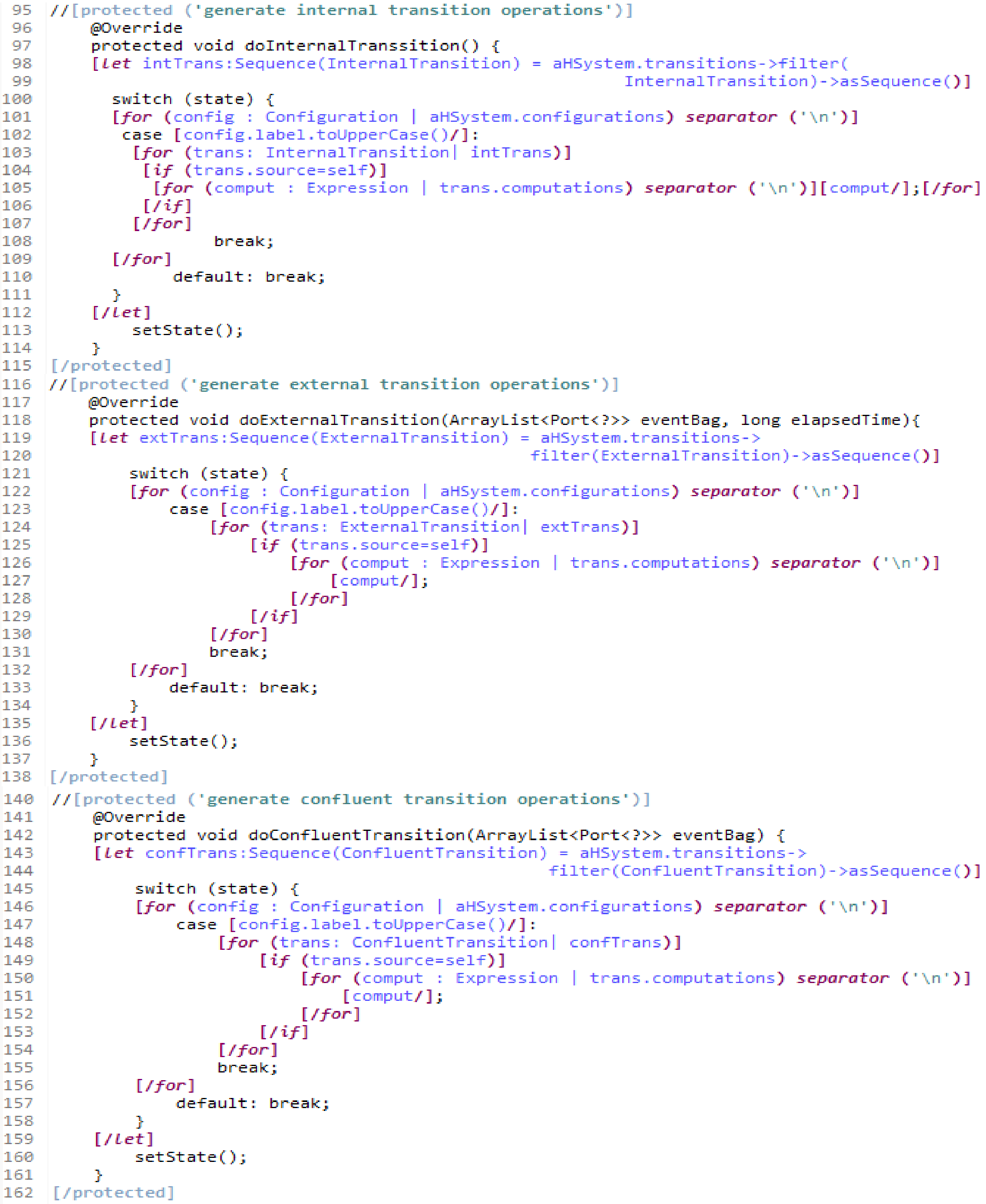
In Figure 7.24, the arrow from the initial state notation terminates on configuration *idle* initialization expressions *credit* = 0; *price* = 0; *current* =0; *badC* = *null*. According to the code generation rules, these translate to the implementation of method *initializeStateVariables*() as shown above.

7.4.2.4 Code generator segments for state transition functions of an atomic system class

We present the code generation segments for the implementations of methods *doInternalTransition*(), *doExternalTransition*() and *doConfluentTransition*() in Figure 7.29.

Lines 95-115 generate the implementation of method *doInternalTransition*(). First, the set of all internal configuration transitions in the input HiLLS model are collected in a local variable *intTrans*. Then, a Java switch statement is created, again with class attribute *state* as its case variable. In lines 101-109, for each configuration defined in the model, create a case for its label and then search for a transition in *intTrans*, whose *source* configuration is the current configurations; if any is found, print the sequence of expressions that define its *computations*. Print the command "*break*;" before taking the next configuration in the loop. Then, print *default: break;* and then *setState*() after exiting the loop. The implementations of the other two transition methods follow similar patterns as specified in Figure 7.29.

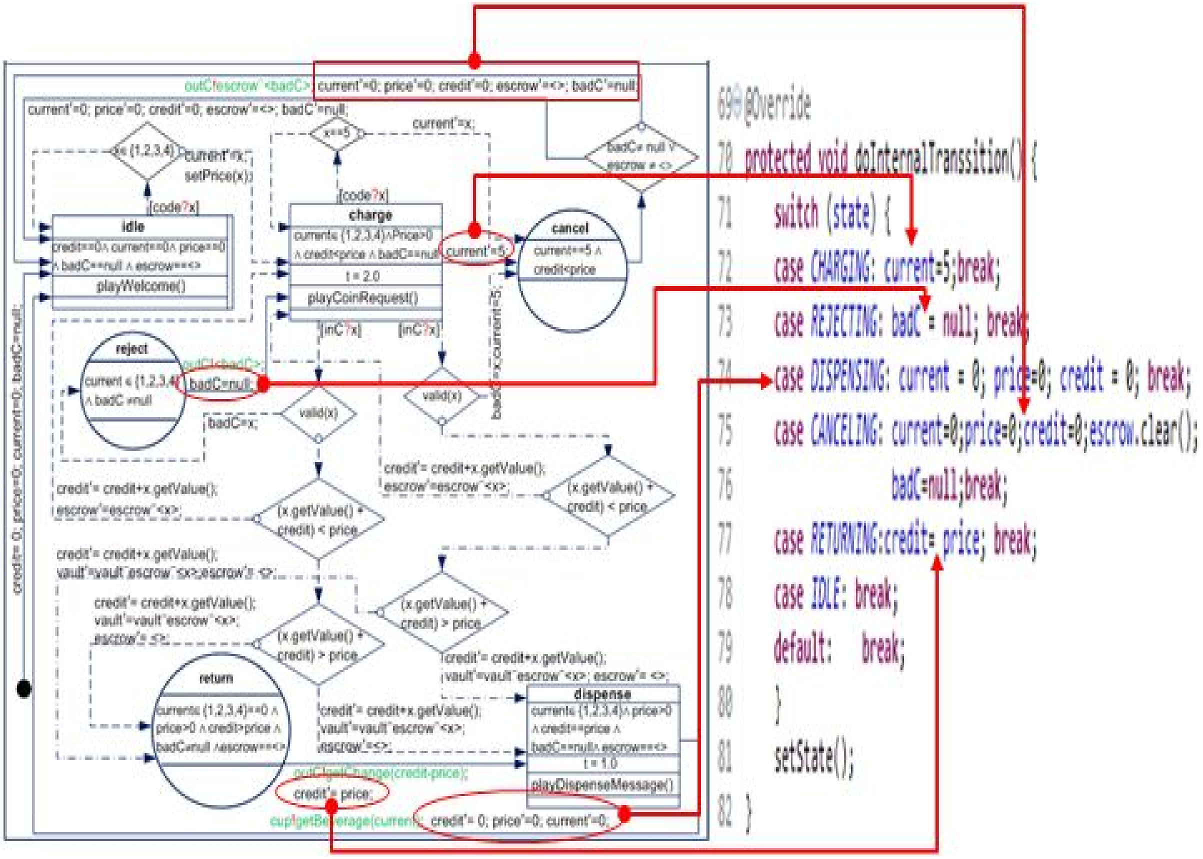
For this generator to be efficient, however, there is need to define a mechanism for the generator to identify branching of transition paths along condition nodes in the configuration transition diagram and to generate appropriate "if" statements. The present solution is most effective for transitions paths without such branches. We intend to address this limitation in our future work.



```
//[protected ('generate internal transition operations')]
    @Override
    protected void doInternalTranssition() {
    [let intTrans:Sequence(InternalTransition) = aHSystem.transitions->filter(
                                                        InternalTransition)->asSequence()]
      switch (state) {
      [for (config : Configuration | aHSystem.configurations) separator ('\n')]
       case [config.label.toUpperCase()/]:
        [for (trans: InternalTransition| intTrans)]
         [if (trans.source=self)]
          [for (comput : Expression | trans.computations) separator ('\n')][comput/];[/for]
         [/if]
        [/for]
                break;
      [/for]
            default: break;
      }
    [/let]
        setState();
    }
[/protected]
//[protected ('generate external transition operations')]
    @Override
    protected void doExternalTransition(ArrayList<Port<?>> eventBag, long elapsedTime){
    [let extTrans:Sequence(ExternalTransition) = aHSystem.transitions->
                                                filter(ExternalTransition)->asSequence()]
        switch (state) {
        [for (config : Configuration | aHSystem.configurations) separator ('\n')]
            case [config.label.toUpperCase()/]:
                [for (trans: ExternalTransition| extTrans)]
                    [if (trans.source=self)]
                        [for (comput : Expression | trans.computations) separator ('\n')]
                            [comput/];
                        [/for]
                    [/if]
                [/for]
                break;
        [/for]
            default: break;
        }
    [/let]
        setState();
    }
[/protected]
//[protected ('generate confluent transition operations')]
    @Override
    protected void doConfluentTransition(ArrayList<Port<?>> eventBag) {
    [let confTrans:Sequence(ConfluentTransition) = aHSystem.transitions->
                                                filter(ConfluentTransition)->asSequence()]
        switch (state) {
        [for (config : Configuration | aHSystem.configurations) separator ('\n')]
            case [config.label.toUpperCase()/]:
                [for (trans: ConfluentTransition| confTrans)]
                    [if (trans.source=self)]
                        [for (comput : Expression | trans.computations) separator ('\n')]
                            [comput/];
                        [/for]
                    [/if]
                [/for]
                break;
        [/for]
            default: break;
        }
    [/let]
        setState();
    }
[/protected]
```

Figure 7.29 Code generatorsegments for state transition functions

Figure 7.30 shows the correspondences between the HiLLS model of BVM and the implementation of method *doInternalTransition*() in its enactment code to illustrate the expected result of executing these rules on the HiLLS model. The arrows show equivalences of the computations accompanying internal transitions in the two models.


outC!escrow^<badC>
current'=0; price'=0; credit'=0; escrow'=<>; badC'=null;
current'=0; price'=0; credit'=0; escrow'=<>; badC'=null;
x==5
current'=x;
x∈{1,2,3,4}
current'=x;
setPrice(x);
badC≠ null ∨ escrow ≠ <>
[code?x]
[code?x]
idle
credit==0∧ current==0∧ price==0 ∧ badC==null ∧ escrow==<>
playWelcome()
charge
current∈{1,2,3,4}∧Price>0 ∧ credit<price ∧ badC==null
t = 2.0
playCoinRequest()
current'=5;
cancel
current==5 ∧ credit<price
[inC?x]
[inC?x]
reject
current ∈ {1,2,3,4} ∧ badC ≠null
outC!<badC>
badC=null;
badC=x;
valid(x)
valid(x)
badC=x;current=5;
credit'= credit+x.getValue();
escrow'=escrow^<x>;
(x.getValue() + credit) < price
credit'= credit+x.getValue();
escrow'=escrow^<x>;
(x.getValue() + credit) < price
credit'= credit+x.getValue();
vault'=vault^escrow^<x>;escrow'= <>;
credit'= credit+x.getValue();
vault'=vault^escrow^<x>;
escrow'= <>;
(x.getValue() + credit) > price
(x.getValue() + credit) > price
credit'= credit+x.getValue();
vault'=vault^escrow^<x>; escrow'= <>;
return
current∈{1,2,3,4}==0 ∧ price>0 ∧ credit>price ∧ badC≠null ∧escrow==<>
credit'= credit+x.getValue();
vault'=vault^escrow^<x>;
escrow'=<>;
dispense
current∈{1,2,3,4}∧ price>0 ∧ credit==price ∧ badC==null∧ escrow==<>
t = 1.0
playDispenseMessage()
outC!getChange(credit-price);
credit'= price;
cup!getBeverage(current);
credit'= 0; price'=0; current'=0;
credit= 0; price=0; current=0; badC=null;

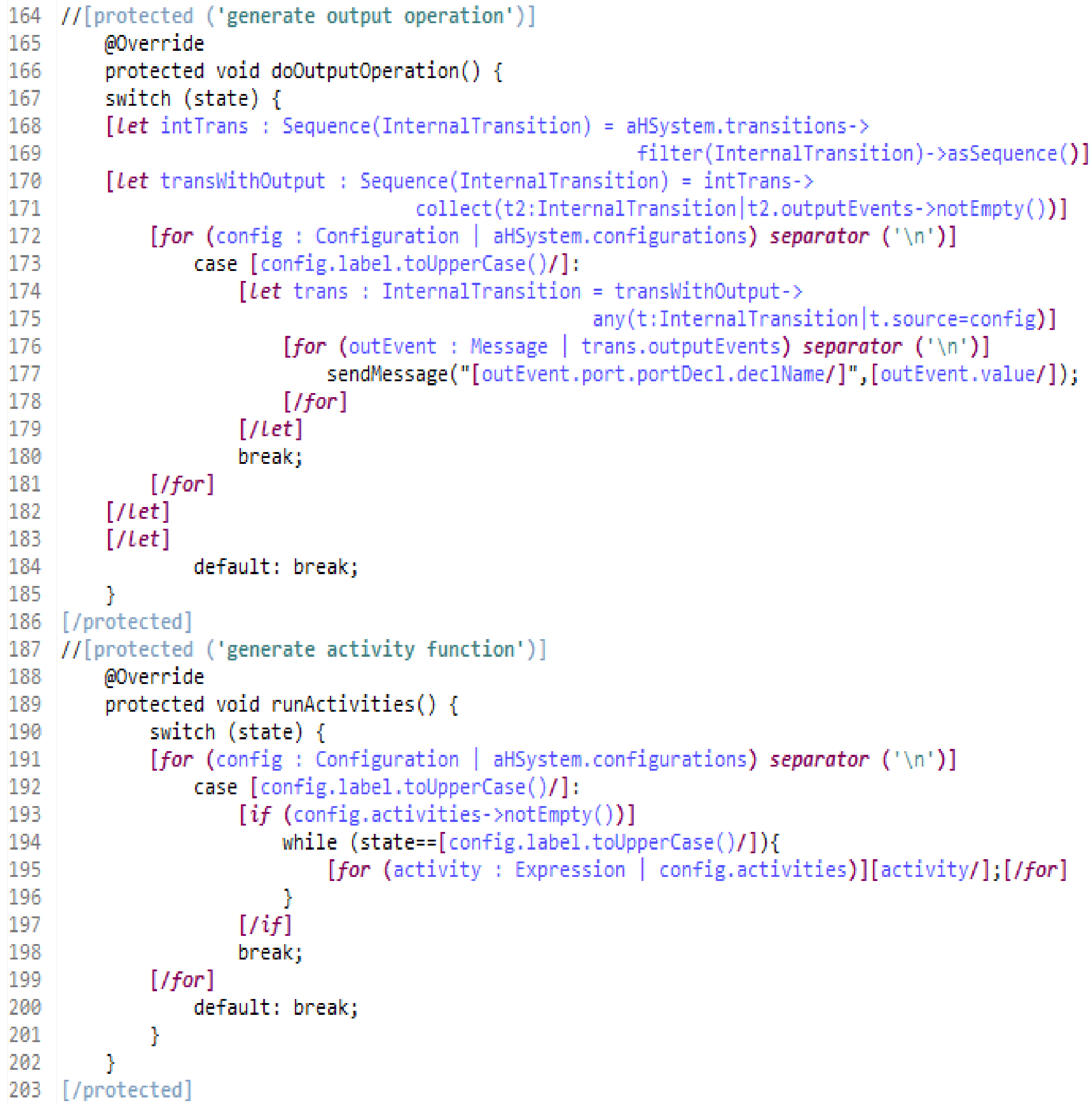
```
@Override
protected void doInternalTranssition() {
    switch (state) {
    case CHARGING: current=5;break;
    case REJECTING: badC = null; break;
    case DISPENSING: current = 0; price=0; credit = 0; break;
    case CANCELING: current=0;price=0;credit=0;escrow.clear();
                    badC=null;break;
    case RETURNING:credit= price; break;
    case IDLE: break;
    default:   break;
    }
    setState();
}
```

Figure 7.30Relations between the enactment code for internal state transitions in BVM and its HiLLS model

7.4.2.5 Code generator segments for output and activity functions of an atomic system class

Figure 7.31 presents the segments of the code generator that synthesize the implementations of methods *doOutputOperation*() (lines 164-186) and *runActivities*() (lines 186-203) from a given HiLLS atomic HSystem.

To implement the *doOutputOperation*(), the generator creates a switch statement with attribute *state* as case variable and collects all transitions with associated output operations in a local variable *transWithOutput* (line 170-171). For each configuration *config* in the HiLLS model, a switch case is created. Then, a search is made if there is an element of *transWithOutput,* which has *config* as its source configuration; if found, then the value of each message and its associated output port name are used to invoke the framework method *sendMessage*(). Every iteration of configurations terminates with the printing of "break;"

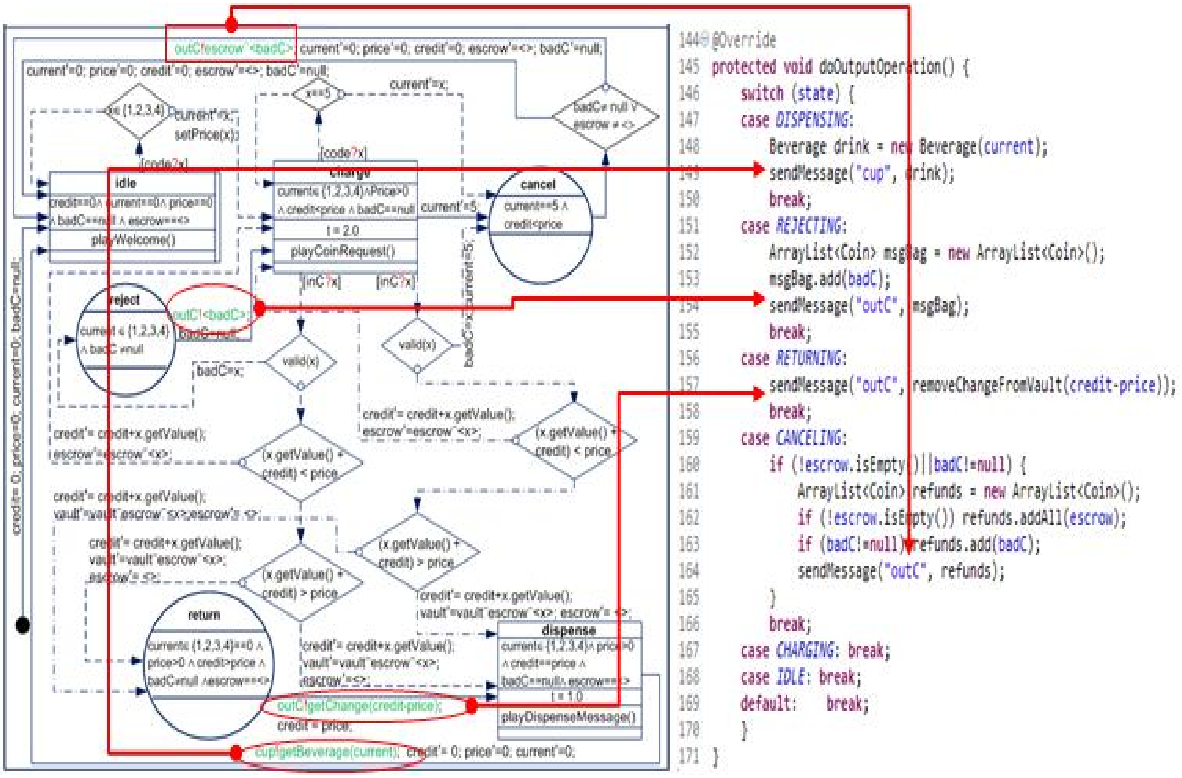


```
//[protected ('generate output operation')]
    @Override
    protected void doOutputOperation() {
    switch (state) {
    [let intTrans : Sequence(InternalTransition) = aHSystem.transitions->
                                                    filter(InternalTransition)->asSequence()]
    [let transWithOutput : Sequence(InternalTransition) = intTrans->
                                collect(t2:InternalTransition|t2.outputEvents->notEmpty())]
        [for (config : Configuration | aHSystem.configurations) separator ('\n')]
            case [config.label.toUpperCase()/]:
                [let trans : InternalTransition = transWithOutput->
                                                any(t:InternalTransition|t.source=config)]
                    [for (outEvent : Message | trans.outputEvents) separator ('\n')]
                        sendMessage("[outEvent.port.portDecl.declName/]",[outEvent.value/]);
                    [/for]
                [/let]
                break;
        [/for]
    [/let]
    [/let]
            default: break;
    }
[/protected]
//[protected ('generate activity function')]
    @Override
    protected void runActivities() {
        switch (state) {
        [for (config : Configuration | aHSystem.configurations) separator ('\n')]
            case [config.label.toUpperCase()/]:
                [if (config.activities->notEmpty())]
                    while (state==[config.label.toUpperCase()/]){
                        [for (activity : Expression | config.activities)][activity/];[/for]
                    }
                [/if]
                break;
        [/for]
            default: break;
        }
    }
[/protected]
```

Figure 7.31 Code generator segments for output and activity functions

For the implementation of method *runActivities*(), a switch statement is created with *state* as its case variable as usual and a case is created with the label of each HiLLS configuration. In each case, if the activity field of the associated configuration is not empty, a while loop is created within which the activity expressions are printed. The exit condition of the loop is set to make it continue while the configuration persists.

Figure 7.32 presents the correspondences between the implementation of *doOutputOperation*() for the BVM and the output operations specified in its HiLLS model to illustrate the expected output if the generator were applied.



outC!escrow^<badC>
current'=0; price'=0; credit'=0; escrow'=<>; badC'=null;
x∈{1,2,3,4}
current'=x; setPrice(x);
x==5
current'=x;
badC≠null ∨ escrow≠<>
[code?x]
idle
credit==0∧current==0∧price==0 ∧badC==null∧escrow==<>
playWelcome()
charge
current∈{1,2,3,4}∧Price>0 ∧credit<price∧badC==null
t=2.0
playCoinRequest()
current'=5;
cancel
current==5∧credit<price
[inC?x]
reject
current∈{1,2,3,4} ∧badC≠null
outC!<badC>
badC=null
badC'=x;
valid(x)
credit'=credit+x.getValue(); escrow'=escrow^<x>;
(x.getValue()+credit) < price
(x.getValue()+credit) > price
return
current∈{1,2,3,4}==0∧price>0∧credit>price∧badC=null∧escrow==<>
dispense
current∈{1,2,3,4}∧price>0 ∧credit==price∧badC==null∧escrow==<>
t=1.0
playDispenseMessage()
outC!getChange(credit-price);
credit'=price;
cup!getBeverage(current); credit'=0; price'=0; current'=0;
credit'=0; price=0; current=0; badC=null;

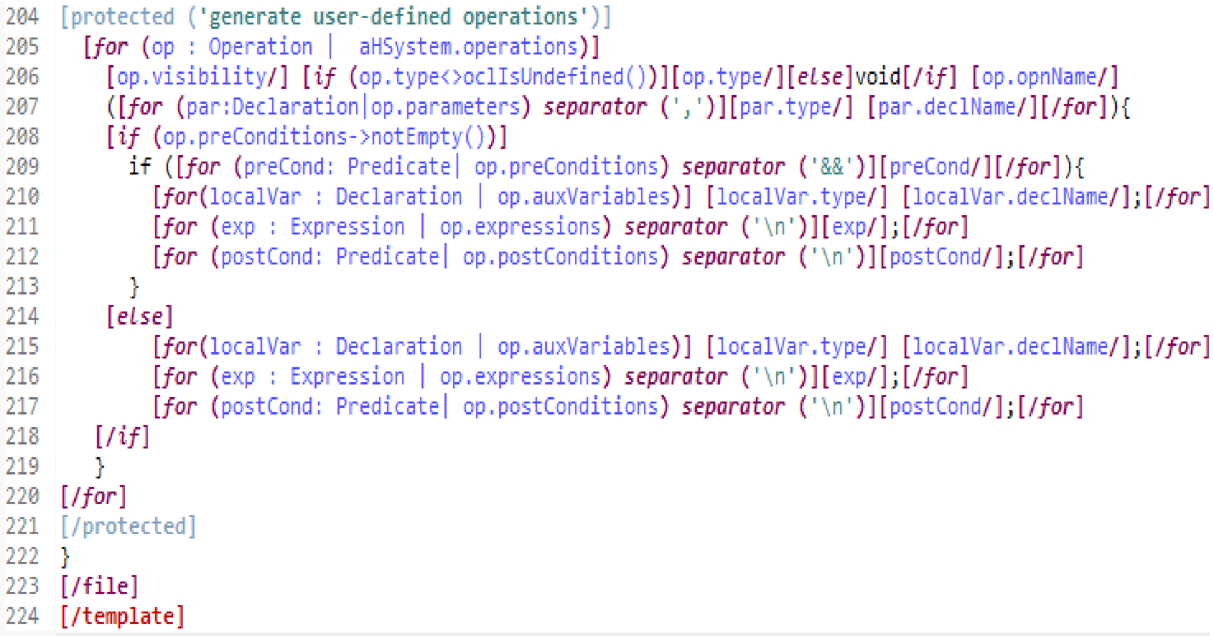
```
@Override
protected void doOutputOperation() {
    switch (state) {
    case DISPENSING:
        Beverage drink = new Beverage(current);
        sendMessage("cup", drink);
        break;
    case REJECTING:
        ArrayList<Coin> msgBag = new ArrayList<Coin>();
        msgBag.add(badC);
        sendMessage("outC", msgBag);
        break;
    case RETURNING:
        sendMessage("outC", removeChangeFromVault(credit-price));
        break;
    case CANCELING:
        if (!escrow.isEmpty()||badC!=null) {
            ArrayList<Coin> refunds = new ArrayList<Coin>();
            if (!escrow.isEmpty()) refunds.addAll(escrow);
            if (badC!=null) refunds.add(badC);
            sendMessage("outC", refunds);
        }
        break;
    case CHARGING: break;
    case IDLE: break;
    default:    break;
    }
}
```

Figure 7.32Correspondences between the enactment code of output operations in BVM and its HiLLS model

7.4.2.6 Code generator segments for output and activity functions of an atomic system class

Finally, on atomic HSystem, Figure 7.33 presents the segment of the code generator that translates a HiLLS operation to a Java method for special-purpose operations. Given HiLLS operation *op*, lines 206-207 define the signature of a Java method with the same *visibility* as *op*. the *type* of *op*, if defined, is extracted to define the type of the generated method; otherwise, type *void* is generated before extracting the *name* of *op* to generate the name of the method being synthesized. Line 207 retrieves the *parameters* of *op*, if any, to generate a comma-separated list of parameters in a parenthesis before opening the curved bracket for the method's body.

If *op* has preconditions, they are extracted to generate the condition of an "if" statement that encloses the method's body as specified in lines 208-214.



```
204 [protected ('generate user-defined operations')]
205   [for (op : Operation |  aHSystem.operations)]
206     [op.visibility/] [if (op.type<>oclIsUndefined())][op.type/][else]void[/if] [op.opnName/]
207     ([for (par:Declaration|op.parameters) separator (',')][par.type/] [par.declName/][/for]){
208     [if (op.preConditions->notEmpty())]
209       if ([for (preCond: Predicate| op.preConditions) separator ('&&')][preCond/][/for]){
210         [for(localVar : Declaration | op.auxVariables)] [localVar.type/] [localVar.declName/];[/for]
211         [for (exp : Expression | op.expressions) separator ('\n')][exp/];[/for]
212         [for (postCond: Predicate| op.postConditions) separator ('\n')][postCond/];[/for]
213       }
214     [else]
215         [for(localVar : Declaration | op.auxVariables)] [localVar.type/] [localVar.declName/];[/for]
216         [for (exp : Expression | op.expressions) separator ('\n')][exp/];[/for]
217         [for (postCond: Predicate| op.postConditions) separator ('\n')][postCond/];[/for]
218    [/if]
219    }
220 [/for]
221 [/protected]
222 }
223 [/file]
224 [/template]
```

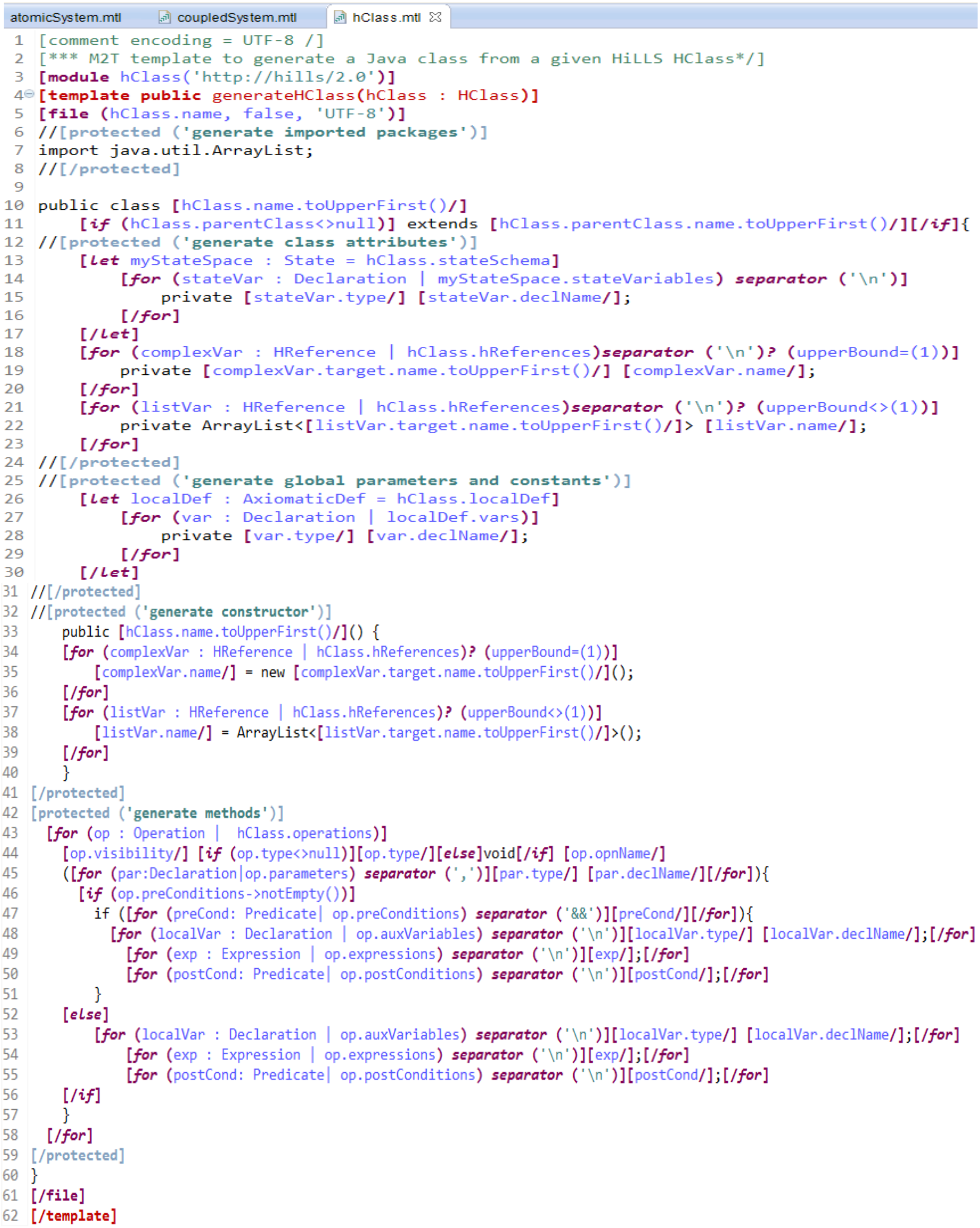
Figure 7.33 Code generator segment for translating HiLLS operations to methods

7.4.3 Enactment Semantics of HiLLS HClass

Intuitively, the enactment semantics of a HiLLS HClass is obtained by executing its software equivalent, which is a class. This provide support for the synthesis of appropriate program codes for the enactment (and simulation) of non-primitive input and output elements specified in HiLLS.

Since our enactment framework is Java-based, we generate an equivalent Java class for every HClass in a HiLLS specification. We do not need a special template to generate the class; the structure of the conventional Java class is sufficient.

Figure 7.34 presents the code generator to synthesize a Java class from a given HiLLS HClass *hClass*. The different segments are self-explanatory as they use the same constructs and structures presented previously in this section. Lines 12-31 generate the class' attributes and global variables if any, lines 32-41 generate the constructor while lines 42-59 generate the methods before closing the class at line 60.



```
[comment encoding = UTF-8 /]
[*** M2T template to generate a Java class from a given HiLLS HClass*/]
[module hClass('http://hills/2.0')]
[template public generateHClass(hClass : HClass)]
[file (hClass.name, false, 'UTF-8')]
//[protected ('generate imported packages')]
import java.util.ArrayList;
//[/protected]

public class [hClass.name.toUpperFirst()/]
    [if (hClass.parentClass<>null)] extends [hClass.parentClass.name.toUpperFirst()/][/if]{
//[protected ('generate class attributes')]
    [let myStateSpace : State = hClass.stateSchema]
        [for (stateVar : Declaration | myStateSpace.stateVariables) separator ('\n')]
            private [stateVar.type/] [stateVar.declName/];
        [/for]
    [/let]
    [for (complexVar : HReference | hClass.hReferences)separator ('\n')? (upperBound=(1))]
        private [complexVar.target.name.toUpperFirst()/] [complexVar.name/];
    [/for]
    [for (listVar : HReference | hClass.hReferences)separator ('\n')? (upperBound<>(1))]
        private ArrayList<[listVar.target.name.toUpperFirst()/]> [listVar.name/];
    [/for]
//[/protected]
//[protected ('generate global parameters and constants')]
    [let localDef : AxiomaticDef = hClass.localDef]
        [for (var : Declaration | localDef.vars)]
            private [var.type/] [var.declName/];
        [/for]
    [/let]
//[/protected]
//[protected ('generate constructor')]
    public [hClass.name.toUpperFirst()/]() {
    [for (complexVar : HReference | hClass.hReferences)? (upperBound=(1))]
        [complexVar.name/] = new [complexVar.target.name.toUpperFirst()/]();
    [/for]
    [for (listVar : HReference | hClass.hReferences)? (upperBound<>(1))]
        [listVar.name/] = ArrayList<[listVar.target.name.toUpperFirst()/]>();
    [/for]
    }
[/protected]
[protected ('generate methods')]
  [for (op : Operation |  hClass.operations)]
    [op.visibility/] [if (op.type<>null)][op.type/][else]void[/if] [op.opnName/]
    ([for (par:Declaration|op.parameters) separator (',')][par.type/] [par.declName/][/for]){
      [if (op.preConditions->notEmpty())]
        if ([for (preCond: Predicate| op.preConditions) separator ('&&')][preCond/][/for]){
          [for (localVar : Declaration | op.auxVariables) separator ('\n')][localVar.type/] [localVar.declName/];[/for]
            [for (exp : Expression | op.expressions) separator ('\n')][exp/];[/for]
            [for (postCond: Predicate| op.postConditions) separator ('\n')][postCond/];[/for]
        }
    [else]
        [for (localVar : Declaration | op.auxVariables) separator ('\n')][localVar.type/] [localVar.declName/];[/for]
            [for (exp : Expression | op.expressions) separator ('\n')][exp/];[/for]
            [for (postCond: Predicate| op.postConditions) separator ('\n')][postCond/];[/for]
    [/if]
    }
  [/for]
[/protected]
}
[/file]
[/template]
```

Figure 7.34 Code generator to translate HiLLS HClass to Java class

7.5 CONCLUSION

We have presented the HiLLS semantics in this chapter. As envisioned in the architecture of SimStudio II framework in Chapter 4, HiLLS has four semantics domains: DEVS for simulation-based analysis, Z and Temporal Logic for logic-based formal analysis and the enactment framework presented in Chapter 5 for enactment. Using model transformation techniques, we defined, in this chapter, the semantics mappings of HiLLS' abstract syntax onto the different semantics domains to take benefit of their respective semantics and supporting tools.

We used ATL, a model-to-model transformation language to define the HiLLS-to-DEVS and HiLLS-to-Z semantics mappings by defining model transformation rules between HiLLS metamodel and their respective metamodels. The DEVS-based enactment framework provides Java-based templates to write enactment models for DES; thus, we use the Acceleo MTL, a model-to-text transformation language to define the code generators that synthesize enactment codes, based on the framework's templates from a given HiLLS model.

The HiLLS model editor is not yet available to enable us validate the automated synthesis of the artifacts for simulation, formal analysis and enactment from a given HiLLS model as envisioned in the SimStudio II manifesto. Nevertheless, in accordance to the semantics mapping rules specified in each case, we demonstrated its feasibility by showing the correspondences between the elements of the HiLLS model of the running example of the thesis - the beverage vending system - and its manually written DEVS, Z and enactment models. These correspondences justify largely, our hypothesis in the beginning of the thesis that it is possible to have a unified high-level language, which will be expressive enough to integrate the essential concepts required for the three analysis methodologies. Hence, we believe that further research in this direction will be worthwhile in the end.

In addition to the development of a HiLLS model editor around which can be built the SimStudio II MDSE environment, we intend to address some of the limitations of the current semantics mapping rules in our future work. The two main limitations are: 1) the mapping rules can efficiently translate only simple predicates and expressions to the appropriate constructs in the target semantics domains; there may be the need for some transformation libraries that will recognize the pattern of a given predicate or expression and map it to the appropriate construct(s) in the target domain. 2) The transformation rules, in their current forms, cannot efficiently translate conditional branching in the HiLLS configuration transition diagrams to the appropriate data structures in the synthesized models. They are, however efficient in translating non-branching transitions. More work is required on this aspect to let the model transformers recognize the transition patterns and take appropriate actions.

8 GENERAL CONCLUSIONS

8.1 SUMMARY OF THE THESIS

This thesis explores the integration of Model-Driven Systems Engineering (MDSE) theories and technologies along three dimensions of design, analysis, and verification methodologies for Discrete Event Systems (DES): *Simulation*, *Formal Methods* (FM) and *Enactment*. The goal is to harness the synergy of diverse theories, tools, and expertise for complementary analyses of static and dynamic properties of complex DESs.

The design and development of a complex system may require an iterative process of modeling, performance evaluation, logical analysis for requirement verifications, and prototype implementation for run-time testing [HK06]. Such iterations of analysis processes are often necessary for early revelations of subtle knowledge about the systems, which are, in most cases, beyond intuition. An undesired behavior discovered in the analysis of a system can be a signal of a fundamental flaw in the system's design; such discovery must be made at an early stage of development to forestall costly errors in the final system.

Depending on the questions to be answered about the system under study, suitable MDSE approaches based on theoretically sound analysis methodologies like simulation, FM or enactment are employed in the iteration loops to mine the desired knowledge from models of the system. More than one of the three methodologies - simulation, FM, and enactment - are often required for complementary studies of different aspects of complex systems; in such cases, the combined methodologies are used to reason about the system's models from divergent viewpoints to provide answers that complement one another. This thesis aims to exploit the benefits of MDSE techniques and tools to put the three methodologies together under the umbrella of a unified high-level viewpoint to make them accessible to non-experienced users as well as ease the tasks of experienced users.

MDSE is a discipline that applies Model-Driven Engineering (MDE) practices to automate processes in the Model-Based Systems Engineering (MBSE) paradigm [MM13]. MBSE is "the formalized application of modeling to support system requirements, design, analysis, verification, and validation activities beginning in the conceptual design phase and continuing throughout development and later life cycle″ [Est07]. MDSE, particularly, aims at concretizing the envisioned benefits of MBSE through the applications of metamodeling and automated model transformations for more productive and effective uses of models in the systems engineering domain [BD14].

In recent years, the systematic combinations of disparate MDSE approaches to maximize the synergy between the different disciplines have been growing in importance both in industry and in academia; this is evidenced by the volume of work published on the topic within the last decade. Examples (non-exhaustive) of such work, with preference for DEVS-based simulation methodology and FM with Z and Temporal Logic, include [Tra08, TTH11, Tou12, Shu11,TH14, BD14, Cri07, Cri08, TFH09, MWB+13, MDL+14,TB15].Nevertheless, much research efforts are still needed to bring this idea and its benefits to the fore.

8.1.1 Problems Addressed and Research Questions

8.1.1.1 Lack of requisite logic and mathematical skills to deal with most formalisms

The problem of lack of requisite mathematical skills to deal with the underlying formalisms of the different analysis methodologies on the part of domain experts, which has always been an inhibitor to the wide adoption of each of the methodologies, is being continuously addressed with MDSE approaches. Essentially, this involves the provision of high-level notations for model creation, and the automated synthesis, from high-level models, of the low-level artifacts required by the analysis tools. However, this problem is often solved for each methodology in isolation; little progress has been made in addressing the problem collectively for different methodologies.

8.1.1.2 Little chances of portability of models between computational analysis methodologies

Apparently, due to the disparate purposes for which their underlying formalisms have been created, and the difference in the sets of concepts expressed in such formalisms, there are usually little chances of portability of models between different analysis methodologies. The implication of this situation is that a complementary application of multiple methodologies to study different aspects of a system will require manual, or at best semi-automated, creation and updating of separate models, in different formalisms, of yet the same system to answer the different questions of the different stakeholders. This task can be herculean and error-prone.

8.1.1.3 Little coexistence of disparate methodologies in the same environment

In most cases, a computational analysis environment is built to support a specific analysis methodology; as such, extending it to accommodate other methodologies can be very difficult, if not impossible. Unfortunately, none of the methodologies is guaranteed to provide answers to all questions about the different aspects of a system. Thus, for an exhaustive analysis of system's properties, the analyst must face the challenge of maintaining consistencies between the different models, having multiple disconnected views of the same system, in separate MDSE environments. The authors of [BSD+12] have acknowledged that this kind of situation has the potential to create miscommunication among the different teams involved in the development of a system.

In our attempt to propose solutions to these problems, we formulated the following research questions, to which we tried to provide answers in this thesis:

RQ1. Is it possible to build an integrative framework that can be continuously populated with best practices in MDSE for simulation, formal methods and enactment such that the various components are federated through a seamless sharing of high-level system model?

RQ2. Which formalism should we adopt to write the shared model?

RQ3. How can the disparate concerns of the different methodologies be captured in the so-called unified model

RQ4. In what order should the process of the different methodologies be executed?

By these research questions, we envisioned an integrative MDSE framework that has the infrastructure to accommodate legacy tools for simulation, formal analysis, and enactment such that the disparate tools get their synthesized artifacts from one unified and consistent model of the system under study. Another important requirement is that the framework must provide high-level notations to create and edit the shared model as well as communicate it among the stakeholders.

8.1.2 Contributions of the Thesis

The main contributions of the thesis to provide answers to the research questions and by extension, proffer solutions to the identified problems are:

8.1.2.1 A multi-layered framework that emulates the Model-Driven Architecture (MDA) by defining a unified model specification layer on top of the layers containing the specific analysis methodologies:

In Chapter 4, we proposed a methodological framework, called SimStudio II, as an answer to answering RQ1. SimStudio II has a multi-level architecture, which emulates a cascaded MDA, with MDSE tools and artifactsat the different layers. Following the MDA principle, SimStudio II architecture can be described as a cascade of two MDA-like architectures.

The architecture at the top has two layers for *Methodology-Independent Model* (MIM) and *Methodology-Specific Model* (MSM). MIM refers to a model of the system under study and its requirements, which is not specifically dedicated to any of the three target analysis methodology but contains the information necessary to synthesize the artifacts required by the tools for each methodology. The MSMs consists of the models of the system under study and its requirements, which are target specific analysis methodologies and must be synthesized from the MIM at the topmost layer. The essence is to ensure that only MIM is specified manually and used to drive the syntheses of the MSMs.

The MDA-like architecture at the bottom of the cascaded architectures takes the system models among the MIMSs as the conventional MDA's PIMs (*Platform-Independent Model*) and the requirement model among them as conventional MDA's CIMs (*Computational-Independent Model*). From there, it generates the PSMs (*Platform-Specific Model*) for the different analysis tools available. This architecture makes it possible to share system models among new and legacy tools of the same and different analysis methodologies.

The architecture of the proposed framework is accompanied by a process model, as an answer to RQ4, which describes the workflow to be followed as a guide to using the framework.

8.1.2.2 A preliminary framework for the enactment of DES

Enactment methodology has yet to permeate significantly into the MDSE practice with DES unlike simulation and formal analysis which both have well established formalisms and operational/logical protocols that are accepted by considerably large communities. The current practices of enactment for DES are mostly based on UML and SysML (System Modeling Language) and their profiles.

In Chapter 5, we proposed a DEVS-based enactment framework for DES. The framework adopts and extends the DEVS syntax to express DESs but uses the behavior of the object-oriented observer design pattern to define enactment protocol for the analysis of functional and operational properties of a DES through the execution of its software prototype using the physical clock time for the scheduling and execution of events.

8.1.2.3 A high level language whose syntax uniformly combines the DES concepts for simulation, FM and enactment to support the specification of unified models for the three methodologies

To the best of our knowledge, no formalism existed in the literature that could be used to model the MIM described in Section 8.1.2.1above to serve as the kernel of the SimStudio II architecture. Hence, we proposed the *High Level Language for System Specification* (HiLLS) which must be expressive enough to subsume the formalisms of the three methodologies and provide high level notations for the users to create and edit models.

In Chapter6, we proposed the abstract and concrete syntaxes of HiLLS to provide answers to research questions RQ2 and RQ3.

To define the HiLLS' abstract syntax, we used metamodel integration techniques for a disciplined integration of DES concepts adopted from considerably universal formalisms namely: DEVS, Object-Z, UML and Temporal Logic (TL) into one unified language, which is suitable to create and edit the MIM as describe previously.

We adopted and extended some notations from the UML family of languages and the Z schema to define the HiLLS concrete syntax to facilitate its learning by prospective users.

In addition to high-level notations for system modeling, the HiLLS' concrete syntax also provides high-level notations, which are similar to the notations for expressing system behavior, to express the temporal properties that must hold in the system model. We believe that using similar high level notations to model systems and their logical requirements will enhance the adoption of FM by simulation practitioners to complement their simulation results. We have not found in the literature a related that combines DEVS-based simulation with logic-based modeling in the same manner at the time of writing this thesis. We hope this will stimulate further research that direction.

8.1.2.4 Formal mappings of HiLLS concepts to simulation, FM and enactment semantics domains

To consolidate the previous contributions, we proposed, in Chapter 7, the semantics of HiLLS. In accordance with the vision of the SimStudio II framework, we provided a set of three translational semantics for HiLLS, using model transformation techniques. Given a HiLLS model of a system, we defined the model transformation rules to automate the synthesis of DEVS models for simulation, Z specifications for formal analysis, and Java-based enactment codes for enactment, using the enactment framework we proposed.

8.2 PERSPECTIVES

We believe that the results obtained from this thesis sufficiently demonstrate the feasibility of our vision for an integrative framework within which simulation-, FM-, and enactment-based analysis methodologies, can co-exist and co-evolve for complementary analyses of different aspects of DESs. However, there is still a long road ahead to project the main ideas into the reach of potential users. Nevertheless, we are motivated by the potential benefits of the work, in the long term, to continue the efforts to deal with unresolved issues as well as new ones that may arise.

First, we are aware that a concrete software environment that implements the SimStudio II architecture is necessary to enable potential users try the framework and provide feedbacks for further improvements. There is an ongoing work, in our research group, towards the formal specification of both the textual and the graphical elements of the HiLLS' concrete syntax, and eventually, the development of an Eclipse-based drag-and-drop editor for the language. With that, we intend to take advantage of the MDE infrastructure in the Eclipse platform to integrate HiLLS with the other artifacts specified in the SimStudio II architecture.

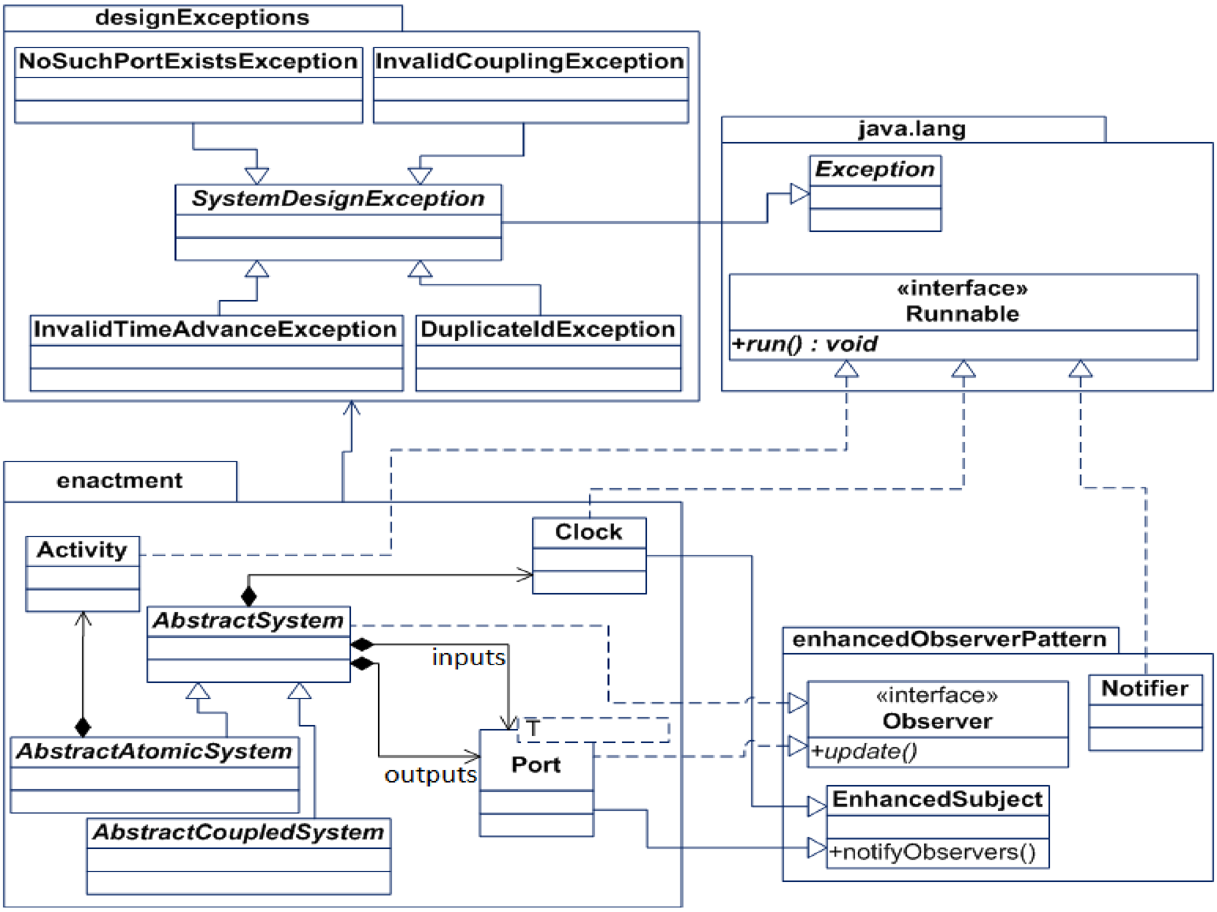
We identified some areas that need refinements in the semantics mapping rules we presented in Chapter 7. Particularly, the HiLLS to Z mapping in Section 7.3.2and the code generator in Section7.4, in their current states, cannot recognize the formats of different kinds of predicates and expressions in the HiLLS model to generate the most suitable constructs in the target models. Similarly, the mapping rules to translate HiLLS configuration transitions in both cases need to be improved to recognize the condition nodes along the transition paths and generate the most suitable constructs in the target models. We intend to address these limitations in our future work.

The TL concepts in the HiLLS metamodel were adopted as is from the metamodel we proposed for the TL property patterns in Section 6.2.3. Hence, we did not provide explicit mapping rules to generate the properties from a HiLLS requirement model since the relations between them is bijective. We intend to identify suitable TL-based tools to define the low-level artifacts for the different TL patterns and generate the codes directly from HiLLS.

Finally, the aforementioned limitations and plans for future work are just some of those that are obvious now. We expect that more questions will still arise, leading to the discovery of some interesting research directions as we go deeper into the different aspects of the work.

Appendix A: Java Implementation of the DEVS-Based Enactment Framework

This appendix documents the Java implementation of the DEVS-based enactment framework presented in Chapter 5. As a reminder, we re-present the design package diagram of the implementation. Next, we will document the implementations of the elements of packages *desginException*, and *enhancedObserverPattern* under separate headings. We have presented the elements of package *enactment* in Section5.4.3.Package *java.lang* refers to the actual Java packages, which we re-use; hence, we provide no implementation for that.



A.1 Package "enhancedObserverPattern "

Class EnhancedSubject

```java
package enhancedObserverPattern;
import java.util.ArrayList;
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;
/*****************************************************************************************
 * EnhancedSubject.java
 * This class is similar to the 'Subject' class of the conventional observer pattern
 * except that the notification of observers is delegated to concurrent asynchronous
 * 'Notifier' threads
 * @author H. O. ALIYU
 * **************************************************************************************/
public class EnhancedSubject {
  private ArrayList<Observer> observers = new ArrayList<Observer>();
  private boolean changedFlag;
  protected final void notifyObservers(){
    if (hasChanged()){
      ExecutorService postMaster = Executors.newCachedThreadPool();
      for(Observer obs:observers){
        postMaster.execute(new Notifier(this, obs));
      }   postMaster.shutdown();
      unsetChanged();
    }
  }
  public final void addObserver(Observer observer){
    observers.add(observer);
  }
  public final void dropObserver(Observer observer){
    observers.remove(observer);
  }
  protected final void setChanged(){
    changedFlag = true;
  }
  private void unsetChanged(){
    changedFlag = false;
  }
  private boolean hasChanged(){
    return changedFlag;
  }
}
```

Interface Observer

```java
package enhancedObserverPattern;

/*****************************************************************************************
 * Observer.java
 * This interface declares the abstract method "update" that takes an EnhancedSubject
 * object as argument.
 * @author H. O. ALIYU
 * **************************************************************************************/
public interface Observer {

  public abstract void update(EnhancedSubject subject);

}
```

Class Notifier

```java
package enhancedObserverPattern;
/***************************************************************************************
 * Notifier.java
 * This class implements the asynchronous notification of an observer
 * @author H. O. ALIYU
 * ************************************************************************************/
public class Notifier implements Runnable{
  private EnhancedSubject subject;
  private Observer observer;

  public Notifier(EnhancedSubject subject, Observer observer){
    this.subject = subject;
    this.observer = observer;
  }
  @Override
  public void run() {
    observer.update(subject);
  }
}
```

A.2 Package "desginException "

Class SystemDesignException

```
package enactment.designExceptions;
/****************************************************************************************
 * SystemDesignException.java
 * Abstract Exception for the framework
 * @author H. O. ALIYU
 * *************************************************************************************/
public abstract class SystemDesignException extends Exception {
  private static final long serialVersionUID = 1L;
  private String errorMessage;

  public SystemDesignException(String errMsg) {
    errorMessage = errMsg;
  }
  public String getErrorMessage(){
    return errorMessage;
  }
}
```

Class DuplicateIdException

```
package enactment.designExceptions;
/****************************************************************************************
 * DuplicateIdException.java
 * Exception thrown when an enactment model attempts to do any of the following
 * 1. Register multiple input/output port with duplicate port names in the same system
 * 2. Register multiple components with duplicate names in a coupled system model
 * @author H. O. ALIYU
 * *************************************************************************************/
public class DuplicateIdException extends SystemDesignException {
  private static final long serialVersionUID = 1L;

  public DuplicateIdException(String msg) {
    super(msg);
  }
}
```

Class InvalidTimeAdvanceException

```java
package enactment.designExceptions;
/*****************************************************************************************
 * InvalidTimeAdvanceException.java
 * Exception thrown when a negative time advance is computed for/assigned to a state
 * @author H. O. ALIYU
 * *************************************************************************************/
public class InvalidTimeAdvanceException extends SystemDesignException {
  private static final long serialVersionUID = 1L;

  public InvalidTimeAdvanceException(String msg) {
    super(msg);
  }
}
```

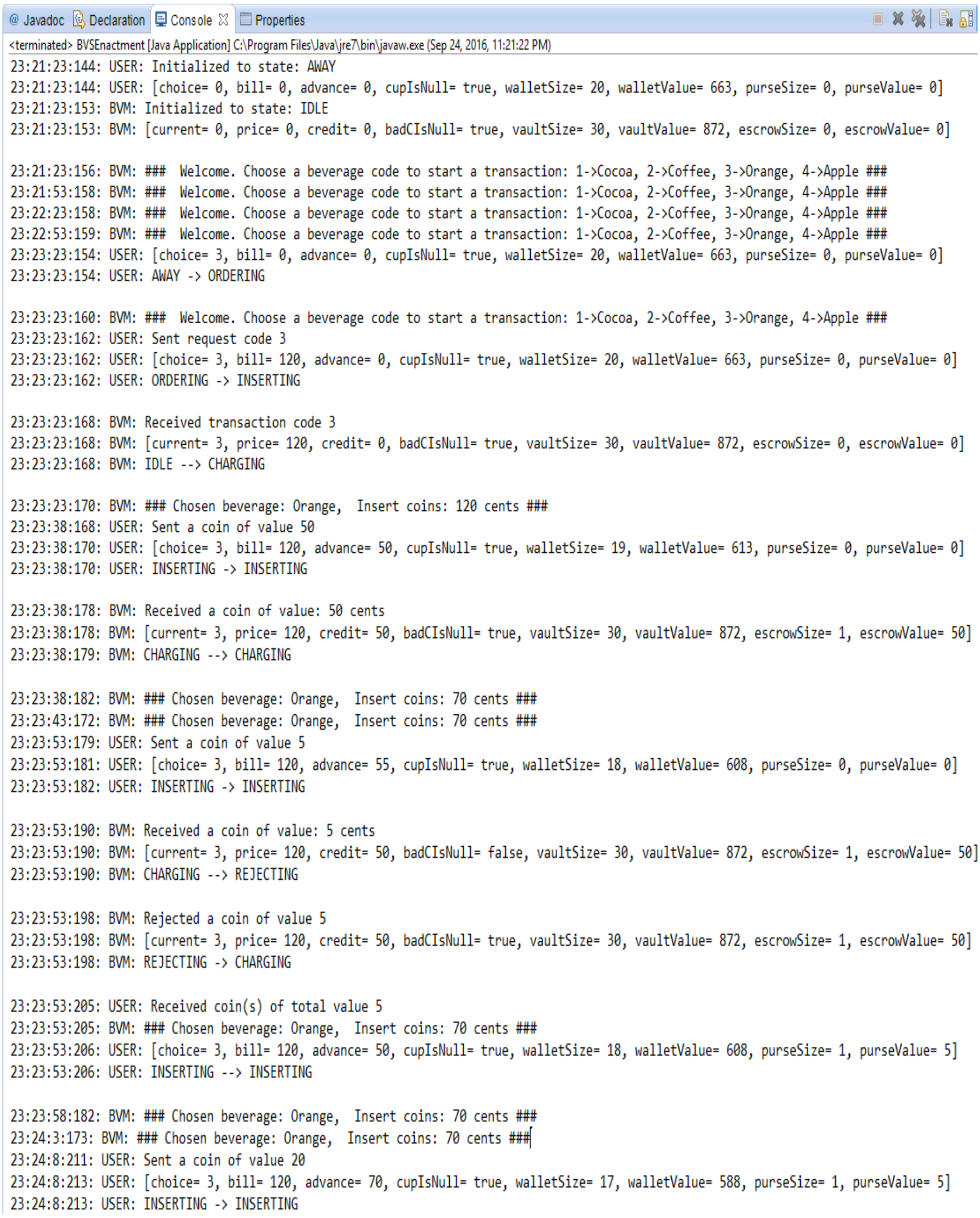
Class InvalidCouplingException

```java
package enactment.designExceptions;
/*****************************************************************************************
 * InvalidCouplingException.java
 * Exception thrown when a coupling specification violates any of the coupling
    constraints
 * @author H. O. ALIYU
 * *************************************************************************************/
public class InvalidCouplingException extends SystemDesignException {
  private static final long serialVersionUID = 1L;

  public InvalidCouplingException(String msg) {
    super(msg);
  }
}
```

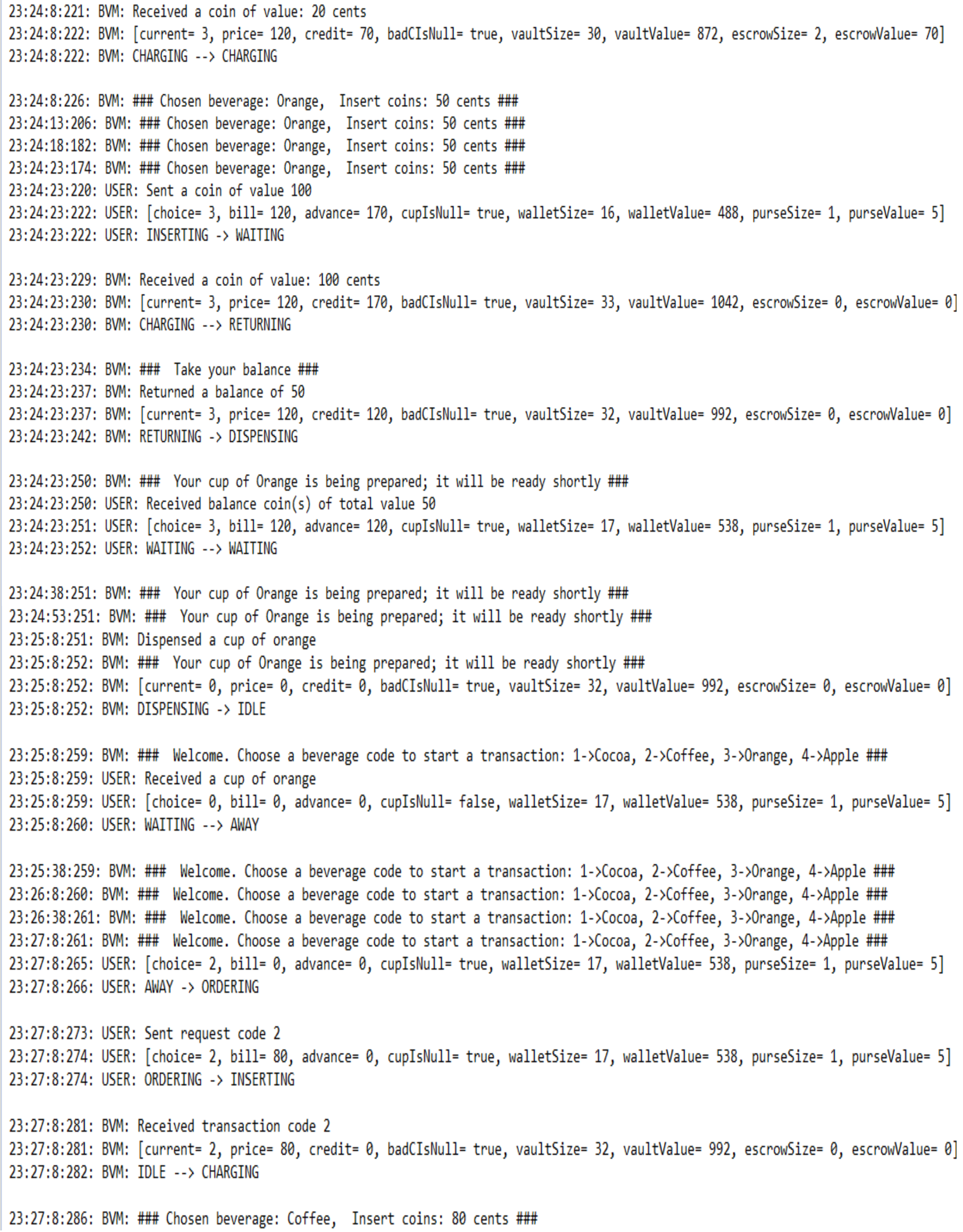
Class NoSuchPortExistsException

```java
package enactment.designExceptions;
/*****************************************************************************************
 * NoSuchPortExistsException.java
 * Exception thrown when a reference is made to a port name that does not exist
 * @author H. O. ALIYU
 * *************************************************************************************/
public class NoSuchPortExistsException extends SystemDesignException {
  private static final long serialVersionUID = 1L;

  public NoSuchPortExistsException(String errMsg) {
    super(errMsg);
  }
}
```

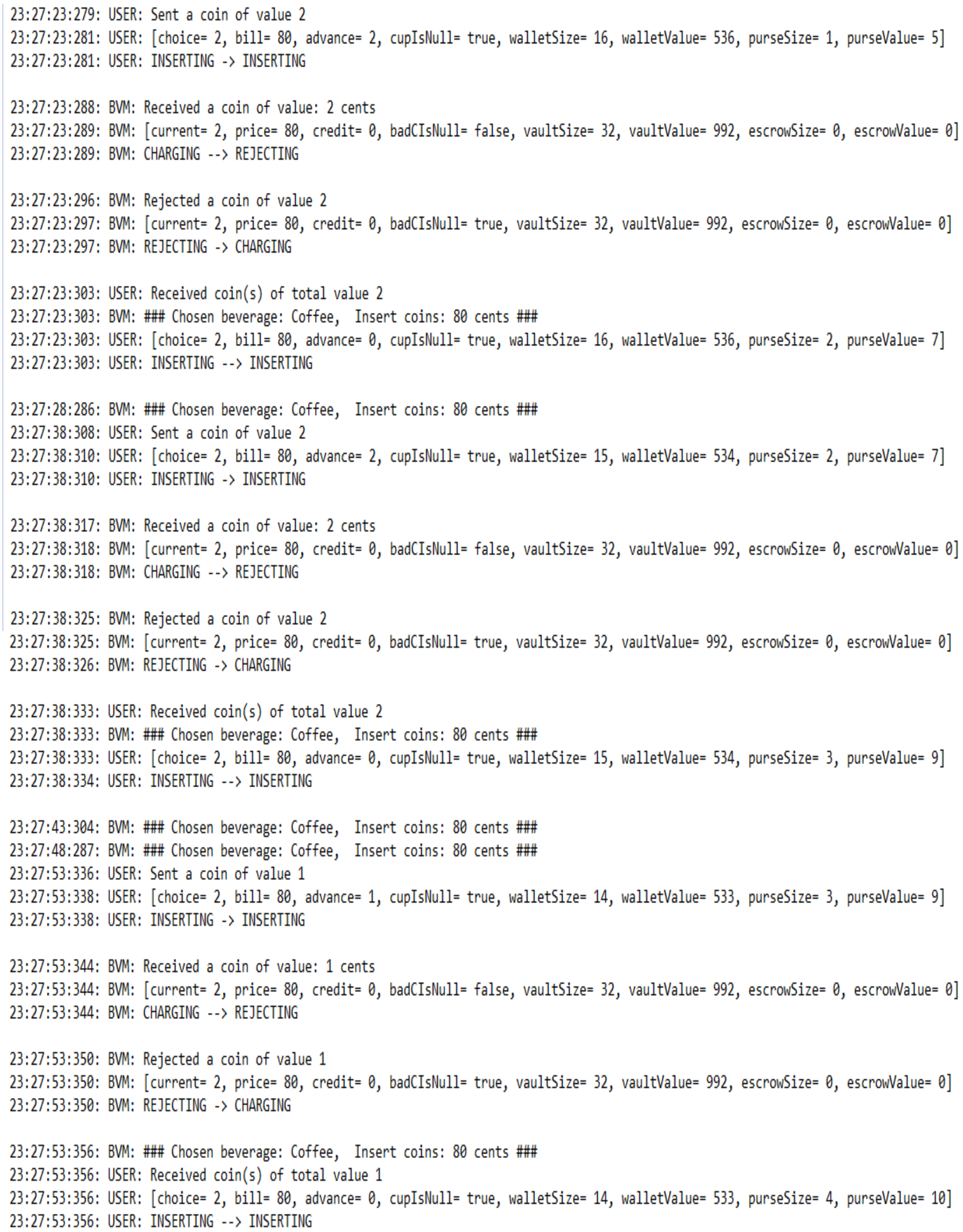
Appendix B: Enactment traces of the BVS



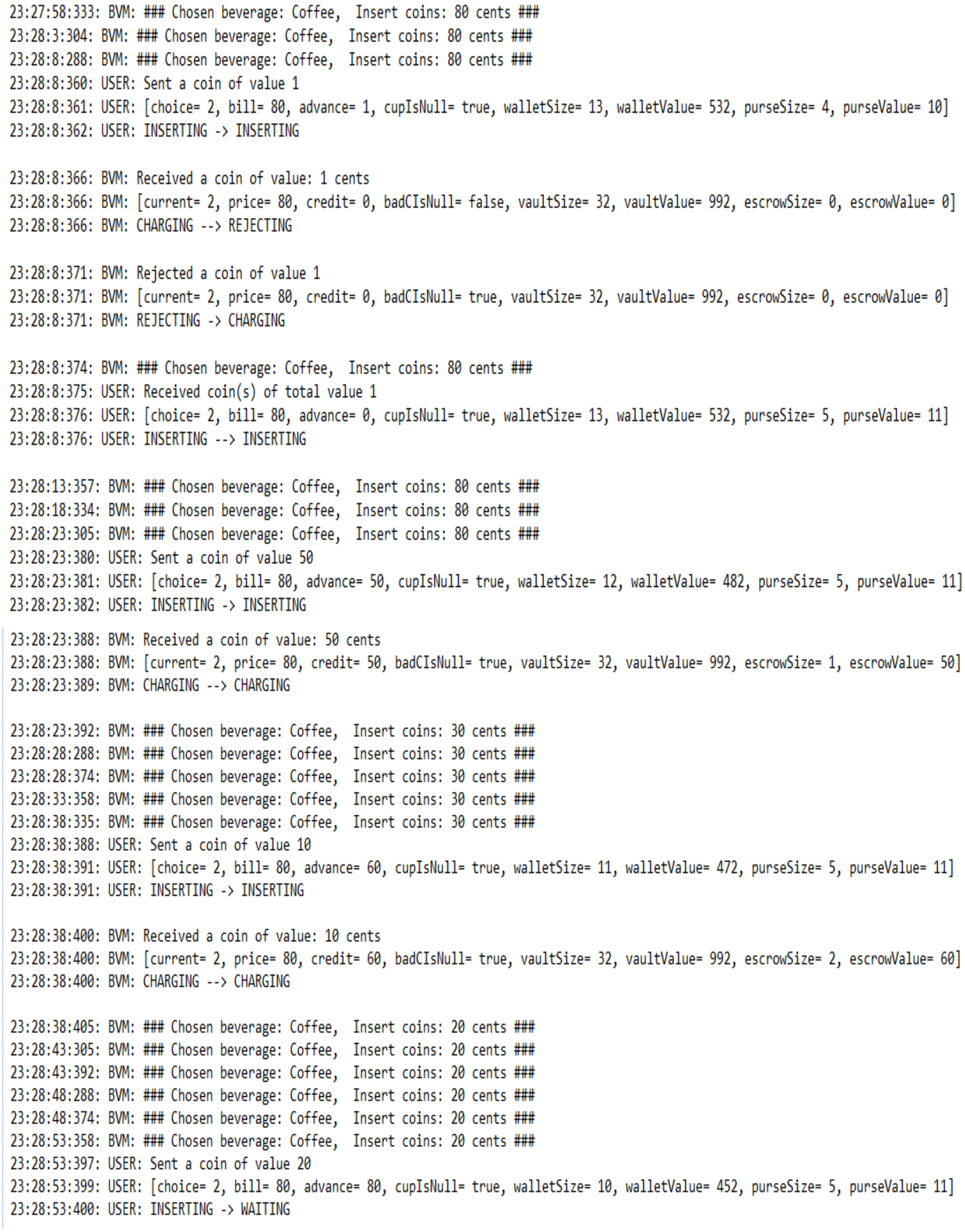
```
<terminated> BVSEnactment [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Sep 24, 2016, 11:21:22 PM)
23:21:23:144: USER: Initialized to state: AWAY
23:21:23:144: USER: [choice= 0, bill= 0, advance= 0, cupIsNull= true, walletSize= 20, walletValue= 663, purseSize= 0, purseValue= 0]
23:21:23:153: BVM: Initialized to state: IDLE
23:21:23:153: BVM: [current= 0, price= 0, credit= 0, badCIsNull= true, vaultSize= 30, vaultValue= 872, escrowSize= 0, escrowValue= 0]

23:21:23:156: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:21:53:158: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:22:23:158: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:22:53:159: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:23:23:154: USER: [choice= 3, bill= 0, advance= 0, cupIsNull= true, walletSize= 20, walletValue= 663, purseSize= 0, purseValue= 0]
23:23:23:154: USER: AWAY -> ORDERING

23:23:23:160: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:23:23:162: USER: Sent request code 3
23:23:23:162: USER: [choice= 3, bill= 120, advance= 0, cupIsNull= true, walletSize= 20, walletValue= 663, purseSize= 0, purseValue= 0]
23:23:23:162: USER: ORDERING -> INSERTING

23:23:23:168: BVM: Received transaction code 3
23:23:23:168: BVM: [current= 3, price= 120, credit= 0, badCIsNull= true, vaultSize= 30, vaultValue= 872, escrowSize= 0, escrowValue= 0]
23:23:23:168: BVM: IDLE --> CHARGING

23:23:23:170: BVM: ### Chosen beverage: Orange,  Insert coins: 120 cents ###
23:23:38:168: USER: Sent a coin of value 50
23:23:38:170: USER: [choice= 3, bill= 120, advance= 50, cupIsNull= true, walletSize= 19, walletValue= 613, purseSize= 0, purseValue= 0]
23:23:38:170: USER: INSERTING -> INSERTING

23:23:38:178: BVM: Received a coin of value: 50 cents
23:23:38:178: BVM: [current= 3, price= 120, credit= 50, badCIsNull= true, vaultSize= 30, vaultValue= 872, escrowSize= 1, escrowValue= 50]
23:23:38:179: BVM: CHARGING --> CHARGING

23:23:38:182: BVM: ### Chosen beverage: Orange,  Insert coins: 70 cents ###
23:23:43:172: BVM: ### Chosen beverage: Orange,  Insert coins: 70 cents ###
23:23:53:179: USER: Sent a coin of value 5
23:23:53:181: USER: [choice= 3, bill= 120, advance= 55, cupIsNull= true, walletSize= 18, walletValue= 608, purseSize= 0, purseValue= 0]
23:23:53:182: USER: INSERTING -> INSERTING

23:23:53:190: BVM: Received a coin of value: 5 cents
23:23:53:190: BVM: [current= 3, price= 120, credit= 50, badCIsNull= false, vaultSize= 30, vaultValue= 872, escrowSize= 1, escrowValue= 50]
23:23:53:190: BVM: CHARGING --> REJECTING

23:23:53:198: BVM: Rejected a coin of value 5
23:23:53:198: BVM: [current= 3, price= 120, credit= 50, badCIsNull= true, vaultSize= 30, vaultValue= 872, escrowSize= 1, escrowValue= 50]
23:23:53:198: BVM: REJECTING -> CHARGING

23:23:53:205: USER: Received coin(s) of total value 5
23:23:53:205: BVM: ### Chosen beverage: Orange,  Insert coins: 70 cents ###
23:23:53:206: USER: [choice= 3, bill= 120, advance= 50, cupIsNull= true, walletSize= 18, walletValue= 608, purseSize= 1, purseValue= 5]
23:23:53:206: USER: INSERTING --> INSERTING

23:23:58:182: BVM: ### Chosen beverage: Orange,  Insert coins: 70 cents ###
23:24:3:173: BVM: ### Chosen beverage: Orange,  Insert coins: 70 cents ###
23:24:8:211: USER: Sent a coin of value 20
23:24:8:213: USER: [choice= 3, bill= 120, advance= 70, cupIsNull= true, walletSize= 17, walletValue= 588, purseSize= 1, purseValue= 5]
23:24:8:213: USER: INSERTING -> INSERTING
```



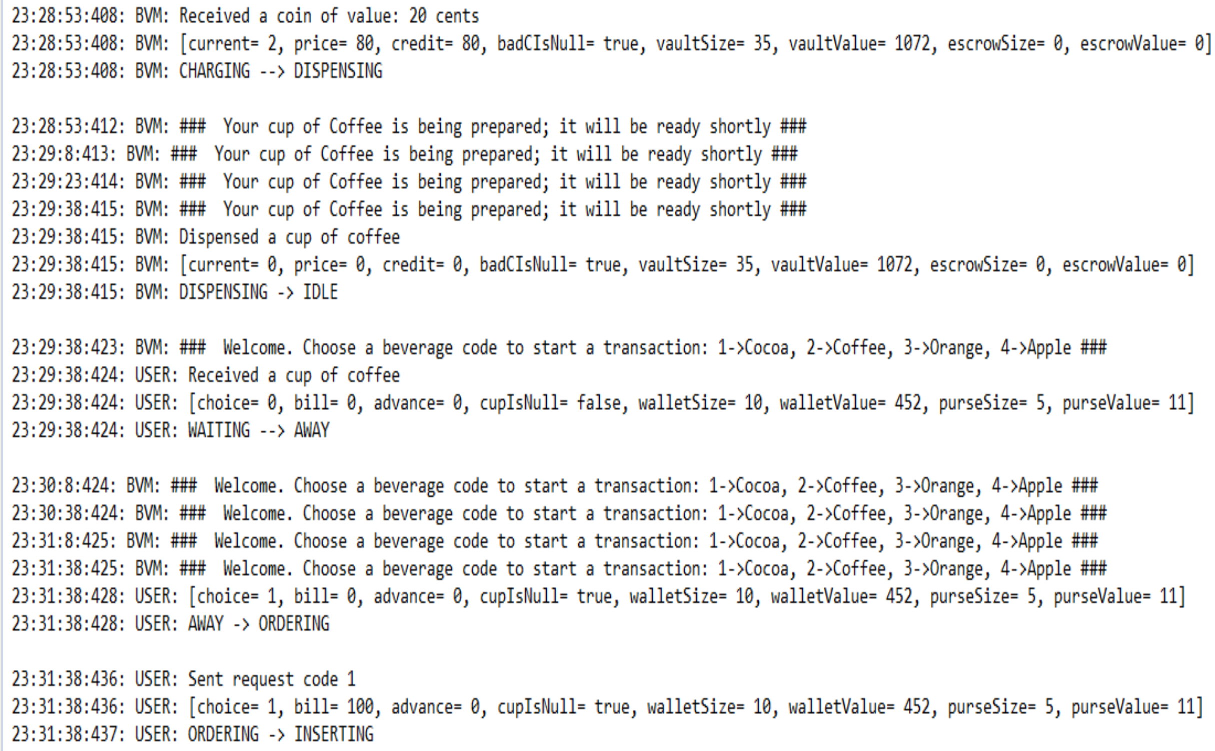
```
23:24:8:221: BVM: Received a coin of value: 20 cents
23:24:8:222: BVM: [current= 3, price= 120, credit= 70, badCIsNull= true, vaultSize= 30, vaultValue= 872, escrowSize= 2, escrowValue= 70]
23:24:8:222: BVM: CHARGING --> CHARGING

23:24:8:226: BVM: ### Chosen beverage: Orange,  Insert coins: 50 cents ###
23:24:13:206: BVM: ### Chosen beverage: Orange,  Insert coins: 50 cents ###
23:24:18:182: BVM: ### Chosen beverage: Orange,  Insert coins: 50 cents ###
23:24:23:174: BVM: ### Chosen beverage: Orange,  Insert coins: 50 cents ###
23:24:23:220: USER: Sent a coin of value 100
23:24:23:222: USER: [choice= 3, bill= 120, advance= 170, cupIsNull= true, walletSize= 16, walletValue= 488, purseSize= 1, purseValue= 5]
23:24:23:222: USER: INSERTING -> WAITING

23:24:23:229: BVM: Received a coin of value: 100 cents
23:24:23:230: BVM: [current= 3, price= 120, credit= 170, badCIsNull= true, vaultSize= 33, vaultValue= 1042, escrowSize= 0, escrowValue= 0]
23:24:23:230: BVM: CHARGING --> RETURNING

23:24:23:234: BVM: ###  Take your balance ###
23:24:23:237: BVM: Returned a balance of 50
23:24:23:237: BVM: [current= 3, price= 120, credit= 120, badCIsNull= true, vaultSize= 32, vaultValue= 992, escrowSize= 0, escrowValue= 0]
23:24:23:242: BVM: RETURNING -> DISPENSING

23:24:23:250: BVM: ###  Your cup of Orange is being prepared; it will be ready shortly ###
23:24:23:250: USER: Received balance coin(s) of total value 50
23:24:23:251: USER: [choice= 3, bill= 120, advance= 120, cupIsNull= true, walletSize= 17, walletValue= 538, purseSize= 1, purseValue= 5]
23:24:23:252: USER: WAITING --> WAITING

23:24:38:251: BVM: ###  Your cup of Orange is being prepared; it will be ready shortly ###
23:24:53:251: BVM: ###  Your cup of Orange is being prepared; it will be ready shortly ###
23:25:8:251: BVM: Dispensed a cup of orange
23:25:8:252: BVM: ###  Your cup of Orange is being prepared; it will be ready shortly ###
23:25:8:252: BVM: [current= 0, price= 0, credit= 0, badCIsNull= true, vaultSize= 32, vaultValue= 992, escrowSize= 0, escrowValue= 0]
23:25:8:252: BVM: DISPENSING -> IDLE

23:25:8:259: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:25:8:259: USER: Received a cup of orange
23:25:8:259: USER: [choice= 0, bill= 0, advance= 0, cupIsNull= false, walletSize= 17, walletValue= 538, purseSize= 1, purseValue= 5]
23:25:8:260: USER: WAITING --> AWAY

23:25:38:259: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:26:8:260: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:26:38:261: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:27:8:261: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:27:8:265: USER: [choice= 2, bill= 0, advance= 0, cupIsNull= true, walletSize= 17, walletValue= 538, purseSize= 1, purseValue= 5]
23:27:8:266: USER: AWAY -> ORDERING

23:27:8:273: USER: Sent request code 2
23:27:8:274: USER: [choice= 2, bill= 80, advance= 0, cupIsNull= true, walletSize= 17, walletValue= 538, purseSize= 1, purseValue= 5]
23:27:8:274: USER: ORDERING -> INSERTING

23:27:8:281: BVM: Received transaction code 2
23:27:8:281: BVM: [current= 2, price= 80, credit= 0, badCIsNull= true, vaultSize= 32, vaultValue= 992, escrowSize= 0, escrowValue= 0]
23:27:8:282: BVM: IDLE --> CHARGING

23:27:8:286: BVM: ### Chosen beverage: Coffee,  Insert coins: 80 cents ###
```



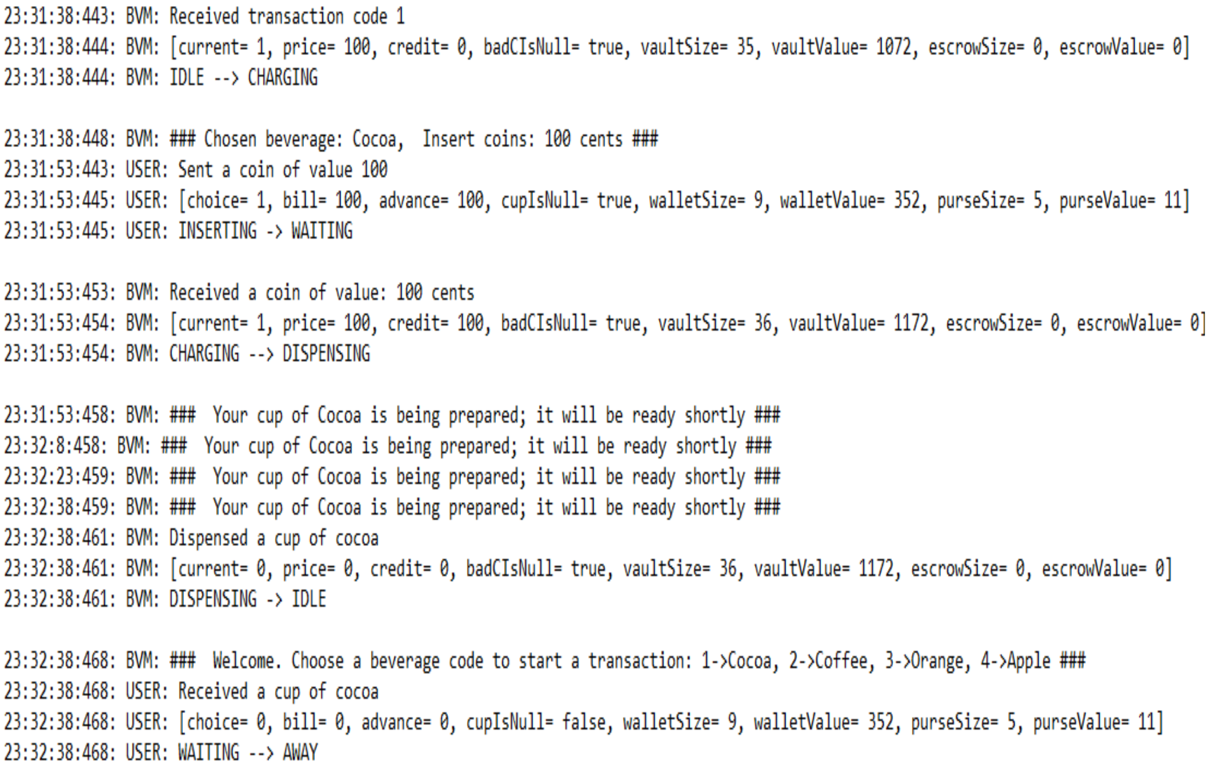
```
23:27:23:279: USER: Sent a coin of value 2
23:27:23:281: USER: [choice= 2, bill= 80, advance= 2, cupIsNull= true, walletSize= 16, walletValue= 536, purseSize= 1, purseValue= 5]
23:27:23:281: USER: INSERTING -> INSERTING

23:27:23:288: BVM: Received a coin of value: 2 cents
23:27:23:289: BVM: [current= 2, price= 80, credit= 0, badCIsNull= false, vaultSize= 32, vaultValue= 992, escrowSize= 0, escrowValue= 0]
23:27:23:289: BVM: CHARGING --> REJECTING

23:27:23:296: BVM: Rejected a coin of value 2
23:27:23:297: BVM: [current= 2, price= 80, credit= 0, badCIsNull= true, vaultSize= 32, vaultValue= 992, escrowSize= 0, escrowValue= 0]
23:27:23:297: BVM: REJECTING -> CHARGING

23:27:23:303: USER: Received coin(s) of total value 2
23:27:23:303: BVM: ### Chosen beverage: Coffee,  Insert coins: 80 cents ###
23:27:23:303: USER: [choice= 2, bill= 80, advance= 0, cupIsNull= true, walletSize= 16, walletValue= 536, purseSize= 2, purseValue= 7]
23:27:23:303: USER: INSERTING --> INSERTING

23:27:28:286: BVM: ### Chosen beverage: Coffee,  Insert coins: 80 cents ###
23:27:38:308: USER: Sent a coin of value 2
23:27:38:310: USER: [choice= 2, bill= 80, advance= 2, cupIsNull= true, walletSize= 15, walletValue= 534, purseSize= 2, purseValue= 7]
23:27:38:310: USER: INSERTING -> INSERTING

23:27:38:317: BVM: Received a coin of value: 2 cents
23:27:38:318: BVM: [current= 2, price= 80, credit= 0, badCIsNull= false, vaultSize= 32, vaultValue= 992, escrowSize= 0, escrowValue= 0]
23:27:38:318: BVM: CHARGING --> REJECTING

23:27:38:325: BVM: Rejected a coin of value 2
23:27:38:325: BVM: [current= 2, price= 80, credit= 0, badCIsNull= true, vaultSize= 32, vaultValue= 992, escrowSize= 0, escrowValue= 0]
23:27:38:326: BVM: REJECTING -> CHARGING

23:27:38:333: USER: Received coin(s) of total value 2
23:27:38:333: BVM: ### Chosen beverage: Coffee,  Insert coins: 80 cents ###
23:27:38:333: USER: [choice= 2, bill= 80, advance= 0, cupIsNull= true, walletSize= 15, walletValue= 534, purseSize= 3, purseValue= 9]
23:27:38:334: USER: INSERTING --> INSERTING

23:27:43:304: BVM: ### Chosen beverage: Coffee,  Insert coins: 80 cents ###
23:27:48:287: BVM: ### Chosen beverage: Coffee,  Insert coins: 80 cents ###
23:27:53:336: USER: Sent a coin of value 1
23:27:53:338: USER: [choice= 2, bill= 80, advance= 1, cupIsNull= true, walletSize= 14, walletValue= 533, purseSize= 3, purseValue= 9]
23:27:53:338: USER: INSERTING -> INSERTING

23:27:53:344: BVM: Received a coin of value: 1 cents
23:27:53:344: BVM: [current= 2, price= 80, credit= 0, badCIsNull= false, vaultSize= 32, vaultValue= 992, escrowSize= 0, escrowValue= 0]
23:27:53:344: BVM: CHARGING --> REJECTING

23:27:53:350: BVM: Rejected a coin of value 1
23:27:53:350: BVM: [current= 2, price= 80, credit= 0, badCIsNull= true, vaultSize= 32, vaultValue= 992, escrowSize= 0, escrowValue= 0]
23:27:53:350: BVM: REJECTING -> CHARGING

23:27:53:356: BVM: ### Chosen beverage: Coffee,  Insert coins: 80 cents ###
23:27:53:356: USER: Received coin(s) of total value 1
23:27:53:356: USER: [choice= 2, bill= 80, advance= 0, cupIsNull= true, walletSize= 14, walletValue= 533, purseSize= 4, purseValue= 10]
23:27:53:356: USER: INSERTING --> INSERTING
```



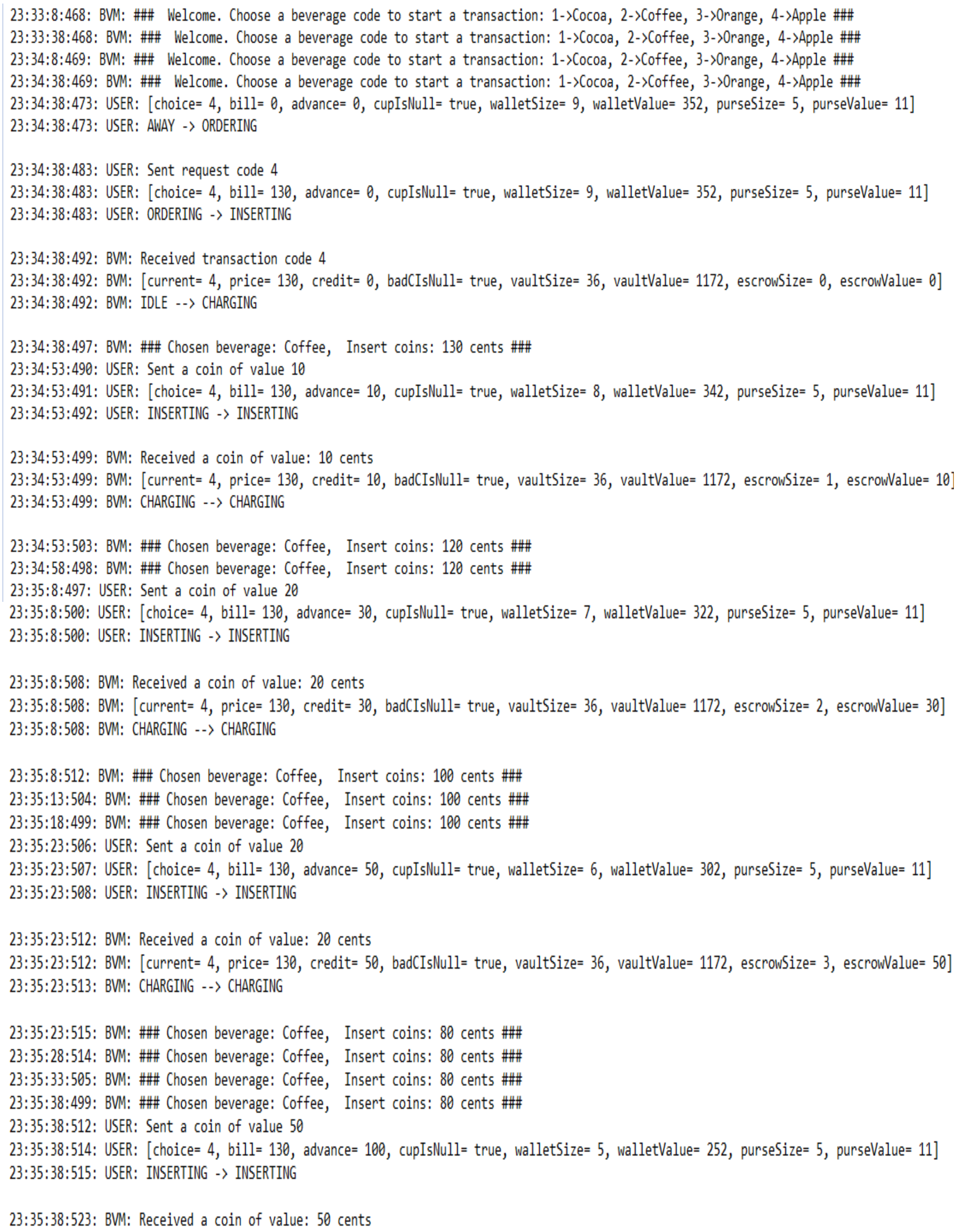
```
23:27:58:333: BVM: ### Chosen beverage: Coffee,  Insert coins: 80 cents ###
23:28:3:304: BVM: ### Chosen beverage: Coffee,  Insert coins: 80 cents ###
23:28:8:288: BVM: ### Chosen beverage: Coffee,  Insert coins: 80 cents ###
23:28:8:360: USER: Sent a coin of value 1
23:28:8:361: USER: [choice= 2, bill= 80, advance= 1, cupIsNull= true, walletSize= 13, walletValue= 532, purseSize= 4, purseValue= 10]
23:28:8:362: USER: INSERTING -> INSERTING

23:28:8:366: BVM: Received a coin of value: 1 cents
23:28:8:366: BVM: [current= 2, price= 80, credit= 0, badCIsNull= false, vaultSize= 32, vaultValue= 992, escrowSize= 0, escrowValue= 0]
23:28:8:366: BVM: CHARGING --> REJECTING

23:28:8:371: BVM: Rejected a coin of value 1
23:28:8:371: BVM: [current= 2, price= 80, credit= 0, badCIsNull= true, vaultSize= 32, vaultValue= 992, escrowSize= 0, escrowValue= 0]
23:28:8:371: BVM: REJECTING -> CHARGING

23:28:8:374: BVM: ### Chosen beverage: Coffee,  Insert coins: 80 cents ###
23:28:8:375: USER: Received coin(s) of total value 1
23:28:8:376: USER: [choice= 2, bill= 80, advance= 0, cupIsNull= true, walletSize= 13, walletValue= 532, purseSize= 5, purseValue= 11]
23:28:8:376: USER: INSERTING --> INSERTING

23:28:13:357: BVM: ### Chosen beverage: Coffee,  Insert coins: 80 cents ###
23:28:18:334: BVM: ### Chosen beverage: Coffee,  Insert coins: 80 cents ###
23:28:23:305: BVM: ### Chosen beverage: Coffee,  Insert coins: 80 cents ###
23:28:23:380: USER: Sent a coin of value 50
23:28:23:381: USER: [choice= 2, bill= 80, advance= 50, cupIsNull= true, walletSize= 12, walletValue= 482, purseSize= 5, purseValue= 11]
23:28:23:382: USER: INSERTING -> INSERTING
23:28:23:388: BVM: Received a coin of value: 50 cents
23:28:23:388: BVM: [current= 2, price= 80, credit= 50, badCIsNull= true, vaultSize= 32, vaultValue= 992, escrowSize= 1, escrowValue= 50]
23:28:23:389: BVM: CHARGING --> CHARGING

23:28:23:392: BVM: ### Chosen beverage: Coffee,  Insert coins: 30 cents ###
23:28:28:288: BVM: ### Chosen beverage: Coffee,  Insert coins: 30 cents ###
23:28:28:374: BVM: ### Chosen beverage: Coffee,  Insert coins: 30 cents ###
23:28:33:358: BVM: ### Chosen beverage: Coffee,  Insert coins: 30 cents ###
23:28:38:335: BVM: ### Chosen beverage: Coffee,  Insert coins: 30 cents ###
23:28:38:388: USER: Sent a coin of value 10
23:28:38:391: USER: [choice= 2, bill= 80, advance= 60, cupIsNull= true, walletSize= 11, walletValue= 472, purseSize= 5, purseValue= 11]
23:28:38:391: USER: INSERTING -> INSERTING

23:28:38:400: BVM: Received a coin of value: 10 cents
23:28:38:400: BVM: [current= 2, price= 80, credit= 60, badCIsNull= true, vaultSize= 32, vaultValue= 992, escrowSize= 2, escrowValue= 60]
23:28:38:400: BVM: CHARGING --> CHARGING

23:28:38:405: BVM: ### Chosen beverage: Coffee,  Insert coins: 20 cents ###
23:28:43:305: BVM: ### Chosen beverage: Coffee,  Insert coins: 20 cents ###
23:28:43:392: BVM: ### Chosen beverage: Coffee,  Insert coins: 20 cents ###
23:28:48:288: BVM: ### Chosen beverage: Coffee,  Insert coins: 20 cents ###
23:28:48:374: BVM: ### Chosen beverage: Coffee,  Insert coins: 20 cents ###
23:28:53:358: BVM: ### Chosen beverage: Coffee,  Insert coins: 20 cents ###
23:28:53:397: USER: Sent a coin of value 20
23:28:53:399: USER: [choice= 2, bill= 80, advance= 80, cupIsNull= true, walletSize= 10, walletValue= 452, purseSize= 5, purseValue= 11]
23:28:53:400: USER: INSERTING -> WAITING
```



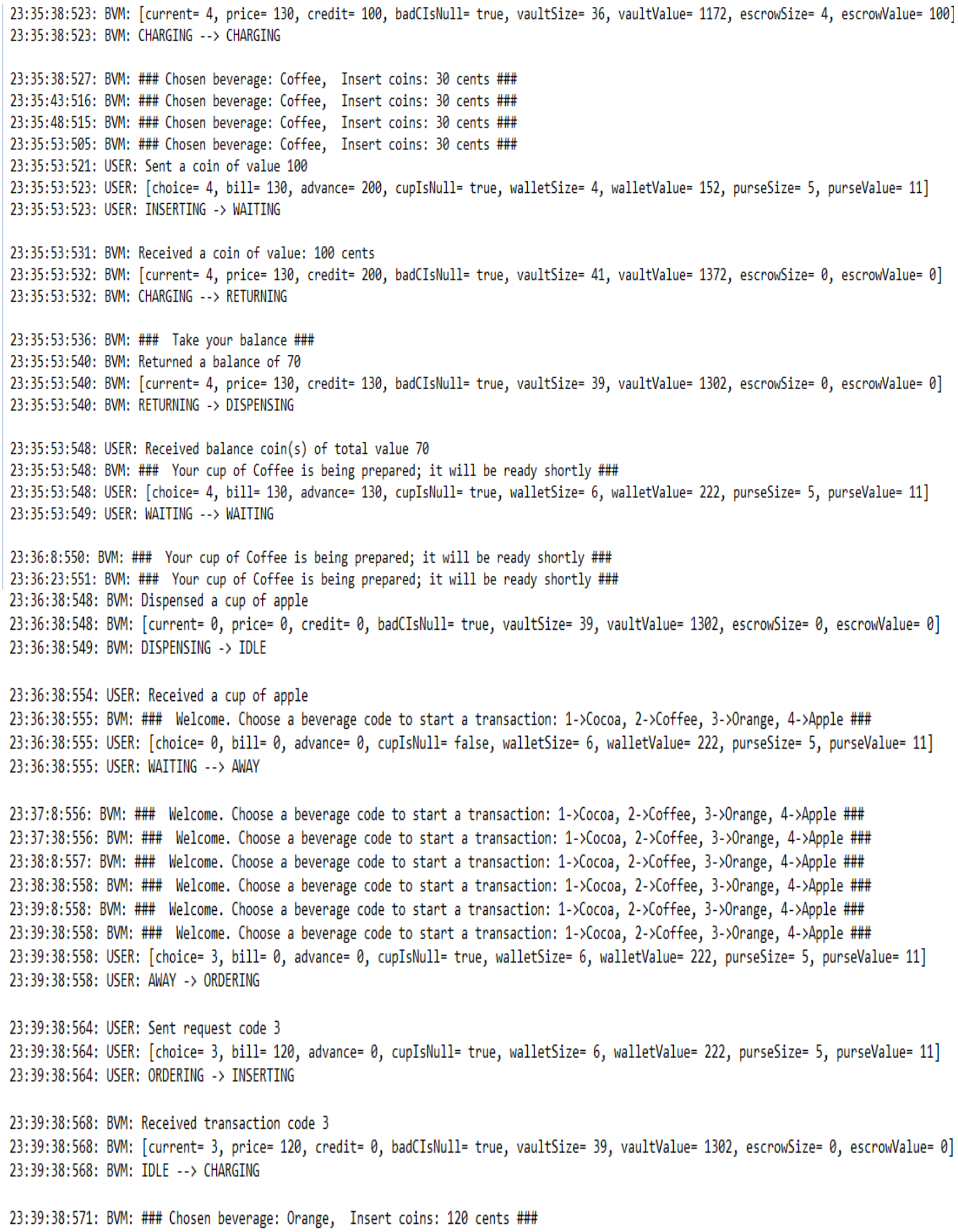
```
23:28:53:408: BVM: Received a coin of value: 20 cents
23:28:53:408: BVM: [current= 2, price= 80, credit= 80, badCIsNull= true, vaultSize= 35, vaultValue= 1072, escrowSize= 0, escrowValue= 0]
23:28:53:408: BVM: CHARGING --> DISPENSING

23:28:53:412: BVM: ###  Your cup of Coffee is being prepared; it will be ready shortly ###
23:29:8:413: BVM: ###  Your cup of Coffee is being prepared; it will be ready shortly ###
23:29:23:414: BVM: ###  Your cup of Coffee is being prepared; it will be ready shortly ###
23:29:38:415: BVM: ###  Your cup of Coffee is being prepared; it will be ready shortly ###
23:29:38:415: BVM: Dispensed a cup of coffee
23:29:38:415: BVM: [current= 0, price= 0, credit= 0, badCIsNull= true, vaultSize= 35, vaultValue= 1072, escrowSize= 0, escrowValue= 0]
23:29:38:415: BVM: DISPENSING -> IDLE

23:29:38:423: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:29:38:424: USER: Received a cup of coffee
23:29:38:424: USER: [choice= 0, bill= 0, advance= 0, cupIsNull= false, walletSize= 10, walletValue= 452, purseSize= 5, purseValue= 11]
23:29:38:424: USER: WAITING --> AWAY

23:30:8:424: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:30:38:424: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:31:8:425: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:31:38:425: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:31:38:428: USER: [choice= 1, bill= 0, advance= 0, cupIsNull= true, walletSize= 10, walletValue= 452, purseSize= 5, purseValue= 11]
23:31:38:428: USER: AWAY -> ORDERING

23:31:38:436: USER: Sent request code 1
23:31:38:436: USER: [choice= 1, bill= 100, advance= 0, cupIsNull= true, walletSize= 10, walletValue= 452, purseSize= 5, purseValue= 11]
23:31:38:437: USER: ORDERING -> INSERTING
```



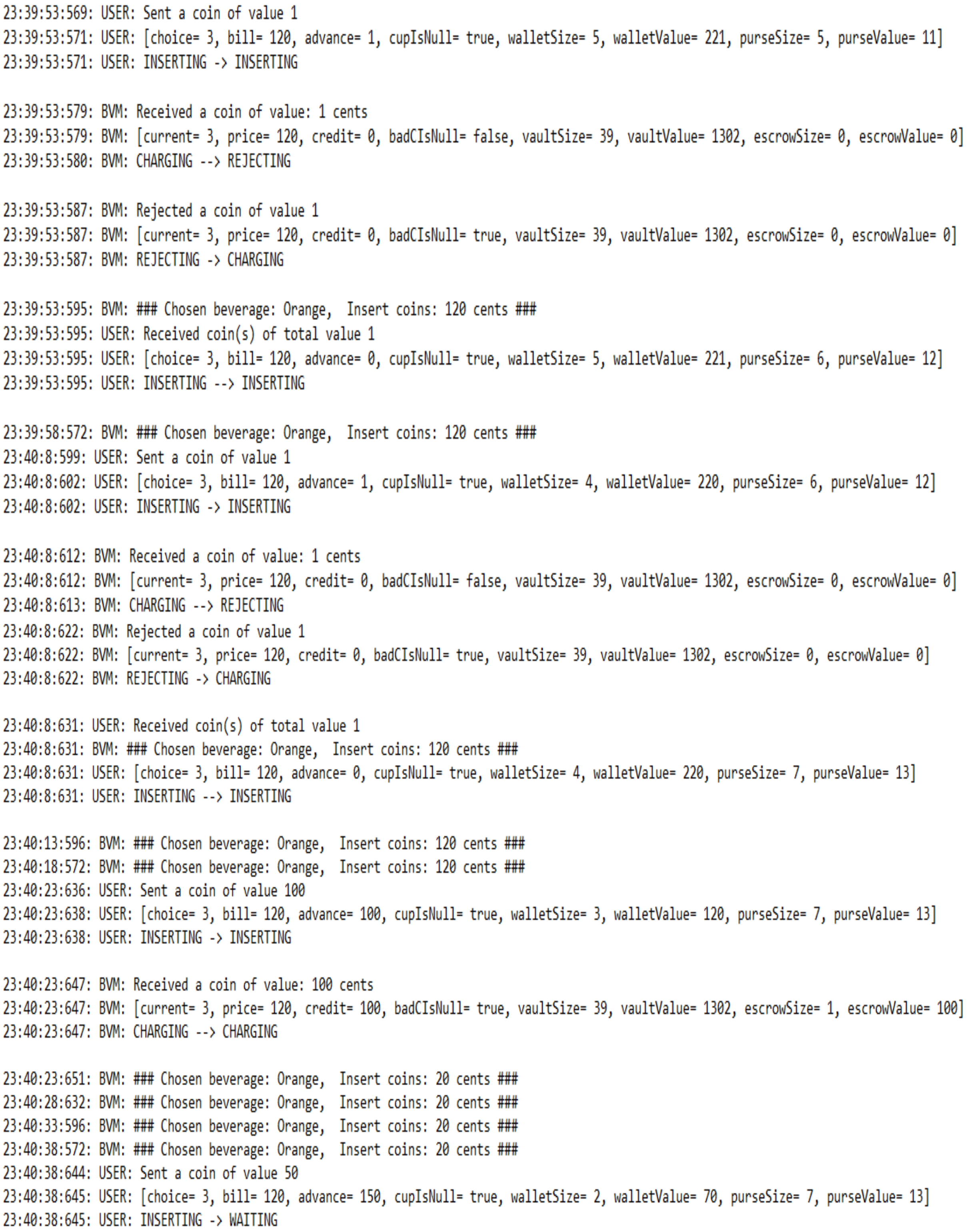
```
23:31:38:443: BVM: Received transaction code 1
23:31:38:444: BVM: [current= 1, price= 100, credit= 0, badCIsNull= true, vaultSize= 35, vaultValue= 1072, escrowSize= 0, escrowValue= 0]
23:31:38:444: BVM: IDLE --> CHARGING

23:31:38:448: BVM: ### Chosen beverage: Cocoa,  Insert coins: 100 cents ###
23:31:53:443: USER: Sent a coin of value 100
23:31:53:445: USER: [choice= 1, bill= 100, advance= 100, cupIsNull= true, walletSize= 9, walletValue= 352, purseSize= 5, purseValue= 11]
23:31:53:445: USER: INSERTING -> WAITING

23:31:53:453: BVM: Received a coin of value: 100 cents
23:31:53:454: BVM: [current= 1, price= 100, credit= 100, badCIsNull= true, vaultSize= 36, vaultValue= 1172, escrowSize= 0, escrowValue= 0]
23:31:53:454: BVM: CHARGING --> DISPENSING

23:31:53:458: BVM: ###  Your cup of Cocoa is being prepared; it will be ready shortly ###
23:32:8:458: BVM: ###  Your cup of Cocoa is being prepared; it will be ready shortly ###
23:32:23:459: BVM: ###  Your cup of Cocoa is being prepared; it will be ready shortly ###
23:32:38:459: BVM: ###  Your cup of Cocoa is being prepared; it will be ready shortly ###
23:32:38:461: BVM: Dispensed a cup of cocoa
23:32:38:461: BVM: [current= 0, price= 0, credit= 0, badCIsNull= true, vaultSize= 36, vaultValue= 1172, escrowSize= 0, escrowValue= 0]
23:32:38:461: BVM: DISPENSING -> IDLE

23:32:38:468: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:32:38:468: USER: Received a cup of cocoa
23:32:38:468: USER: [choice= 0, bill= 0, advance= 0, cupIsNull= false, walletSize= 9, walletValue= 352, purseSize= 5, purseValue= 11]
23:32:38:468: USER: WAITING --> AWAY
```



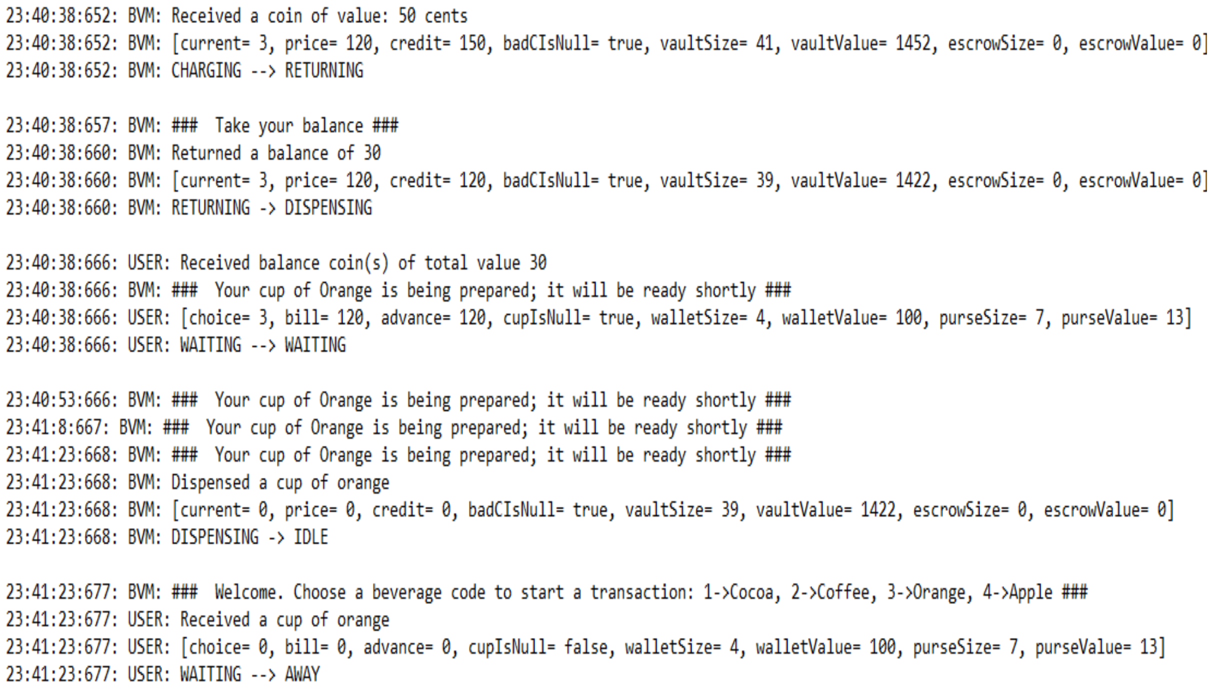
```
23:33:8:468: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:33:38:468: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:34:8:469: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:34:38:469: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:34:38:473: USER: [choice= 4, bill= 0, advance= 0, cupIsNull= true, walletSize= 9, walletValue= 352, purseSize= 5, purseValue= 11]
23:34:38:473: USER: AWAY -> ORDERING

23:34:38:483: USER: Sent request code 4
23:34:38:483: USER: [choice= 4, bill= 130, advance= 0, cupIsNull= true, walletSize= 9, walletValue= 352, purseSize= 5, purseValue= 11]
23:34:38:483: USER: ORDERING -> INSERTING

23:34:38:492: BVM: Received transaction code 4
23:34:38:492: BVM: [current= 4, price= 130, credit= 0, badCIsNull= true, vaultSize= 36, vaultValue= 1172, escrowSize= 0, escrowValue= 0]
23:34:38:492: BVM: IDLE --> CHARGING

23:34:38:497: BVM: ### Chosen beverage: Coffee,  Insert coins: 130 cents ###
23:34:53:490: USER: Sent a coin of value 10
23:34:53:491: USER: [choice= 4, bill= 130, advance= 10, cupIsNull= true, walletSize= 8, walletValue= 342, purseSize= 5, purseValue= 11]
23:34:53:492: USER: INSERTING -> INSERTING

23:34:53:499: BVM: Received a coin of value: 10 cents
23:34:53:499: BVM: [current= 4, price= 130, credit= 10, badCIsNull= true, vaultSize= 36, vaultValue= 1172, escrowSize= 1, escrowValue= 10]
23:34:53:499: BVM: CHARGING --> CHARGING

23:34:53:503: BVM: ### Chosen beverage: Coffee,  Insert coins: 120 cents ###
23:34:58:498: BVM: ### Chosen beverage: Coffee,  Insert coins: 120 cents ###
23:35:8:497: USER: Sent a coin of value 20
23:35:8:500: USER: [choice= 4, bill= 130, advance= 30, cupIsNull= true, walletSize= 7, walletValue= 322, purseSize= 5, purseValue= 11]
23:35:8:500: USER: INSERTING -> INSERTING

23:35:8:508: BVM: Received a coin of value: 20 cents
23:35:8:508: BVM: [current= 4, price= 130, credit= 30, badCIsNull= true, vaultSize= 36, vaultValue= 1172, escrowSize= 2, escrowValue= 30]
23:35:8:508: BVM: CHARGING --> CHARGING

23:35:8:512: BVM: ### Chosen beverage: Coffee,  Insert coins: 100 cents ###
23:35:13:504: BVM: ### Chosen beverage: Coffee,  Insert coins: 100 cents ###
23:35:18:499: BVM: ### Chosen beverage: Coffee,  Insert coins: 100 cents ###
23:35:23:506: USER: Sent a coin of value 20
23:35:23:507: USER: [choice= 4, bill= 130, advance= 50, cupIsNull= true, walletSize= 6, walletValue= 302, purseSize= 5, purseValue= 11]
23:35:23:508: USER: INSERTING -> INSERTING

23:35:23:512: BVM: Received a coin of value: 20 cents
23:35:23:512: BVM: [current= 4, price= 130, credit= 50, badCIsNull= true, vaultSize= 36, vaultValue= 1172, escrowSize= 3, escrowValue= 50]
23:35:23:513: BVM: CHARGING --> CHARGING

23:35:23:515: BVM: ### Chosen beverage: Coffee,  Insert coins: 80 cents ###
23:35:28:514: BVM: ### Chosen beverage: Coffee,  Insert coins: 80 cents ###
23:35:33:505: BVM: ### Chosen beverage: Coffee,  Insert coins: 80 cents ###
23:35:38:499: BVM: ### Chosen beverage: Coffee,  Insert coins: 80 cents ###
23:35:38:512: USER: Sent a coin of value 50
23:35:38:514: USER: [choice= 4, bill= 130, advance= 100, cupIsNull= true, walletSize= 5, walletValue= 252, purseSize= 5, purseValue= 11]
23:35:38:515: USER: INSERTING -> INSERTING

23:35:38:523: BVM: Received a coin of value: 50 cents
```



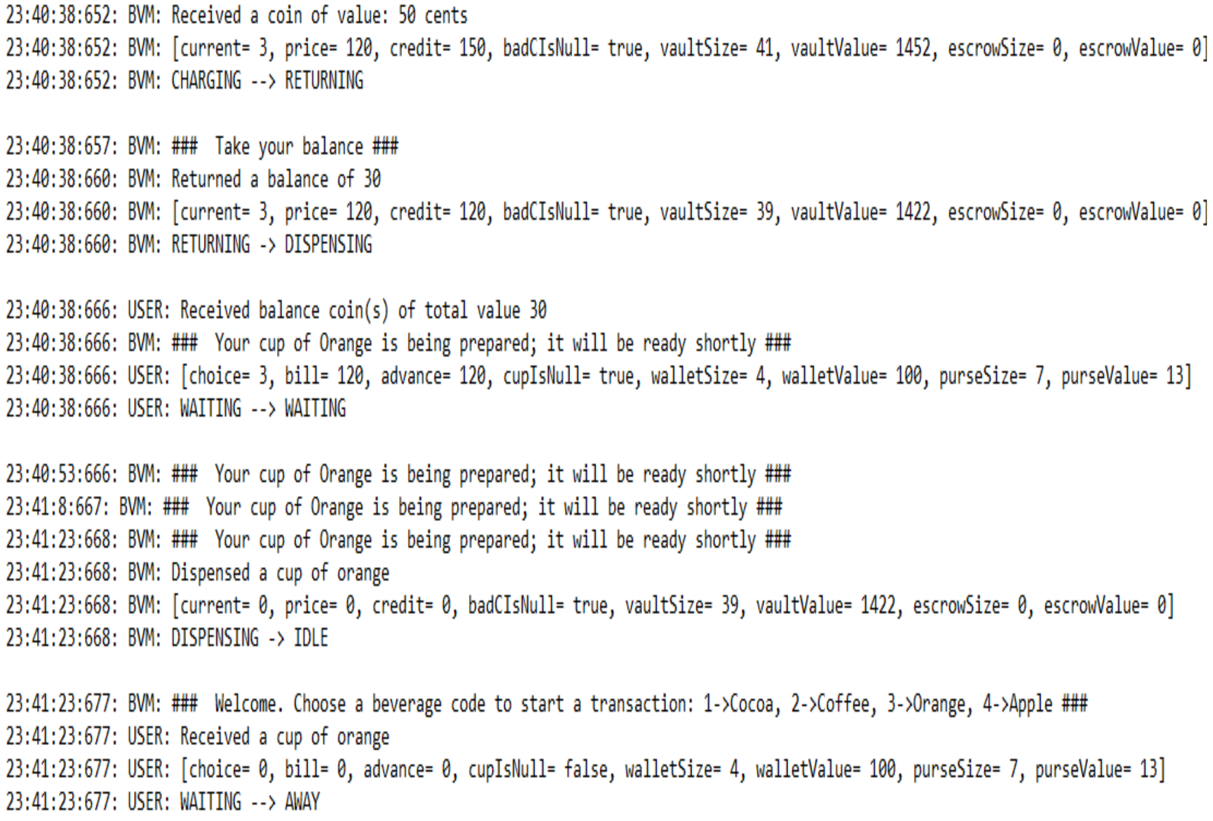
```
23:35:38:523: BVM: [current= 4, price= 130, credit= 100, badCIsNull= true, vaultSize= 36, vaultValue= 1172, escrowSize= 4, escrowValue= 100]
23:35:38:523: BVM: CHARGING --> CHARGING

23:35:38:527: BVM: ### Chosen beverage: Coffee,  Insert coins: 30 cents ###
23:35:43:516: BVM: ### Chosen beverage: Coffee,  Insert coins: 30 cents ###
23:35:48:515: BVM: ### Chosen beverage: Coffee,  Insert coins: 30 cents ###
23:35:53:505: BVM: ### Chosen beverage: Coffee,  Insert coins: 30 cents ###
23:35:53:521: USER: Sent a coin of value 100
23:35:53:523: USER: [choice= 4, bill= 130, advance= 200, cupIsNull= true, walletSize= 4, walletValue= 152, purseSize= 5, purseValue= 11]
23:35:53:523: USER: INSERTING -> WAITING

23:35:53:531: BVM: Received a coin of value: 100 cents
23:35:53:532: BVM: [current= 4, price= 130, credit= 200, badCIsNull= true, vaultSize= 41, vaultValue= 1372, escrowSize= 0, escrowValue= 0]
23:35:53:532: BVM: CHARGING --> RETURNING

23:35:53:536: BVM: ###  Take your balance ###
23:35:53:540: BVM: Returned a balance of 70
23:35:53:540: BVM: [current= 4, price= 130, credit= 130, badCIsNull= true, vaultSize= 39, vaultValue= 1302, escrowSize= 0, escrowValue= 0]
23:35:53:540: BVM: RETURNING -> DISPENSING

23:35:53:548: USER: Received balance coin(s) of total value 70
23:35:53:548: BVM: ###  Your cup of Coffee is being prepared; it will be ready shortly ###
23:35:53:548: USER: [choice= 4, bill= 130, advance= 130, cupIsNull= true, walletSize= 6, walletValue= 222, purseSize= 5, purseValue= 11]
23:35:53:549: USER: WAITING --> WAITING

23:36:8:550: BVM: ###  Your cup of Coffee is being prepared; it will be ready shortly ###
23:36:23:551: BVM: ###  Your cup of Coffee is being prepared; it will be ready shortly ###
23:36:38:548: BVM: Dispensed a cup of apple
23:36:38:548: BVM: [current= 0, price= 0, credit= 0, badCIsNull= true, vaultSize= 39, vaultValue= 1302, escrowSize= 0, escrowValue= 0]
23:36:38:549: BVM: DISPENSING -> IDLE

23:36:38:554: USER: Received a cup of apple
23:36:38:555: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:36:38:555: USER: [choice= 0, bill= 0, advance= 0, cupIsNull= false, walletSize= 6, walletValue= 222, purseSize= 5, purseValue= 11]
23:36:38:555: USER: WAITING --> AWAY

23:37:8:556: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:37:38:556: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:38:8:557: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:38:38:558: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:39:8:558: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:39:38:558: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:39:38:558: USER: [choice= 3, bill= 0, advance= 0, cupIsNull= true, walletSize= 6, walletValue= 222, purseSize= 5, purseValue= 11]
23:39:38:558: USER: AWAY -> ORDERING

23:39:38:564: USER: Sent request code 3
23:39:38:564: USER: [choice= 3, bill= 120, advance= 0, cupIsNull= true, walletSize= 6, walletValue= 222, purseSize= 5, purseValue= 11]
23:39:38:564: USER: ORDERING -> INSERTING

23:39:38:568: BVM: Received transaction code 3
23:39:38:568: BVM: [current= 3, price= 120, credit= 0, badCIsNull= true, vaultSize= 39, vaultValue= 1302, escrowSize= 0, escrowValue= 0]
23:39:38:568: BVM: IDLE --> CHARGING

23:39:38:571: BVM: ### Chosen beverage: Orange,  Insert coins: 120 cents ###
```



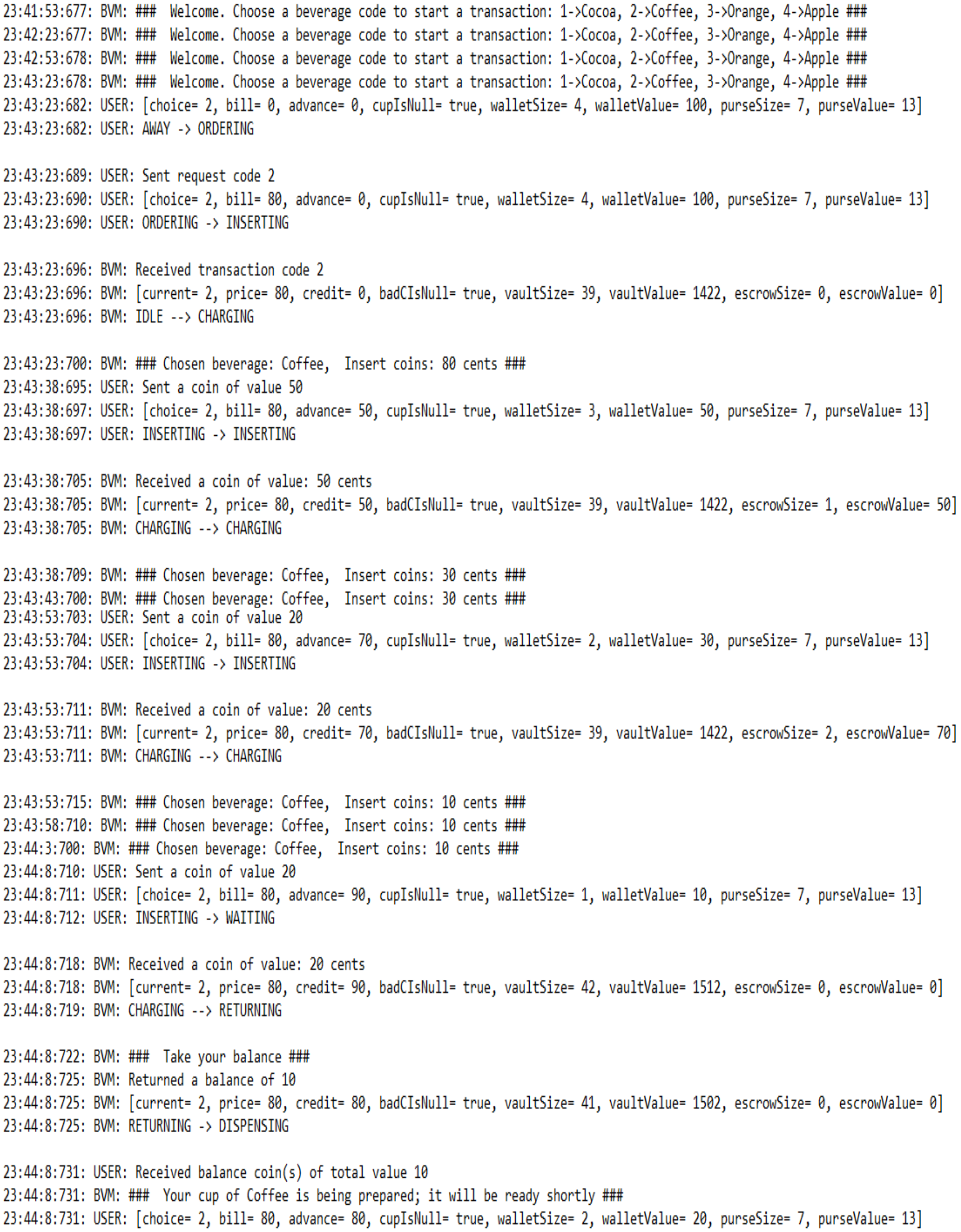
```
23:39:53:569: USER: Sent a coin of value 1
23:39:53:571: USER: [choice= 3, bill= 120, advance= 1, cupIsNull= true, walletSize= 5, walletValue= 221, purseSize= 5, purseValue= 11]
23:39:53:571: USER: INSERTING -> INSERTING

23:39:53:579: BVM: Received a coin of value: 1 cents
23:39:53:579: BVM: [current= 3, price= 120, credit= 0, badCIsNull= false, vaultSize= 39, vaultValue= 1302, escrowSize= 0, escrowValue= 0]
23:39:53:580: BVM: CHARGING --> REJECTING

23:39:53:587: BVM: Rejected a coin of value 1
23:39:53:587: BVM: [current= 3, price= 120, credit= 0, badCIsNull= true, vaultSize= 39, vaultValue= 1302, escrowSize= 0, escrowValue= 0]
23:39:53:587: BVM: REJECTING -> CHARGING

23:39:53:595: BVM: ### Chosen beverage: Orange,  Insert coins: 120 cents ###
23:39:53:595: USER: Received coin(s) of total value 1
23:39:53:595: USER: [choice= 3, bill= 120, advance= 0, cupIsNull= true, walletSize= 5, walletValue= 221, purseSize= 6, purseValue= 12]
23:39:53:595: USER: INSERTING --> INSERTING

23:39:58:572: BVM: ### Chosen beverage: Orange,  Insert coins: 120 cents ###
23:40:8:599: USER: Sent a coin of value 1
23:40:8:602: USER: [choice= 3, bill= 120, advance= 1, cupIsNull= true, walletSize= 4, walletValue= 220, purseSize= 6, purseValue= 12]
23:40:8:602: USER: INSERTING -> INSERTING

23:40:8:612: BVM: Received a coin of value: 1 cents
23:40:8:612: BVM: [current= 3, price= 120, credit= 0, badCIsNull= false, vaultSize= 39, vaultValue= 1302, escrowSize= 0, escrowValue= 0]
23:40:8:613: BVM: CHARGING --> REJECTING
23:40:8:622: BVM: Rejected a coin of value 1
23:40:8:622: BVM: [current= 3, price= 120, credit= 0, badCIsNull= true, vaultSize= 39, vaultValue= 1302, escrowSize= 0, escrowValue= 0]
23:40:8:622: BVM: REJECTING -> CHARGING

23:40:8:631: USER: Received coin(s) of total value 1
23:40:8:631: BVM: ### Chosen beverage: Orange,  Insert coins: 120 cents ###
23:40:8:631: USER: [choice= 3, bill= 120, advance= 0, cupIsNull= true, walletSize= 4, walletValue= 220, purseSize= 7, purseValue= 13]
23:40:8:631: USER: INSERTING --> INSERTING

23:40:13:596: BVM: ### Chosen beverage: Orange,  Insert coins: 120 cents ###
23:40:18:572: BVM: ### Chosen beverage: Orange,  Insert coins: 120 cents ###
23:40:23:636: USER: Sent a coin of value 100
23:40:23:638: USER: [choice= 3, bill= 120, advance= 100, cupIsNull= true, walletSize= 3, walletValue= 120, purseSize= 7, purseValue= 13]
23:40:23:638: USER: INSERTING -> INSERTING

23:40:23:647: BVM: Received a coin of value: 100 cents
23:40:23:647: BVM: [current= 3, price= 120, credit= 100, badCIsNull= true, vaultSize= 39, vaultValue= 1302, escrowSize= 1, escrowValue= 100]
23:40:23:647: BVM: CHARGING --> CHARGING

23:40:23:651: BVM: ### Chosen beverage: Orange,  Insert coins: 20 cents ###
23:40:28:632: BVM: ### Chosen beverage: Orange,  Insert coins: 20 cents ###
23:40:33:596: BVM: ### Chosen beverage: Orange,  Insert coins: 20 cents ###
23:40:38:572: BVM: ### Chosen beverage: Orange,  Insert coins: 20 cents ###
23:40:38:644: USER: Sent a coin of value 50
23:40:38:645: USER: [choice= 3, bill= 120, advance= 150, cupIsNull= true, walletSize= 2, walletValue= 70, purseSize= 7, purseValue= 13]
23:40:38:645: USER: INSERTING -> WAITING
```



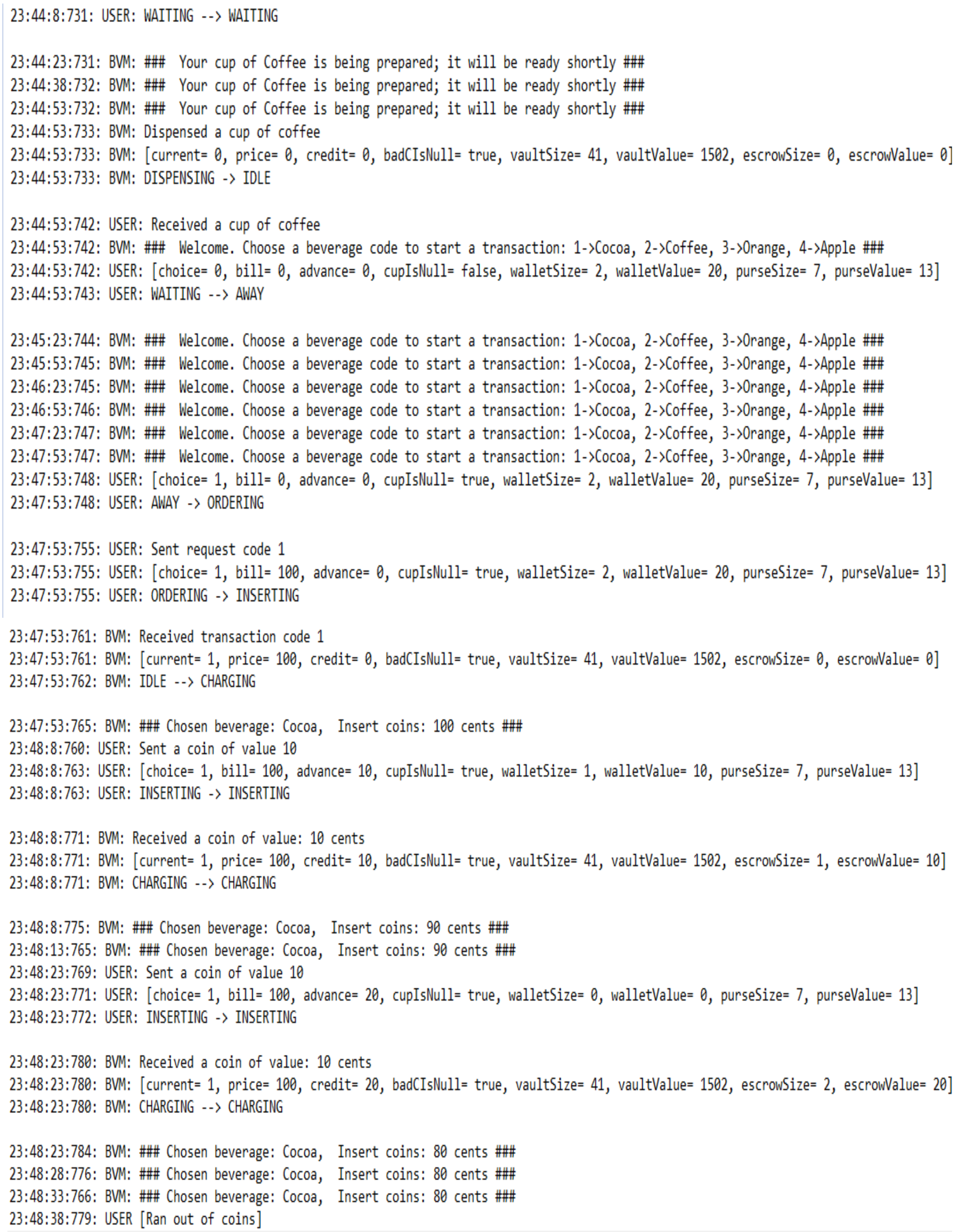
```
23:40:38:652: BVM: Received a coin of value: 50 cents
23:40:38:652: BVM: [current= 3, price= 120, credit= 150, badCIsNull= true, vaultSize= 41, vaultValue= 1452, escrowSize= 0, escrowValue= 0]
23:40:38:652: BVM: CHARGING --> RETURNING

23:40:38:657: BVM: ###  Take your balance ###
23:40:38:660: BVM: Returned a balance of 30
23:40:38:660: BVM: [current= 3, price= 120, credit= 120, badCIsNull= true, vaultSize= 39, vaultValue= 1422, escrowSize= 0, escrowValue= 0]
23:40:38:660: BVM: RETURNING -> DISPENSING

23:40:38:666: USER: Received balance coin(s) of total value 30
23:40:38:666: BVM: ###  Your cup of Orange is being prepared; it will be ready shortly ###
23:40:38:666: USER: [choice= 3, bill= 120, advance= 120, cupIsNull= true, walletSize= 4, walletValue= 100, purseSize= 7, purseValue= 13]
23:40:38:666: USER: WAITING --> WAITING

23:40:53:666: BVM: ###  Your cup of Orange is being prepared; it will be ready shortly ###
23:41:8:667: BVM: ###  Your cup of Orange is being prepared; it will be ready shortly ###
23:41:23:668: BVM: ###  Your cup of Orange is being prepared; it will be ready shortly ###
23:41:23:668: BVM: Dispensed a cup of orange
23:41:23:668: BVM: [current= 0, price= 0, credit= 0, badCIsNull= true, vaultSize= 39, vaultValue= 1422, escrowSize= 0, escrowValue= 0]
23:41:23:668: BVM: DISPENSING -> IDLE

23:41:23:677: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:41:23:677: USER: Received a cup of orange
23:41:23:677: USER: [choice= 0, bill= 0, advance= 0, cupIsNull= false, walletSize= 4, walletValue= 100, purseSize= 7, purseValue= 13]
23:41:23:677: USER: WAITING --> AWAY
```



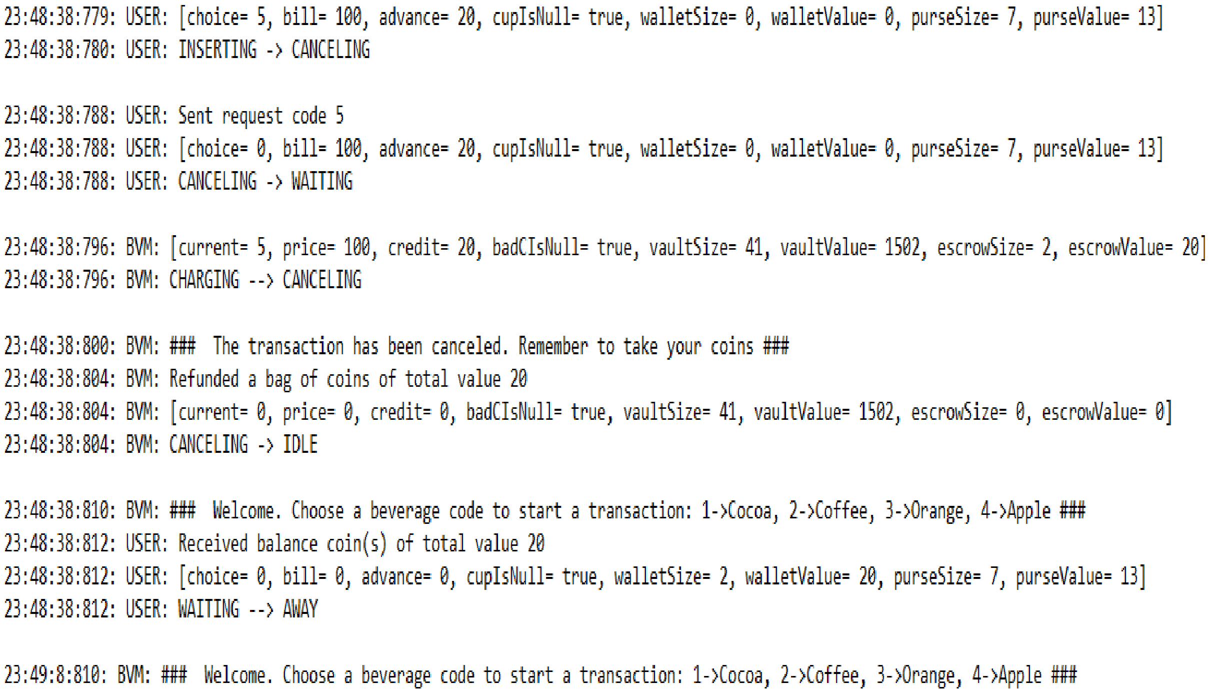
```
23:40:38:652: BVM: Received a coin of value: 50 cents
23:40:38:652: BVM: [current= 3, price= 120, credit= 150, badCIsNull= true, vaultSize= 41, vaultValue= 1452, escrowSize= 0, escrowValue= 0]
23:40:38:652: BVM: CHARGING --> RETURNING

23:40:38:657: BVM: ###  Take your balance ###
23:40:38:660: BVM: Returned a balance of 30
23:40:38:660: BVM: [current= 3, price= 120, credit= 120, badCIsNull= true, vaultSize= 39, vaultValue= 1422, escrowSize= 0, escrowValue= 0]
23:40:38:660: BVM: RETURNING -> DISPENSING

23:40:38:666: USER: Received balance coin(s) of total value 30
23:40:38:666: BVM: ###  Your cup of Orange is being prepared; it will be ready shortly ###
23:40:38:666: USER: [choice= 3, bill= 120, advance= 120, cupIsNull= true, walletSize= 4, walletValue= 100, purseSize= 7, purseValue= 13]
23:40:38:666: USER: WAITING --> WAITING

23:40:53:666: BVM: ###  Your cup of Orange is being prepared; it will be ready shortly ###
23:41:8:667: BVM: ###  Your cup of Orange is being prepared; it will be ready shortly ###
23:41:23:668: BVM: ###  Your cup of Orange is being prepared; it will be ready shortly ###
23:41:23:668: BVM: Dispensed a cup of orange
23:41:23:668: BVM: [current= 0, price= 0, credit= 0, badCIsNull= true, vaultSize= 39, vaultValue= 1422, escrowSize= 0, escrowValue= 0]
23:41:23:668: BVM: DISPENSING -> IDLE

23:41:23:677: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:41:23:677: USER: Received a cup of orange
23:41:23:677: USER: [choice= 0, bill= 0, advance= 0, cupIsNull= false, walletSize= 4, walletValue= 100, purseSize= 7, purseValue= 13]
23:41:23:677: USER: WAITING --> AWAY
```



```
23:41:53:677: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:42:23:677: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:42:53:678: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:43:23:678: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:43:23:682: USER: [choice= 2, bill= 0, advance= 0, cupIsNull= true, walletSize= 4, walletValue= 100, purseSize= 7, purseValue= 13]
23:43:23:682: USER: AWAY -> ORDERING

23:43:23:689: USER: Sent request code 2
23:43:23:690: USER: [choice= 2, bill= 80, advance= 0, cupIsNull= true, walletSize= 4, walletValue= 100, purseSize= 7, purseValue= 13]
23:43:23:690: USER: ORDERING -> INSERTING

23:43:23:696: BVM: Received transaction code 2
23:43:23:696: BVM: [current= 2, price= 80, credit= 0, badCIsNull= true, vaultSize= 39, vaultValue= 1422, escrowSize= 0, escrowValue= 0]
23:43:23:696: BVM: IDLE --> CHARGING

23:43:23:700: BVM: ### Chosen beverage: Coffee,  Insert coins: 80 cents ###
23:43:38:695: USER: Sent a coin of value 50
23:43:38:697: USER: [choice= 2, bill= 80, advance= 50, cupIsNull= true, walletSize= 3, walletValue= 50, purseSize= 7, purseValue= 13]
23:43:38:697: USER: INSERTING -> INSERTING

23:43:38:705: BVM: Received a coin of value: 50 cents
23:43:38:705: BVM: [current= 2, price= 80, credit= 50, badCIsNull= true, vaultSize= 39, vaultValue= 1422, escrowSize= 1, escrowValue= 50]
23:43:38:705: BVM: CHARGING --> CHARGING

23:43:38:709: BVM: ### Chosen beverage: Coffee,  Insert coins: 30 cents ###
23:43:43:700: BVM: ### Chosen beverage: Coffee,  Insert coins: 30 cents ###
23:43:53:703: USER: Sent a coin of value 20
23:43:53:704: USER: [choice= 2, bill= 80, advance= 70, cupIsNull= true, walletSize= 2, walletValue= 30, purseSize= 7, purseValue= 13]
23:43:53:704: USER: INSERTING -> INSERTING

23:43:53:711: BVM: Received a coin of value: 20 cents
23:43:53:711: BVM: [current= 2, price= 80, credit= 70, badCIsNull= true, vaultSize= 39, vaultValue= 1422, escrowSize= 2, escrowValue= 70]
23:43:53:711: BVM: CHARGING --> CHARGING

23:43:53:715: BVM: ### Chosen beverage: Coffee,  Insert coins: 10 cents ###
23:43:58:710: BVM: ### Chosen beverage: Coffee,  Insert coins: 10 cents ###
23:44:3:700: BVM: ### Chosen beverage: Coffee,  Insert coins: 10 cents ###
23:44:8:710: USER: Sent a coin of value 20
23:44:8:711: USER: [choice= 2, bill= 80, advance= 90, cupIsNull= true, walletSize= 1, walletValue= 10, purseSize= 7, purseValue= 13]
23:44:8:712: USER: INSERTING -> WAITING

23:44:8:718: BVM: Received a coin of value: 20 cents
23:44:8:718: BVM: [current= 2, price= 80, credit= 90, badCIsNull= true, vaultSize= 42, vaultValue= 1512, escrowSize= 0, escrowValue= 0]
23:44:8:719: BVM: CHARGING --> RETURNING

23:44:8:722: BVM: ###  Take your balance ###
23:44:8:725: BVM: Returned a balance of 10
23:44:8:725: BVM: [current= 2, price= 80, credit= 80, badCIsNull= true, vaultSize= 41, vaultValue= 1502, escrowSize= 0, escrowValue= 0]
23:44:8:725: BVM: RETURNING -> DISPENSING

23:44:8:731: USER: Received balance coin(s) of total value 10
23:44:8:731: BVM: ###  Your cup of Coffee is being prepared; it will be ready shortly ###
23:44:8:731: USER: [choice= 2, bill= 80, advance= 80, cupIsNull= true, walletSize= 2, walletValue= 20, purseSize= 7, purseValue= 13]
```



```
23:44:8:731: USER: WAITING --> WAITING

23:44:23:731: BVM: ###  Your cup of Coffee is being prepared; it will be ready shortly ###
23:44:38:732: BVM: ###  Your cup of Coffee is being prepared; it will be ready shortly ###
23:44:53:732: BVM: ###  Your cup of Coffee is being prepared; it will be ready shortly ###
23:44:53:733: BVM: Dispensed a cup of coffee
23:44:53:733: BVM: [current= 0, price= 0, credit= 0, badCIsNull= true, vaultSize= 41, vaultValue= 1502, escrowSize= 0, escrowValue= 0]
23:44:53:733: BVM: DISPENSING -> IDLE

23:44:53:742: USER: Received a cup of coffee
23:44:53:742: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:44:53:742: USER: [choice= 0, bill= 0, advance= 0, cupIsNull= false, walletSize= 2, walletValue= 20, purseSize= 7, purseValue= 13]
23:44:53:743: USER: WAITING --> AWAY

23:45:23:744: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:45:53:745: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:46:23:745: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:46:53:746: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:47:23:747: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:47:53:747: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:47:53:748: USER: [choice= 1, bill= 0, advance= 0, cupIsNull= true, walletSize= 2, walletValue= 20, purseSize= 7, purseValue= 13]
23:47:53:748: USER: AWAY -> ORDERING

23:47:53:755: USER: Sent request code 1
23:47:53:755: USER: [choice= 1, bill= 100, advance= 0, cupIsNull= true, walletSize= 2, walletValue= 20, purseSize= 7, purseValue= 13]
23:47:53:755: USER: ORDERING -> INSERTING

23:47:53:761: BVM: Received transaction code 1
23:47:53:761: BVM: [current= 1, price= 100, credit= 0, badCIsNull= true, vaultSize= 41, vaultValue= 1502, escrowSize= 0, escrowValue= 0]
23:47:53:762: BVM: IDLE --> CHARGING

23:47:53:765: BVM: ### Chosen beverage: Cocoa,  Insert coins: 100 cents ###
23:48:8:760: USER: Sent a coin of value 10
23:48:8:763: USER: [choice= 1, bill= 100, advance= 10, cupIsNull= true, walletSize= 1, walletValue= 10, purseSize= 7, purseValue= 13]
23:48:8:763: USER: INSERTING -> INSERTING

23:48:8:771: BVM: Received a coin of value: 10 cents
23:48:8:771: BVM: [current= 1, price= 100, credit= 10, badCIsNull= true, vaultSize= 41, vaultValue= 1502, escrowSize= 1, escrowValue= 10]
23:48:8:771: BVM: CHARGING --> CHARGING

23:48:8:775: BVM: ### Chosen beverage: Cocoa,  Insert coins: 90 cents ###
23:48:13:765: BVM: ### Chosen beverage: Cocoa,  Insert coins: 90 cents ###
23:48:23:769: USER: Sent a coin of value 10
23:48:23:771: USER: [choice= 1, bill= 100, advance= 20, cupIsNull= true, walletSize= 0, walletValue= 0, purseSize= 7, purseValue= 13]
23:48:23:772: USER: INSERTING -> INSERTING

23:48:23:780: BVM: Received a coin of value: 10 cents
23:48:23:780: BVM: [current= 1, price= 100, credit= 20, badCIsNull= true, vaultSize= 41, vaultValue= 1502, escrowSize= 2, escrowValue= 20]
23:48:23:780: BVM: CHARGING --> CHARGING

23:48:23:784: BVM: ### Chosen beverage: Cocoa,  Insert coins: 80 cents ###
23:48:28:776: BVM: ### Chosen beverage: Cocoa,  Insert coins: 80 cents ###
23:48:33:766: BVM: ### Chosen beverage: Cocoa,  Insert coins: 80 cents ###
23:48:38:779: USER [Ran out of coins]
```



```
23:48:38:779: USER: [choice= 5, bill= 100, advance= 20, cupIsNull= true, walletSize= 0, walletValue= 0, purseSize= 7, purseValue= 13]
23:48:38:780: USER: INSERTING -> CANCELING

23:48:38:788: USER: Sent request code 5
23:48:38:788: USER: [choice= 0, bill= 100, advance= 20, cupIsNull= true, walletSize= 0, walletValue= 0, purseSize= 7, purseValue= 13]
23:48:38:788: USER: CANCELING -> WAITING

23:48:38:796: BVM: [current= 5, price= 100, credit= 20, badCIsNull= true, vaultSize= 41, vaultValue= 1502, escrowSize= 2, escrowValue= 20]
23:48:38:796: BVM: CHARGING --> CANCELING

23:48:38:800: BVM: ###  The transaction has been canceled. Remember to take your coins ###
23:48:38:804: BVM: Refunded a bag of coins of total value 20
23:48:38:804: BVM: [current= 0, price= 0, credit= 0, badCIsNull= true, vaultSize= 41, vaultValue= 1502, escrowSize= 0, escrowValue= 0]
23:48:38:804: BVM: CANCELING -> IDLE

23:48:38:810: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
23:48:38:812: USER: Received balance coin(s) of total value 20
23:48:38:812: USER: [choice= 0, bill= 0, advance= 0, cupIsNull= true, walletSize= 2, walletValue= 20, purseSize= 7, purseValue= 13]
23:48:38:812: USER: WAITING --> AWAY

23:49:8:810: BVM: ###  Welcome. Choose a beverage code to start a transaction: 1->Cocoa, 2->Coffee, 3->Orange, 4->Apple ###
```

Bibliography

[AG04] Abrahams, D., & Gurtovoy, A. (2004). C++ template metaprogramming: concepts, tools, and techniques from Boost and beyond. Pearson Education.

[AK03] Atkinson, C., & Kühne, T. (2003). Model-driven development: a metamodeling foundation. Software, IEEE, 20(5), 36-41.

[ALF+11] Annpureddy, Y., Liu, C., Fainekos, G., & Sankaranarayanan, S. (2011). S-taliro: A tool for temporal logic falsification for hybrid systems. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (pp. 254-257). Springer Berlin Heidelberg.

[AMT15] Aliyu, H. O., Maïga, O., & Traoré, M. K. (2015). A framework for discrete event systems enactment. In Proceedings of 29th European Simulation and Modeling Conference (pp. 149–156), ISBN: 978-9077381-908. EUROSIS-ETI.

[AMT16] Aliyu, H. O., Maïga, O., & Traoré, M. K. (2016). The high level language for system specification: A model-driven approach to systems engineering. International Journal of Modeling, Simulation, and Scientific Computing, 7(01), 1641003.

[ASK06] Apvrille, L., de Saqui-Sannes, P., & Khendek, F. (2006). TURTLE-P: a UML profile for the formal validation of critical and distributed systems. Software & Systems Modeling, 5(4), 449-466.

[ASL+01] Apvrille, L., de Saqui-Sannes, P., Lohr, C., Sénac, P., & Courtiat, J. P. (2001). A new UML profile for real-time system formal design and validation. In ≪ UML≫ 2001—The Unified Modeling Language. Modeling Languages, Concepts, and Tools (pp. 287-301). Springer Berlin Heidelberg.

[AT15a] Aliyu, H. O., Traoré, & M. K. (2015). Towards a unified framework for holistic study and analysis of discrete events systems. In Proceedings of The AUST International Conference on Technology -AUSTECH'15, October 12-13, 2015, Abuja, Nigeria

[AT15b] Aliyu, H. O., & Traoré, M. K. (2015). Toward an integrated framework for the simulation, formal analysis and enactment of discrete events systems models. In Proceedings of the 2015 Winter Simulation Conference- WSC'15 (pp. 3090-3091), December 6-9, 2015, Huntington Beach, CA, USA. IEEE Press.

[AT16] Aliyu, H. O., & Traoré, M. K. (2016, April). Integrated framework for model-driven systems engineering: a research roadmap. In *Proceedings of the Symposium on*

Theory of Modeling & Simulation (p. 28). Society for Computer Simulation International.

[BA95] Bruno, G., & Agarwal, R. (1995). Validating software requirements using operational models. In Objective Software Quality (pp. 78-93). Springer Berlin Heidelberg.

[BC10] Bone, M., & Cloutier, R. (2010). The Current State of Model Based Systems Engineering: Results from the OMG™ SysML Request for Information 2009. In Proceedings of the 8th Conference on Systems Engineering Research.

[BCE+08] Behrens, H., Clay, M., Efftinge, S., Eysholdt, M., Friese, P., Köhnlein, J., & Zarnekow, S. (2008). Xtext user guide. Available online at http://www.eclipse.org/Xtext/documentation/1_0_1/xtext.html. (last accessed July 30, 2016).

[BD14] Bocciarelli, P., & D'Ambrogio, A. (2014). Model-driven method to enable simulation-based analysis of complex systems. Chapter 2 in Gianni, D., D'Ambrogio, A., & Tolk, A. (Eds.) Modeling and Simulation-Based Systems Engineering Handbook (pp. 119-148). CRC Press.

[BDL04] Behrmann, G., David, A., & Larsen, K. G. (2004). A tutorial on uppaal. In Formal methods for the design of real-time systems (pp. 200-236). Springer Berlin Heidelberg.

[BDM14] Brooks, B.,Davidson, A., & M^{c}Gregor, I. (2014). The Evolving Relationship Between Simulation and Emulation: Faster than Real-Time Controls Testing. In *Proceedings of the 2014 Winter Simulation Conference* (pp. 4240-4249). IEEE Press.

[Béz04] Bézivin, J. (2004). In search of a basic principle for model driven engineering. Novatica. Journal, Special Issue, 5(2), 21-24.

[Béz05] Bézivin, J. (2005). On the unification power of models. Software & Systems Modeling, 4(2), 171-188.

[Béz06] Bézivin, J. (2006). Model driven engineering: An emerging technical space. In Generative and transformational techniques in software engineering (pp. 36-64). Springer Berlin Heidelberg.

[BFB07] Barbero, M., Fabro, M. D. D., & Bézivin, J. (2007). Traceability and provenance issues in global model management. ECMDA-TW, 7, 47-55.

[BG01] Bézivin, J., & Gerbé, O. (2001). Towards a precise definition of the OMG/MDA framework. In Automated Software Engineering, 2001.(ASE 2001). In Proceedings of 16th Annual International Conference on (pp. 273-280). IEEE.

[BGM+11] Bryant, B. R., Gray, J., Mernik, M., Clarke, P. J., France, R. B., & Karsai, G. (2011). Challenges and directions in formalizing the semantics of modeling languages. Comput. Sci. Inf. Syst., 8(2), 225-253.

[BJR+05] Bézivin, J., Jouault, F., Rosenthal, P., & Valduriez, P. (2005). Modeling in the large and modeling in the small. In Model Driven Architecture (pp. 33-46). Springer Berlin Heidelberg.

[BJV04] Bézivin, J., Jouault, F., & Valduriez, P. (2004). On the need for megamodels. In Proceedings of the OOPSLA/GPCE: Best Practices for Model-Driven Software Development workshop, 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications.

[BK11] Bergero, F., & Kofman, E. (2011). PowerDEVS: a tool for hybrid system modeling and real-time simulation. *Simulation*, *87*(1-2), 113-132.

[BKL08] Baier, C., Katoen, J. P., & Larsen, K. G. (2008). *Principles of model checking*. MIT press.

[BLC07] Boutin, O., L'Anton, A., & Cottenceau, B. (2007). Emulation as a means of designing an Inline-Control. In *IMSM07: International Modeling & Simulation Multiconference 2007*.

[BMC+12] Bousse, E., Mentré, D., Combemale, B., Baudry, B., & Katsuragi, T. (2012). Aligning SysML with the B method to provide V&V for systems engineering. In Proceedings of the Workshop on Model-Driven Engineering, Verification and Validation (pp. 11-16). ACM.

[BNB+07] Balasubramanian, D., Narayanan, A., van Buskirk, C., & Karsai, G. (2007). The graph rewriting and transformation language: GReAT. Electronic Communications of the EASST, 1.

[Bro04] Brown, A. W. (2004). Model driven architecture: Principles and practice. Software and Systems Modeling, 3(4), 314-327.

[BSD+12] Bajaj, M., Scott, A., Deming, D., Wickstrom, G., Spain, M. D., Zwemer, D., & Peak, R. (2012). Maestro–A model-based systems engineering environment for complex electronic systems. In INCOSE International Symposium (Vol. 22, No. 1, pp. 1999-2015).

[BWC13] Bonaventura, M., Wainer, G. A., & Castro, R. (2013). Graphical modeling and simulation of discrete-event systems with CD++ Builder. Simulation, 89(1), 4-27.

[CA95] Corbett, J. C., & Avrunin, G. S. (1995). Using integer programming to verify general safety and liveness properties. *Formal Methods in System Design*,*6*(1), 97-123.

[Car04] Carson II, J. S. (2004). Introduction to modeling and simulation. In Proceedings of the 36th conference on Winter simulation (pp. 9-16). IEEE Computer Society.

[CCG+00] Cimatti, A., Clarke, E., Giunchiglia, F., & Roveri, M. (2000). NuSMV: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, *2*(4), 410-425.

[CCG+02] Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., & Tacchella, A. (2002). Nusmv 2: An opensource tool for symbolic model checking. In *International Conference on Computer Aided Verification*(pp. 359-364). Springer Berlin Heidelberg.

[CCG+99] Cimatti, A., Clarke, E., Giunchiglia, F., & Roveri, M. (1999). NuSMV: A new symbolic model verifier. In *International conference on computer aided verification* (pp. 495-499). Springer Berlin Heidelberg.

[CDL+10] Chaudhuri, K., Doligez, D., Lamport, L., & Merz, S. (2010, July). Verifying safety properties with the TLA+ proof system. In *International Joint Conference on Automated Reasoning* (pp. 142-148). Springer Berlin Heidelberg.

[CDL+12] Cousineau, D., Doligez, D., Lamport, L., Merz, S., Ricketts, D., & Vanzetto, H. (2012). TLA+ proofs. In *International Symposium on Formal Methods*(pp. 147-154). Springer Berlin Heidelberg.

[CES86] Clarke, E. M., Emerson, E. A., & Sistla, A. P. (1986). Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Transactions on Programming Languages and Systems (TOPLAS), 8(2), 244-263.

[Çet13] Çetinkaya, D. K. (2013). Model Driven Development of Simulation Models: Defining and Transforming Conceptual Models into Simulation Models by Using Metamodels and Model Transformations. Doctoral Dissertation, TU Delft, Delft University of Technology.

[CGP99] Clarke, E. M., Grumberg, O., & Peled, D. (1999). *Model checking*. MIT press.

[CH03] Czarnecki, K., & Helsen, S. (2003). Classification of model transformation approaches. In Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture (Vol. 45, No. 3, pp. 1-17).

[CH06] Czarnecki, K., & Helsen, S. (2006). Feature-based survey of model transformation approaches. IBM Systems Journal, 45(3), 621-645.

[CL09] Cassandras, C. G., & Lafortune, S. (2009). Introduction to discrete event systems. Springer Science & Business Media.

[Cle01] Cleaveland, C. C., & Cleaveland, J. C. (2001). Program Generators with XML and Java. Prentice Hall PTR.

[Cri07] Cristiá, M. (2007). A TLA+ encoding of DEVS models. In Proceedings of International Modeling and Simulation Multiconference, Buenos Aires, Argentina (pp. 17-22).

[Cri08] Cristiá, M. (2008). Formalizing the Semantics of Modular DEVS Models with Temporal Logic. In Proceedings of 7e Conférence Francophone de MOdélisation et SIMulation - MOSIM'08 - du 31 mars au 2 avril 2008 - Paris - France.

[CS15] Capocchi, L., & Santucci, J. F. (2015). DEVSimPy: Interface graphique développée en langage Python et la librairie wxPython. In *Journées Développement Logiciel de l'Enseignement Supérieur et de la Recherche (JDEVS 2015)*.

[CSA+05] Chen, K., Sztipanovits, J., Abdelwalhed, S., & Jackson, E. (2005). Semantic anchoring with model transformations. In European Conference on Model Driven Architecture-Foundations and Applications (pp. 115-129). Springer Berlin Heidelberg.

[CSP+11] Capocchi, L., Santucci, J. F., Poggi, B., & Nicolai, C. (2011). DEVSimPy: A collaborative python software for modeling and simulation of DEVS systems. In *2nd International Track on Collaborative Modeling & Simulation-CoMetS'11* (pp. 6-pages). IEEE.

[CW+96] Clarke, E. M., Wing, J. M. et al. (1996). Formal methods: State of the art and future directions. ACM Computing Surveys (CSUR), 28(4), 626-643.

[CZ94] Chow, A. C. H., & Zeigler, B. P. (1994). Parallel DEVS: a parallel, hierarchical, modular, modeling formalism. In Proceedings of the 26th conference on Winter simulation (pp. 716-722). Society for Computer Simulation International.

[DAC98] Dwyer, M. B., Avrunin, G. S., & Corbett, J. C. (1998). Property specification patterns for finite-state verification. In *Proceedings of the second workshop on Formal methods in software practice* (pp. 7-15). ACM.

[DAC99] Dwyer, M. B., Avrunin, G. S., & Corbett, J. C. (1999). Patterns in property specifications for finite-state verification. In *Software Engineering, 1999. Proceedings of the 1999 International Conference on* (pp. 411-420). IEEE.

[Dav03] Davis, J. (2003). GME: the generic modeling environment. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (pp. 82-83). ACM.

[DC94] Dwyer, M. B., & Clarke, L. A. (1994). *Data flow analysis for verifying properties of concurrent programs* (Vol. 19, No. 5, pp. 62-75). ACM. doi:10.1145/193173.195295

[DF94] Dowson, M., & Fernström, C. (1994). Towards requirements for enactment mechanisms. In Software Process Technology (pp. 90-106). Springer Berlin Heidelberg.

[DV02] De Lara, J., & Vangheluwe, H. (2002). AToM3: A Tool for Multi-formalism and Meta-modelling. In *International Conference on Fundamental Approaches to Software Engineering* (pp. 174-188). Springer Berlin Heidelberg.

[DW04] des Rivières, J., & Wiegand, J. (2004). Eclipse: A platform for integrating development tools. IBM Systems Journal, 43(2), 371.

[ES06] Emerson, M., & Sztipanovits, J. (2006). Techniques for metamodel composition. In OOPSLA–6th Workshop on Domain Specific Modeling (pp. 123-139).

[Est07] Estefan, J. A. (2007). Survey of model-based systems engineering (MBSE) methodologies. Incose MBSE Focus Group, 25(8).

[Fav04] Favre, J. M. (2004). Towards a basic theory to model model driven engineering. In 3rd Workshop in Software Model Engineering, WiSME (pp. 262-271).

[Fav05a] Favre, J. M. (2005). Foundations of meta-pyramids: Languages vs. metamodels-episode II: Story of thotus the baboon1. In Dagstuhl Seminar Proceedings. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[Fav05b] Favre, J. M. (2005). Megamodeling and etymology-a story of words: From MED to MDE via MODEL in five milleniums. In In Dagstuhl Seminar on Transformation Techniques in Software Engineering, number 05161 in DROPS 04101. IFBI.

[FBT+14] Franceschini, R., Bisgambiglia, P. A., Touraille, L., Bisgambiglia, P., & Hill, D. (2014). A survey of modelling and simulation software frameworks using Discrete Event System Specification. In OASIcs-Open Access Series in Informatics (Vol. 43).

[FCS+03] France, R., Chosh, S., Song, E., & Kim, D. K. (2003). A metamodeling approach to pattern-based model refactoring. Software, IEEE, 20(5), 52-58.

[Fer94] Ferro, G. (1994). AMC: ACTL Model Checker. Reference Manual. *lEl-lnternal Report B4-47*.

[FKN+92] Finkelstein, A., Kramer, J., Nuseibeh, B., Finkelstein, L., & Goedicke, M. (1992). Viewpoints: A framework for integrating multiple perspectives in system development. International Journal of Software Engineering and Knowledge Engineering, 2(01), 31-57.

[FMS14] Friedenthal, S., Moore, A., & Steiner, R. (2014). A practical guide to SysML: the systems modeling language. Morgan Kaufmann.

[FN05] Favre, J. M., & Nguyen, T. (2005). Towards a megamodel to model software evolution through transformations. Electronic Notes in Theoretical Computer Science, 127(3), 59-74.

[Fow10] Fowler, M. (2010). Domain-specific languages. ISBN 978-0-321-71294-3, Pearson Education, Boston

[FP08] Fainekos, G. E., & Pappas, G. J. (2008). *A user guide for TaLiRo*. Technical report, Department of CIS, University of Pennsylvania.

[FPB+09] Fritzson, P., Pop, A., Broman, D., & Aronsson, P. (2009). Formal semantics based translator generation and tool development in practice. In Proceedings of 2009 Australian Software Engineering Conference (pp. 256-266). IEEE.

[FR07] France, R., & Rumpe, B. (2007). Model-driven development of complex software: A research roadmap. In 2007 Future of Software Engineering (pp. 37-54). IEEE Computer Society.

[FSU+12] Fainekos, G. E., Sankaranarayanan, S., Ueda, K., & Yazarel, H. (2012). Verification of automotive control applications using s-taliro. In *2012 American Control Conference (ACC)* (pp. 3567-3572). IEEE.

[GA09] Goeken, M., & Alter, S. (2009). Towards conceptual metamodeling of it governance frameworks approach-use-benefits. In System Sciences, 2009. HICSS'09. 42nd Hawaii International Conference on (pp. 1-10). IEEE.

[GCG+05] Groupe Conseil Général des Technologies de l'Information. (2005). La Politique Française Dans le Domaine du Calcul Scientifique, Rapport n° II.B.14.2004, Mars.

[GDT14] Gianni, D., D'Ambrogio, A., & Tolk, A. (Eds.). (2014). Modeling and Simulation-Based Systems Engineering Handbook. CRC Press.

[GG15] Goranko, V, &Galton, A. (2015). Temporal Logic. *The Stanford Encyclopedia of Philosophy*. url- http://plato.stanford.edu/archives/win2015/entries/logic-temporal/. Last accessed August 20, 2016.

[GHJ+95] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). Design patterns. Elements of reusable object-oriented software. Addison-Wesley.

[Gor12] Gordon, M. J. (2012). The denotational description of programming languages: an introduction. Springer Science & Business Media.

[Got05] Goth, G. (2005). Beware the March of this IDE: Eclipse is overshadowing other tool technologies. IEEE software, 22(4), 108-111.

[GRL05] Guruprasad, S., Ricci, R., & Lepreau, J. (2005). Integrated network experimentation using simulation and emulation. In *First International Conference on Testbeds and Research Infrastructures for the DEvelopment of NeTworks and COMmunities* (pp. 204-212). IEEE.

[Gron09] Gronback, R. C. (2009). *Eclipse modeling project: a domain-specific language (DSL) toolkit*. Pearson Education.

[GS03] Greenfield, J., & Short, K. (2003). Software factories: assembling applications with patterns, models, frameworks and tools. In Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (pp. 16-27). ACM.

[GS04] Greenfield, J., & Short, K. (2004). Software Factories: Assembling Applications with Patterns, Frameworks, Models and Tools. ISBN 978-0-471-20284-4, Wiley Publishing Inc., Indiana

[GS05] Groupe Simulation – Académie des Technologies. (2005). Enquête sur les Frontières de la Simulation Numérique en France. La Situation en France et Dans le Monde. Diagnostic et Propositions, Rapport de l'Académie des Technologies, Mai.

[Hal05] Hall, A. (2005). Realising the benefits of formal methods. In Formal Methods and Software Engineering (pp. 1-4). Springer Berlin Heidelberg.

[Hal07] Hall, A. (2007). Realising the Benefits of Formal Methods. J. UCS, 13(5), 669-678.

[Har78] Harrison, M. A. (1978). Introduction to formal language theory. Addison-Wesley Longman Publishing Co., Inc..

[Har87] Harel, D. (1987). Statecharts: A visual formalism for complex systems. Science of computer programming, 8(3), 231-274.

[HE07] Holmlid, S., & Evenson, S. (2007). Prototyping and enacting services: Lessons learned from human-centered methods. In Proceedings from the 10th Quality in Services conference, QUIS (Vol. 10).

[Hen90] Hennessy, M. (1990). The semantics of programming languages: an elementary introduction using structural operational semantics. John Wiley & Sons.

[HF04] Heaven, W., & Finkelstein, A. (2004). UML profile to support requirements engineering with KAOS. In Software, IEE Proceedings- (Vol. 151, No. 1, pp. 10-27). IET.

[HK06] Hong, K. J., & Kim, T. G. (2006). DEVSpecL: DEVS specification language for modeling, simulation and analysis of discrete event systems. Information and Software Technology, 48(4), 221-234.

[Hoa69] Hoare, C. A. R. (1969). An axiomatic basis for computer programming. Communications of the ACM, 12(10), 576-580.

[Hol05] Holmquist, L. E. (2005). Prototyping: Generating ideas or cargo cult design? Interactions, 12(2), 48-54.

[Hol97] Holzmann, G. J. (1997). The model checker SPIN. IEEE Transactions on software engineering, 23(5), 279.

[Hou12] Sqali Houssaini, M. (2012). Utilisation du formalisme DEVS pour la validation de comportements des systèmes à partir des scénarios UML. Doctoral dissertation, Aix-Marseille.

[HR00] Harel, D., & Rumpe, B. (2000). Modeling Languages: Syntax, Semantics and All That Stuff Part I: The Basic Stuff. Technical Report MCS00-16, The Weizmann Institute of Science, Rehovot, Israel.

[HR04] Harel, D., & Rumpe, B. (2004). Meaningful modeling: what's the semantics of" semantics"?. Computer, 37(10), 64-72.

[HRM07] Huang, E., Ramamurthy, R., & McGinnis, L. F. (2007). System and simulation modeling using SysML. In Proceedings of the 39th conference on Winter simulation (pp. 796-803). IEEE Press.

[HSG12] Hebig, R., Seibel, A., & Giese, H. (2012). On the unification of megamodels. Electronic Communications of the EASST, 42.

[HU79] Hopcroft, J. E., & Ullman, J. D. (1979). Introduction to automata theory, languages and computation. Addison-Wesley Longman Publishing Co., Inc.

[HWR+11] Hutchinson, J., Whittle, J., Rouncefield, M., & Kristoffersen, S. (2011). Empirical assessment of MDE in industry. In Proceedings of the 33rd International Conference on Software Engineering (pp. 471-480). ACM.

[HZ07] Hamri, M. E. A., & Zacharewicz, G. (2007). LSIS-DME: An environment for modeling and simulation of DEVS specifications. In AIS-CMS International modeling and simulation multiconference (pp. 55-60).

[IMT12] Ighoroje, U. B., Maïga, O., & Traoré, M. K. (2012). The DEVS-driven modeling language: syntax and semantics definition by meta-modeling and graph transformation. In Proceedings of the 2012 Symposium on Theory of Modeling and

Simulation-DEVS Integrative M&S Symposium (p. 49). Society for Computer Simulation International.

[ISO02] ISO/IEC FDIS 13568:2002(E) (2002). Information technology—Z formal specification notation—Syntax, type system and semantics.

[JAB+06] Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I., & Valduriez, P. (2006). ATL: a QVT-like transformation language. In Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications (pp. 719-720). ACM.

[Jac97] Jacky, J. (1997). The way of Z: practical programming with formal methods. Cambridge University Press.

[JB06] Jouault, F., & Bézivin, J. (2006). KM3: a DSL for Metamodel Specification. In *International Conference on Formal Methods for Open Object-Based Distributed Systems* (pp. 171-185). Springer Berlin Heidelberg.

[JN14] Jeston, J., & Nelis, J. (2014). Business process management. Routledge.

[KAS11] Knorreck, D., Apvrille, L., & de Saqui-Sannes, P. (2011). TEPE: a SysML language for time-constrained property modeling and formal verification. ACM SIGSOFT Software Engineering Notes, 36(1), 1-8.

[KBA02] Kurtev, I., Bézivin, J., & Akşit, M. (2002). Technological spaces: An initial appraisal.

[KCS03] Kuhn, D. R., Craigen, D., & Saaltink, M. (2003). Practical application of formal methods in modeling and simulation. In Summer Computer Simulation Conference (pp. 726-731). Society for Computer Simulation International; 1998.

[Ken02] Kent, S. (2002). Model driven engineering. In Integrated formal methods(pp. 286-298). Springer Berlin Heidelberg.

[KES03] Kefalas, P., Eleftherakis, G., & Sotiriadou, A. (2003). Developing tools for formal methods. In Proceedings of the 9th Panhellenic Conference in Informatics (pp. 625-639).

[KG06] Klein, F., & Giese, H. (2006). *Integrated Visual Specification of Structural and Temporal Properties*. Technical Report tr-ri-06-277, Computer Science Department, University of Paderborn.

[KG07] Klein, F., & Giese, H. (2007). Joint structural and temporal property specification using timed story scenario diagrams. In *International Conference on Fundamental Approaches to Software Engineering* (pp. 185-199). Springer Berlin Heidelberg.

[KGJ10] Kouvas, G., Grefen, P., & Juan, A. (2010). Business Process Enactment. In Dynamic Business Process Formation for Instant Virtual Enterprises (pp. 113-132). Springer London.

[Kla08] Klatt, B. (2008). Xpand: A closer look at the model2text transformation language. Language, 10(16), 2008.

[Kle08] Kleppe, A. (2008). Software language engineering: creating domain-specific languages using metamodels. Pearson Education.

[KLP03] Kofman, E., Lapadula, M., & Pagliero, E. (2003). PowerDEVS: A DEVS–based environment for hybrid system modeling and simulation. *School of Electronic Engineering, Universidad Nacional de Rosario, Tech. Rep. LSD0306*.

[KLV05] Klint, P., Lämmel, R., & Verhoef, C. (2005). Toward an engineering discipline for grammarware. ACM Transactions on Software Engineering and Methodology (TOSEM), 14(3), 331-380.

[KNT+14] Kinoshita, S., Nishimura, H., Takamura, H., & Mizuguchi, D. (2014). Describing software specification by combining SysML with the B method. In Software Reliability Engineering Workshops (ISSREW), 2014 IEEE International Symposium on (pp. 146-151). IEEE.

[Koz83] Kozen, D. (1983). Results on the propositional μ-calculus. *Theoretical computer science*, *27*(3), 333-354.

[KP88] Krasner, G. E., & Pope, S. T. (1988). A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *Journal of object oriented programming*, *1*(3), 26-49.

[KSE09] Kim, S., Sarjoughian, H. S., & Elamvazhuthi, V. (2009). DEVS-suite: a simulator supporting visual experimentation design and behavior monitoring. In*Proceedings of the 2009 Spring Simulation Multiconference* (p. 161). Society for Computer Simulation International.

[KT08] Kelly, S., & Tolvanen, J. P. (2008). Domain-specific modeling: enabling full code generation. ISBN 978-0-470-03666-2, John Wiley & Sons, New Jersey.

[Küh06] Kühne, T. (2006). Matters of (meta-) modeling. Software & Systems Modeling,5(4), 369-385.

[KW07] Kurpjuweit, S., & Winter, R. (2007). Viewpoint-based Meta Model Engineering. In EMISA2007, pp. 143-159.

[KWB03] Kleppe, A. G., Warmer, J. B., & Bast, W. (2003). MDA explained: the model driven architecture: practice and promise. Addison-Wesley Professional.

[Lam00] Lamsweerde, A. V. (2000). Formal specification: a roadmap. In Proceedings of the Conference on the Future of Software Engineering (pp. 147-159). ACM.

[Lam02] Lamport, L. (2002). Specifying systems: the TLA+ language and tools for hardware and software engineers. Addison-Wesley Longman Publishing Co., Inc.

[Lam77] Lamport, L. (1977). Proving the correctness of multiprocess programs. *IEEE transactions on software engineering*, (2), 125-143.

[Lam94] Lamport, L. (1994). The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, *16*(3), 872-923.

[Lan12] Lano, K. (2012). The B language and method: a guide to practical formal development. Springer Science & Business Media.

[LCA04] Lano, K., Clark, D., & Androutsopoulos, K. (2004). UML to B: Formal verification of object-oriented models. In Integrated Formal Methods (pp. 187-206). Springer Berlin Heidelberg.

[LMB+01] Ledeczi, A., Maroti, M., Bakay, A., Karsai, G., Garrett, J., Thomason, C., Nordstrom, G., Sprinkle, J., & Volgyesi, P. (2001, May). The generic modeling environment. In Workshop on Intelligent Signal Processing, Budapest, Hungary (Vol. 17, p. 114).

[LP99] Lilius, J., & Paltor, I. P. (1999, October). vUML: A tool for verifying UML models. In Automated Software Engineering, 1999. 14th IEEE International Conference on. (pp. 255-258). IEEE.

[LSM+10] Laleau, R., Semmak, F., Matoussi, A., Petit, D., Hammad, A., & Tatibouet, B. (2010). A first attempt to combine SysML requirements diagrams and B. Innovations in Systems and Software Engineering, 6(1-2), 47-54.

[LV02] De Lara, J., & Vangheluwe, H. (2002). AToM3: A Tool for Multi-formalism and Meta-modelling. In FASE (Vol. 2, pp. 174-188).

[LZ13] Lämmel, R., & Zaytsev, V. (2013). Language support for megamodel renarration. In XM 2013–Extreme Modeling Workshop (p. 38).

[Mar97] Maria, A. (1997). Introduction to modeling and simulation. In Proceedings of the 29th conference on Winter simulation (pp. 7-13). IEEE Computer Society.

[Mat03] Mather, J. (2003). *The DEVSJAVA Simulation Viewer: A modular GUI that visualizes the structure and behavior of hierarchical DEVS models* (Masters Thesis, University of Arizona).

[MC11] Medina, J. L., & Cuesta, A. G. (2011). Model-based analysis and design of real-time distributed systems with Ada and the UML profile for MARTE. In Reliable Software Technologies-Ada-Europe 2011 (pp. 89-102). Springer Berlin Heidelberg.

[MD08] Mohagheghi, P., & Dehlen, V. (2008). Where is the proof?-a review of experiences from applying mde in industry. In Model Driven Architecture–Foundations and Applications (pp. 432-443). Springer Berlin Heidelberg.

[MD12] Mittal, S., Douglass, S. A. (2012). DEVSML 2.0: The language and the stack. In. Proceedings of the 2012 Symposium on Theory of Modeling and Simulation-DEVS Integrative M&S Symposium (p. 17). Society for Computer Simulation International.

[MDL+14] Meyers, B., Deshayes, R., Lucio, L., Syriani, E., Vangheluwe, H., & Wimmer, M. (2014). ProMoBox: A framework for generating domain-specific property languages. In International Conference on Software Language Engineering (pp. 1-20). Springer International Publishing.

[Mel04] Mellor, S. J. (2004). MDA distilled: principles of model-driven architecture. Addison-Wesley Professional.

[Mel88] Melliar-Smith, P. M. (1988). A graphical representation of interval logic. In *International Conference on Concurrency* (pp. 106-120). Springer Berlin Heidelberg.

[Men06] Mens, T. (2006). On the use of graph transformations for model refactoring. In Generative and transformational techniques in software engineering (pp. 219-257). Springer Berlin Heidelberg.

[Men16] Mentré, D. (2016). SysML2B: Automatic Tool for B Project Graphical Architecture Design Using SysML. In International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z (pp. 308-311). Springer International Publishing.

[Mey16] Meyers, B. (2016). A Multi-Paradigm Modelling Approach to Design and Evolution of Domain-Specific Modeling Languages, Doctoral dissertation, Universiteit Antwerpen.

[MFM+05] Miller, T., Freitas, L., Malik, P., & Utting, M. (2005). CZT support for Z extensions. In International Conference on Integrated Formal Methods (pp. 227-245). Springer Berlin Heidelberg.

[MG06] Mens, T., & Van Gorp, P. (2006). A taxonomy of model transformation. Electronic Notes in Theoretical Computer Science, 152, 125-142.

[MGr02] M^{c}Gregor, I. (2002). The relationship between simulation and emulation. In *Simulation Conference, 2002. Proceedings of the Winter* (Vol. 2, pp. 1683-1688). IEEE.

[MGV+06] Mens, T., Van Gorp, P., Varró, D., & Karsai, G. (2006). Applying a model transformation taxonomy to graph transformation technology. Electronic Notes in Theoretical Computer Science, 152, 143-159.

[MH05] Muller, P. A., & Hassenforder, M. (2005). HUTN as a bridge between modelware and grammarware-an experience report. In WISME Workshop, MODELS/UML.

[Mit07] Mittal, S. (2007). DEVS unified process for integrated development and testing of service oriented architectures, Doctoral dissertation, University of Arizona.

[MIT12] Maïga, O., Ighoroje, U. B., & Traoré, M. K. (2012). DDML: A Support for Communication in M& S. In Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), 2012 IEEE 21st International Workshop on (pp. 238-243). IEEE.

[MJL+06] Musset, J., Juliot, É., Lacrampe, S., Piers, W., Brun, C., Goubet, L., Lussaud, Y., & Allilaire, F. (2006). Acceleo user guide. http://acceleo. org/doc/obeo/en/acceleo-2.6-user-guide. pdf, 2. Accessed June 7, 2016.

[MM13a] Mittal, S., & Martín, J. L. R. (2013). Model-driven systems engineering for netcentric system of systems with DEVS unified process. In Proceedings of the 2013 Winter Simulation Conference: Simulation (pp. 1140-1151). IEEE Press.

[MM13b] Mittal, S., & Martín, J. L. R. (2013). Netcentric system of systems engineering with DEVS unified process. CRC Press.

[MS04] Markey, N., & Schnoebelen, P. (2004). TSMV: A symbolic model checker for quantitative analysis of systems. In *Proceedings of the 1st International Conference on Quantitative Evaluation of Systems (QEST'04)* (pp. 330-331). IEEE Computer Society Press.

[MS97] Meisels, I., & Saaltink, M. (1997). The Z/EVES reference manual (for version 1.5). Reference manual, ORA Canada.

[MU05] Malik, P., & Utting, M. (2005). CZT: A framework for Z tools. In International Conference of B and Z Users (pp. 65-84). Springer Berlin Heidelberg.

[MWB+13]Meyers, B., Wimmer, M., Vangheluwe, H., & Denil, J. (2013). Towards domain-specific property languages: The ProMoBox approach. In Proceedings of the 2013 ACM Workshop on Domain-specific Modeling (pp. 39-44). ACM.

[NDA10] Nikolaidou, M., Dalakas, V., & Anagnostopoulos, D. (2010). Integrating Simulation Capabilities in SysML using DEVS. In Proceedings of IEEE Systems Conference.

[NDK+07] Nikolaidou, M., Dalakas, V., Kapos, G. D., Mitsi, L., & Anagnostopoulos, D. (2007). A UML2. 0 profile for DEVS: Providing code generation capabilities for simulation. In SEDE (pp. 314-319).

[NDM+08] Nikolaidou, M., Dalakas, V., Mitsi, L., Kapos, G. D., & Anagnostopoulos, D. (2008). A SysML profile for classical DEVS simulators. In The Third International Conference on Software Engineering Advances, 2008. ICSEA'08. (pp. 445-450). IEEE.

[NN92] Nielson, H. R., & Nielson, F. (1992). Semantics with Applications: A Formal Introduction. John Wiley & Sons, Inc. New York, NY, USA. ISBN:0-471-92980-8

[NSF06] National Science Foundation. (2006). Simulation-based Engineering Science. Revolutionizing Engineering Science Through Simulation, NSF Report. May

[OF07] Ottensooser, A., & Fekete, A. (2007). An enactment-engine based on use-cases.In Business Process Management (pp. 230-245). Springer Berlin Heidelberg.

[OMG04] Object Management Group (2004). The Meta Object Facility (MOF) Core Specification. Version 2.0, OMG, http://www.omg.org.

[OMG15] Object Management Group (2015). XML Metadata Interchange (XMI) Specification, Version 2.5.1. URL: http://www.omg.org/spec/XMI/2.5.1. last accessed 3rd September, 2016.

[PAF07] Pop, A., Akhvlediani, D., & Fritzson, P. (2007). Integrated UML and modelica system modeling with ModelicaML in Eclipse. In Proceedings of the International Conference on Software Engineering and Applications (pp. 557-563). ACTA Press.

[Pai97] Paige, R. F. (1997). A meta-method for formal method integration. In International Symposium of Formal Methods Europe (pp. 473-494). Springer Berlin Heidelberg.

[PBR09] Perovich, D., Bastarrica, M. C., & Rojas, C. (2009). Model-driven approach to software architecture design. In Proceedings of the 2009 ICSE Workshop on Sharing and Reusing Architectural Knowledge (pp. 1-8). IEEE Computer Society.

[PIT+05] President’s Information Technology Advisory Committee. (2005). Computational Science: Ensuring America’s Competitiveness, Report to the President.

[Plo04] Plotkin, G. D. (2004). A structural approach to operational semantics. J. Logic and Algebraic Programming, 60(61), 17-139.

[Pnu77] Pnueli, A. (1977). The temporal logic of programs. In 18th Annual Symposium on Foundations of Computer Science, (pp. 46-57). IEEE.

[RJB04] Rumbaugh, J., Jacobson, I., & Booch, G. (2004). Unified Modeling Language Reference Manual, The. Pearson Higher Education.

[RJM+09] Risco-Martín, J. L., Jesús, M., Mittal, S., & Zeigler, B. P. (2009). eUDEVS: Executable UML with DEVS theory of modeling and simulation. Simulation,85(11-12), 750-777.

[RMZ07] Risco-Martın, J. L., Mittal, S., Zeigler, B. P., & Jesús, M. (2007). From UML state charts to DEVS state machines using XML. In Proceedings of the workshop on multi-paradigm modeling: concepts and tools, Nashville, TN.

[RPC+12] Reichwein, A., Paredis, C. J., Canedo, A., Witschel, P., Stelzig, P. E., Votintseva, A., & Wasgint, R. (2012). Maintaining consistency between system architecture and dynamic system models with SysML4Modelica. In Proceedings of the 6th International Workshop on Multi-Paradigm Modeling (pp. 43-48). ACM.

[RŞ12] Roşu, G., & Ştefănescu, A. (2012). Towards a unified theory of operational and axiomatic semantics. In International Colloquium on Automata, Languages, and Programming (pp. 351-363). Springer Berlin Heidelberg.

[RW89] Ramadge, P. J., & Wonham, W. M. (1989). The control of discrete event systems. In Proceedings of the IEEE, 77(1), 81-98.

[RWB97] Raje, R. R., Williams, J. I., & Boyles, M. (1997). Asynchronous remote method invocation (ARMI) mechanism for Java. *Concurrency - Practice and Experience*, *9*(11), 1207-1211.

[Saa03] Saaltink, M. (2003). The Z/EVES 2.2 mathematical toolkit. Ottawa, Canada: ORA Canada.

[Saa97] Saaltink, M. (1997, April). The Z/EVES system. In International Conference of Z Users (pp. 72-85). Springer Berlin Heidelberg.

[SAB09] Shah, S. M., Anastasakis, K., & Bordbar, B. (2009). From UML to Alloy and back again. In Models in Software Engineering (pp. 158-171). Springer Berlin Heidelberg.

[SB06] Snook, C., & Butler, M. (2006). UML-B: Formal modeling and design aided by UML. ACM Transactions on Software Engineering and Methodology (TOSEM), 15(1), 92-122.

[SB08] Snook, C., & Butler, M. (2008). UML-B and Event-B: an integration of languages and tools.

[SBM+08] Steinberg, D., Budinsky, F., Merks, E., & Paternostro, M. (2008). EMF: eclipse modeling framework. Pearson Education.

[Sch01] Schiess, C. (2001). Emulation: debug it in the lab-not on the floor. In *Simulation Conference, 2001. Proceedings of the Winter* (Vol. 2, pp. 1463-1465). IEEE.

[Sch06] Schmidt, D. C. (2006). Guest Editor's Introduction: Model-Driven Engineering. COMPUTER, 39(2), 0025-31.

[Sch09] Schamai, W. (2009). Modelica modeling language (ModelicaML): A UML profile for Modelica. A Technical Report in Computer and Information Science, Linköping Electronic Press.

[Sch96] Schmidt, D. A. (1996). Programming language semantics. ACM Computing Surveys (CSUR), 28(1), 265-267.

[Sei03] Seidewitz, E. (2003). What models mean. IEEE software, 20(5), 26.

[Sel03] Selic, B. (2003). The pragmatics of model-driven development. IEEE software, 20(5), 19.

[Sel09] Selic, B. (2009). The theory and practice of modeling language design for model-based software engineering—a personal perspective. In International Summer School on Generative and Transformational Techniques in Software Engineering (pp. 290-321). Springer Berlin Heidelberg.

[Sel12] Selic, B. (2012). What will it take? A view on adoption of model-based methods in practice. Software & Systems Modeling, 11(4), 513-526.

[SFP+09] Schamai, W., Fritzson, P., Paredis, C., & Pop, A. (2009). Towards unified system modeling and simulation with ModelicaML: modeling of executable behavior using graphical notations. In Proceedings of the 7th International Modelica Conference; (pp. 612-621). Linköping University Electronic Press.

[SK03] Sendall, S., & Kozaczynski, W. (2003). Model transformation the heart and soul of model-driven software development (No. LGL-REPORT-2003-007).

[SK10] Song, H. S., & Kim, T. G. (2010). DEVS diagram revised: a structured approach for DEVS modeling. In Proc. European Simulation Conference (Eurosis, Belgium, 2010) (pp. 94-101).

[SK95] Slonneger, K., & Kurtz, B. L. (1995). Formal syntax and semantics of programming languages: A Laboratory Approach (Vol. 340). Reading: Addison-Wesley.

[SK97] Sztipanovits, J., & Karsai, G. (1997). Model-integrated computing. Computer,30(4), 110-111.

[SLT+91] Smolander, K., Lyytinen, K., Tahvanainen, V. P., & Marttiin, P. (1991). MetaEdit - a flexible graphical environment for methodology modelling. In*International Conference on Advanced Information Systems Engineering* (pp. 168-193). Springer Berlin Heidelberg.

[SME09] Salay, R., Mylopoulos, J., & Easterbrook, S. (2009). Using macromodels to manage collections of related models. In International Conference on Advanced Information Systems Engineering (pp. 141-155). Springer Berlin Heidelberg.

[Smi12] Smith, G. (2012). The Object-Z specification language (Vol. 1). Springer Science & Business Media.

[Smi92] Smith, G. (1992). *An Object-Oriented Approach to Formal Specification* (Doctoral dissertation, University of Queensland).

[Spi88] Spivey, J. M. (1988). Understanding Z: a specification language and its formal semantics (Vol. 3). Cambridge University Press.

[Spi92] Spivey, J. M. (1992). The Z notation: A reference manual. *Prentice Hall*.

[Spr04] Sprinkle, J. (2004). Model-integrated computing. Potentials, IEEE, 23(1), 28-30.

[Sto77] Stoy, J. E. (1977). Denotational semantics: the Scott-Strachey approach to programming language theory. MIT press.

[SV09] Seck, M., & Verbraeck, A. (2009). DEVS in DSOL: adding devs operational semantics to a generic event-scheduling simulation environment. In Proceedings of the 2009 Summer Computer Simulation Conference (pp. 261-266). Society for Modeling & Simulation International.

[SV11] Shaikh, R., & Vangheluwe, H. (2011). Transforming UML2. 0 class diagrams and statecharts to atomic DEVS. In Proceedings of the 2011 Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium (pp. 205-212). Society for Computer Simulation International.

[SVM+13] Syriani, E., Vangheluwe, H., Mannadiar, R., Hansen, C., Van Mierlo, S., & Ergin, H. (2013). AToMPM: A Web-based Modeling Environment. In Demos/Posters /StudentResearch@ MoDELS (pp. 21-25).

[SWA+05] Sun, J., Wang, H., Athauda, S., & Sheik, T. (2005). SVG web environment for Z specification language. In *International Conference on Formal Engineering Methods* (pp. 480-494). Springer Berlin Heidelberg.

[SZ98] Sarjoughian, H. S., & Zeigler, B. R. (1998). DEVSJAVA: Basis for a DEVS-based collaborative M&S environment. *Simulation Series*, *30*, 29-36.

[SZC+13] Seo, C., Zeigler, B. P., Coop, R., & Kim, D. (2013). DEVS modeling and simulation methodology with ms4me software. In Symposium on Theory of Modeling and Simulation-DEVS (TMS/DEVS).

[TB11] Tanriöver, Ö. Ö., & Bilgen, S. (2011). A framework for reviewing domain specific conceptual models. Computer Standards & Interfaces, 33(5), 448-464.

[TB15] Trojet, W., & Berradia, T. (2015). System Reliability using Simulation Models and Formal Methods. International Journal of Computer Applications, 132(17).

[TFH09] Trojet, M. W., Frydman, C., & Hamri, M. E. A. (2009). Practical application of lightweight Z in DEVS framework. In Proceedings of the 2009 Spring Simulation Multiconference (p. 154). Society for Computer Simulation International.

[TH14] Tolk, A., & Hughes, T. K. (2014). Systems engineering, architecture, and simulation. Chapter 2 in Gianni, D., D'Ambrogio, A., & Tolk, A. (Eds.) Modeling and Simulation-Based Systems Engineering Handbook (pp. 11-41). CRC Press.

[TM06] Traoré, M. K., & Muzy, A. (2006). Capturing the dual relationship between simulation models and their context. Simulation Modelling Practice and Theory, 14(2), 126-142.

[TM95] Toyn, I., & McDermid, J. A. (1995). CADiZ: An architecture for Z tools and its implementation. Softw., Pract. Exper., 25(3), 305-330.

[Tou12] Touraille, L. (2012). Application of Model-Driven Engineering and Metaprogramming to EVS Modeling & Simulation. Doctoral dissertation, Université Blaise Pascal-Clermont-Ferrand II.

[Tra05] Traoré, M. K. (2005). Combining DEVS and logic. In Open international Conference on Modelling and Simulation-OICMS (pp. 307-317).

[Tra06a] Traoré, M. K. (2006). Analyzing static and temporal properties of simulation models. In Proceedings of the 38th conference on Winter simulation(pp. 897-904). Winter Simulation Conference.

[Tra06b] Traoré, M. K. (2006). Making DEVS models amenable to formal analysis. In Proceedings of the 2006 Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium. Society for Computer Simulation International.

[Tra08] Traoré, M. K. (2008). SimStudio: a next generation modeling and simulation framework. In Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops (p. 67). ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[Tra09] Traoré, M. K. (2009). A graphical notation for DEVS. In Proceedings of the 2009 Spring Simulation Multiconference (p. 162). Society for Computer Simulation International.

[Tru06] Truyen, F. (2006). The Fast Guide to Model Driven Architecture The Basics of Model Driven Architecture. Cephas Consulting Corp. url: http://www.omg.org/mda/mda_files/Cephas_MDA_Fast_Guide.pdf

[TSF09] Trojet, M. W., Sqali, M., Frydman, C., & Torres, L. (2009). MSC scenarios analysis via simulation and formal verification techniques. In Proceedings of the 2009 Grand Challenges in Modeling & Simulation Conference (pp. 35-42). Society for Modeling & Simulation International.

[TTH09] Touraille, L., Traoré, M. K., & Hill, D. R. (2009). A mark-up language for the storage, retrieval, sharing and interoperability of DEVS models. In Proceedings of the 2009 Spring Simulation Multiconference (p. 163). Society for Computer Simulation International.

[TTH10] Touraille, L., Traoré, M. K., & Hill, D. R. (2010). Enhancing DEVS simulation through template metaprogramming: DEVS-MetaSimulator. In Proceedings of the 2010 Summer Computer Simulation Conference (pp. 394-402). Society for Computer Simulation International.

[TTH11] Touraille, L., Traoré, M. K., & Hill, D. R. (2011). A model-driven software environment for modeling, simulation and analysis of complex systems. In Proceedings of the 2011 Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium (pp. 229-237). Society for Computer Simulation International.

[UTS+03] Utting, M., Toyn, I., Sun, J., Martin, A., Dong, J. S., Daley, N., & Currie, D. (2003). ZML: XML support for standard Z. In ZB2003: International Conference of B and Z Users (pp. 437-456). Springer Berlin Heidelberg.

[Van00] Vangheluwe, H. L. (2000). DEVS as a common denominator for multi-formalism hybrid systems modelling. In Computer-Aided Control System Design, 2000. CACSD 2000. IEEE International Symposium on (pp. 129-134). IEEE.

[VDW12] Visser, W., Dwyer, M. B., & Whalen, M. (2012). The hidden models of model checking. Software & Systems Modeling, 11(4), 541-555.

[VSB+06] Viehl, A., Schönwald, T., Bringmann, O., & Rosenstiel, W. (2006). Formal performance analysis and simulation of UML/SysML models for ESL design. In Proceedings of the conference on Design, automation and test in Europe (pp. 242-247). European Design and Automation Association.

[VTW03] Van Der Aalst, W. M., Ter Hofstede, A. H., & Weske, M. (2003). Business process management: A survey. In Business process management (pp. 1-12). Springer Berlin Heidelberg.

[Wai02] Wainer, G. (2002). CD++: a toolkit to develop DEVS models. Software: Practice and Experience, 32(13), 1261-1306.

[War07] Warmer, J. (2007). A model driven software factory using domain specific languages. In Model Driven Architecture-Foundations and Applications(pp. 194-203). Springer Berlin Heidelberg.

[WC01] Woodcock, J., & Cavalcanti, A. (2001). A concurrent language for refinement. In Proceedings of the 5th Irish conference on Formal Methods (pp. 93-115). British Computer Society.

[WCD01] Wainer, G., Christen, G., & Dobniewski, A. (2001). Defining DEVS models with the CD++ toolkit. In Proceedings of ESS (pp. 633-637).

[WD96] Woodcock, J., & Davies, J. (1996). Using Z: specification, refinement, and proof (Vol. 39). Englewood Cliffs: Prentice Hall.

[Weg72] Wegner, P. (1972). Operational semantics of programming languages. ACM SIGACT News, 7(14), 128-141.

[WFH09] Trojet, M. W., Frydman, C., & Hamri, M. E. A. (2009). Practical application of lightweight Z in DEVS framework. In Proceedings of the 2009 Spring Simulation Multiconference (p. 154). Society for Computer Simulation International.

[Whi04] White, S. A. (2004). Introduction to BPMN. IBM Cooperation, 2(0).

[WHR14] Whittle, J., Hutchinson, J., & Rouncefield, M. (2014). The state of practice in model-driven engineering. Software, IEEE, 31(3), 79-85.

[Win90] Wing, J. M. (1990). A specifier's introduction to formal methods. Computer, 23(9), 8-22.

[Wir96] Wirth, N. (1996). Extended Backus-Naur Form (EBNF). ISO/IEC, 14977, 2996.

[WK05] Wimmer, M., & Kramler, G. (2005). Bridging grammarware and modelware. In Satellite Events at the MoDELS 2005 Conference (pp. 159-168). Springer Berlin Heidelberg.

[WW06] Weigert, T., & Weil, F. (2006). Practical experiences in using model-driven engineering to develop trustworthy computing systems. In Sensor Networks, Ubiquitous, and Trustworthy Computing, 2006. IEEE International Conference on (Vol. 1, pp. 8-pp). IEEE.

[YHF14] Yacoub, A., Hamri, M., & Frydman, C. (2014). A Method for Improving the Verification and Validation of Systems by the Combined Use of Simulation and Formal Methods. In Proceedings of the 2014 IEEE/ACM 18th International Symposium on Distributed Simulation and Real Time Applications(pp. 155-162). IEEE Computer Society.

[YJ07] Yang, Z., & Jiang, M. (2007). Using Eclipse as a tool-integration platform for software development. IEEE Software, 24(2), 87.

[ZDD06] Zito, A., Diskin, Z., & Dingel, J. (2006). Package merge in UML2: Practice vs. theory?. In *International Conference on Model Driven Engineering Languages and Systems* (pp. 185-199). Springer Berlin Heidelberg.

[Zei76] Zeigler, B. P. (1976). Theory of modeling and simulation. John Wiley & Sons. Inc., New York, NY.

[Zin05] Zinoviev, D. (2005). Mapping DEVS Models onto UML Models. In DEVS Symposium, Spring Simulation Multiconference.

[ZLG05] Zhang, J., Lin, Y., & Gray, J. (2005). Generic and domain-specific model refactoring using a model transformation engine. In Model-driven Software Development (pp. 199-217). Springer Berlin Heidelberg.

[ZN16] Zeigler, B. P., & Nutaro, J. J. (2016). Towards a framework for more robust validation and verification of simulation models for systems of systems. The Journal of Defense Modeling and Simulation: Applications, Methodology, Technology, DOI: 10.1177/1548512914568657

[ZPK00] Zeigler, B. P., Praehofer, H., & Kim, T. G. (2000). Theory of modeling and simulation: integrating discrete event and continuous complex dynamic systems. Academic press.

[ZS13] Zeigler, B.P., Sarjoughian, H. S (2013). Guide to modeling and simulation of systems of systems. Springer Science & Business Media.