

A domain-specific language approach to hybrid cps modelling

KLIKOVITS, Stefan

Abstract

The recent advent of cyber-physical systems (CPSs) in end-user applications extends the need for sophisticated model creation, simulation and system verification to new application areas. CPSs such as smart homes seamlessly integrate technology into every-day life, rendering their safety and correctness paramount. The intricacy of these systems' modelling stems from the merging of two opposing views: While flows of physical energy are mostly described using mathematical methods such as differential equations, engineered applications are usually best expressed using discrete formalisms. This thesis describes the creation of the Continuous REactive SysTems language (CREST), a domain-specific language (DSL) dedicated to the combined modelling of physical resource flows and engineered behaviour. The language coherently merges architectural concerns, reactive dataflow and non-determinism. Its Python implementation allows convenient system modelling and supports advanced concerns such as the simulation and formal verification of hybrid systems based on sound theoretical foundations.

Reference

KLIKOVITS, Stefan. *A domain-specific language approach to hybrid cps modelling*.
Thèse de doctorat : Univ. Genève, 2019, no. Sc. 5354

DOI : [10.13097/archive-ouverte/unige:121355](https://doi.org/10.13097/archive-ouverte/unige:121355)

URN : [urn:nbn:ch:unige-1213552](https://nbn-resolving.org/urn:nbn:ch:unige-1213552)

Available at:

<http://archive-ouverte.unige.ch/unige:121355>

Disclaimer: layout of this document may differ from the published version.



UNIVERSITÉ
DE GENÈVE

UNIVERSITÉ DE GENÈVE FACULTÉ DES SCIENCES

Département d'Informatique Professeur D. Buchs

A Domain-Specific Language Approach to Hybrid CPS Modelling

THÈSE

présentée à la Faculté des sciences de l'Université de Genève
pour obtenir le grade de

Docteur ès sciences, mention informatique

par

Stefan Klikovits
de
Autriche

Thèse No 5354

Genève
Atelier d'impression ReproMail
2019



**UNIVERSITÉ
DE GENÈVE**

FACULTÉ DES SCIENCES

DOCTORAT ÈS SCIENCES, MENTION INFORMATIQUE

Thèse de Monsieur Stefan KLIKOVITS

intitulée :

**«A Domain-Specific Language Approach to
Hybrid CPS Modelling»**

La Faculté des sciences, sur le préavis de Monsieur D. BUCHS, professeur ordinaire et directeur de thèse (Département d'informatique), Monsieur G. FALQUET, professeur associé (Faculté d'économie et de Management, Information Science Institute), Monsieur M. AMRANI, docteur (Research center in information system engineering, Faculty of computer sciences, University of Namur, Belgium), Monsieur M. WIMMER, professeur (Institutsvorstand, Institut für Wirtschaftsinformatik, Johannes Kepler Universität, Linz, Österreich) et Monsieur J. DENIL, professeur (Electronics-ICT, University of Antwerp, Belgium), autorise l'impression de la présente thèse, sans exprimer d'opinion sur les propositions qui y sont énoncées.

Genève, le 14 juin 2019

Thèse - 5354 -

Le Doyen

*Für meine Familie –
Ich wäre weder wer ich bin, noch wo ich bin,
wäre es nicht wegen der Möglichkeiten, Unterstützung
und Liebe, die ihr mir gegeben habt.*

*To Paul –
Thank you for all your help, support and friendship.*

Acknowledgements

This thesis concludes a long, fulfilling journey that formed me as a researcher and as a person. I am delighted to have found so many lovely people who encouraged me and helped me reach my destination.

First, I would like to express my gratitude to Professor Didier Buchs. You trusted me in finding my own way and were available whenever I needed guidance. Thank you for giving me the opportunity to grow as a researcher and to travel, to meet and exchange with other researchers, and to create a wide academic network. I could not have asked for a better environment to start my scientific career.

I am heavily indebted to Dr Paul Burkimsher, who has helped me ever since I first arrived at CERN as a Technical Student. You have been an inspiring supervisor, supportive colleague and dear friend – I learned a lot more than just computer science from you. Thank you for sharing your experience with me.

Marthe deserves special recognition in this list. Thank you for believing in me and encouraging me on a daily basis. You were my biggest supporter, you reassured me when I was in doubt. I am grateful for the many adventures we had on the way.

To my colleagues in the SMV lab, Dimitri, Alban and Damien, who I could not have done it without: Thank you for the fun, help and encouragement. I also thank the former members of the group David, Edmundo, Maximilien and Steve, and all members of the CUI department. You made me feel welcome from the first day.

My supervisor at CERN, Manuel Gonzalez-Berges: Thank you for giving me the opportunity to start my PhD at CERN and learn so many lessons. I am also deeply grateful to my friends and colleagues at CERN. Daniel, James, Josef, Łukasz, Matej, Valentin and all other members of the (former) EN-ICE group: Thank you for keeping me sane. I look back happily to the entertaining lunch and coffee conversations.

I want to thank all my friends in and around Geneva, Austria and across the globe who I cannot possibly all list here. You have provided me with the often-needed opportunities to distract myself from work. Thank you to everybody who joined me for skiing and climbing, played football with or against me, joined BBQs in the park, swims in the lake and concerts. I will treasure these memories forever.

Finally, I would like to express my gratitude to the Hasler Foundation and the Swiss National Science Foundation for enabling me to perform my research.

Summary

The recent advent of cyber-physical systems (CPSs) in end-user applications extends the need for sophisticated model creation, simulation and system verification from classical systems engineering domains to new application areas. Since CPSs such as smart homes and office automation seamlessly integrate technology into everyday life, their safety and correctness become paramount. The intricacy of modelling these systems stems from the merging of two opposing system views: While flows of physical energy and resources are mostly described using mathematical methods such as differential equations, engineered applications are usually best expressed using automata and similar discrete formalisms. Many tools that support such hybrid models lean toward academic use, requiring extensive modelling experience, and neglect usability. Commercial platforms try to mitigate these shortcomings but involve significant financial investment. Additionally, tool creators aim to maximise their products' versatility and application areas, thereby widening the distance between software and target domain. This introduces complexity and configuration effort and increases the risk for errors not directly related to the system itself.

This thesis explores the use of domain-specific languages (DSLs) to bridge the gap between systems and models. It describes the creation of the Continuous REactive SysTems language (CREST), a DSL dedicated to the combined modelling of physical resources and engineered behaviour. The language offers architectural concepts such as hierarchical system composition and typed ports, reactive dataflow aspects that assert a synchronous model behaviour, continuous variable evolution and support for non-deterministic systems. While the language is certainly the main contribution, CREST's design considerations provide additional value to the modelling community. The findings of this project are described according to three research phases.

First, an initial analysis investigates the requirements of CPSs whose behaviour is based on the flow of resources such as heat or electricity and extracts the properties that must be provided by a modelling language or tool. These results are then used to evaluate current modelling software and formalisms.

The second part builds upon these insights to design CREST, a hybrid modelling DSL. CREST reuses well-established concepts from existing formalisms and merges them into a coherent language, whose formal semantics open the door to well-defined execution and simulation. CREST is implemented as `crestdsl`, a Python-based, internal DSL that allows efficient modelling and simulation.

The last research topic describes the application of formal verification on CREST models. This advanced use case is explored from theoretical and practical points of view. Additionally, it has been implemented in `crestdsl` proving its viability. The positive result of the approach highlights the capabilities of CREST, the practicability of the hybrid DSL modelling approach and confirms their effectiveness.

Résumé

L'avènement récent des systèmes cyber-physiques (CPSs) dans les applications proches d'utilisateurs finaux a accentué le besoin d'outils sophistiqués, pour la création, la simulation et la vérification de système dans ces nouveaux domaines. En particulier, la domotique et la bureautique intelligente intègrent de manière transparente diverses technologies modernes d'automatisation dans la vie quotidienne, en faisant de leur sécurité et leur exactitude une priorité. La complexité de la modélisation de tels systèmes provient de la fusion de deux vues opposées. Tandis que les flux d'énergie et les ressources physiques sont principalement décrits à l'aide de méthodes mathématiques telles que les équations différentielles, les applications d'ingénierie sont mieux exprimées à l'aide d'automates et de formalismes discrets. De nombreux outils supportent ce mariage, mais s'adressent à une utilisation académique, nécessitant ainsi une vaste expérience en modélisation, au détriment de la facilité d'utilisation. Il existe certes des plate-formes commerciales qui pallient à ces manques, mais celles-ci induisent généralement des investissements financiers significatifs. De plus, on observe que la plupart des créateurs d'outils mettent l'accent sur des atouts tels que la polyvalence et le nombre de domaines d'applications de leurs produits, élargissant ainsi la distance entre logiciel et domaine ciblé. Ceci a pour effet d'introduire de la complexité et des efforts de configuration, augmentant d'autant plus le risque d'erreurs non relatives au système lui-même.

Cette thèse explore l'utilisation de langages spécifiques à un domaine (DSL) pour combler le fossé entre systèmes et modèles. Elle décrit la création de CREST (Continuous REactive SysTems language), un DSL qui combine à la fois des ressources physiques et des comportements d'ingénierie. Le langage offre des aspects architecturaux tels que la composition de systèmes hiérarchique et de ports typés, des aspects de réactivité sur les flux de données pour assurer un comportement synchrone, ou bien encore l'évolution continue de variables et le support de systèmes non déterministes. La contribution de cette thèse inclut des détails sur le développement de CREST, apportant une valeur non négligeable à la communauté de modélisation et de simulation. Les résultats sont décrits selon trois phases de recherche.

D'abord, une analyse initiale examine les exigences des CPS, dont le comportement est basé sur des flux de ressources tels que l'électricité. L'analyse permet ainsi d'extraire les propriétés qui doivent être fournies par un langage de modélisation. Ces résultats sont ensuite utilisés pour évaluer les logiciels de modélisation actuels.

La deuxième partie poursuit sur ces informations pour la conception de CREST. CREST réutilise des concepts bien établis issus de formalismes existants et les fusionne dans un langage cohérent, dont la sémantique formelle ouvre la porte à une exécution et une simulation bien définie. Celui-ci est implémenté sous la forme de `crestdsl`, un DSL interne basé sur Python.

Un dernier sujet de recherche décrit l'application de la vérification formelle sur les modèles CREST, en l'explorant d'un point de vue théorique et pratique. Les résultats positifs de l'approche mettent en évidence les capacités de CREST, son accessibilité en tant que modèle DSL hybride et démontre la faisabilité de l'approche.

*To speak another language
is to possess another soul.*

*Avoir une autre langue,
c'est posséder une deuxième âme.*

CHARLEMAGNE

Contents

Abstract	vii
Résumé	ix
1 Introduction	1
1.1 Motivation	1
1.2 Approach and Contributions	4
1.2.1 Properties of a Resource Flow Model	4
1.2.2 Evaluation of Existing Languages	5
1.2.3 Creation of a Modelling DSL	6
1.2.4 Simulation	7
1.2.5 Verification	8
1.3 Organisation of the Dissertation	9
2 State of the Art: Systems Modelling	11
2.1 Preliminaries: The Systems Modelling World	11
2.1.1 Viewpoints, Formalisms and Languages	11
2.1.2 The Modelling Universe	12
2.2 Overview of Systems Modelling Concerns	13
2.3 Existing Languages and Formalisms	16
2.3.1 General Purpose and Software Modelling Languages	16
2.3.2 Architecture Description Languages	17
2.3.3 Hardware Description Languages	18
2.3.4 Synchronous Languages	19
2.3.5 Automata	19
2.3.6 Discrete Event System Specification	21
2.3.7 Petri Nets	22
2.3.8 Bond Graphs	23
2.4 Summary	23
3 Resource Flow Modelling – Analysis	25
3.1 Case Study Systems	26
3.1.1 Smart Home	27
3.1.2 Office Automation	29
3.1.3 Automated Gardening	31
3.2 Modelling Criteria	32

3.3	Evaluation of Languages and Formalisms	34
3.3.1	Additional Selection Criteria	35
3.3.2	Language Evaluations	36
3.3.3	Discussion	40
3.4	Summary	40
4	The CREST Language	43
4.1	Syntax	44
4.1.1	Formal Language Structure	49
4.1.2	Global State of a CREST System	56
4.1.3	CREST Syntactic Structure	57
4.1.4	Changes to the System State	57
4.1.5	Semantic Constraints	58
4.2	CREST Semantics	60
4.2.1	Modifiers and Precedence – Formalisation	64
4.2.2	Formal Operational Semantics	68
4.3	Language Extensions	74
4.3.1	Influences	75
4.3.2	Transition Actions	76
4.4	Language Analysis	77
4.4.1	Language Design and Modelling Considerations	78
4.4.2	Zeno Behaviour	79
4.4.3	Modifier Execution Order and Parallel Computation	80
4.4.4	Structural vs. Temporal Non-Determinism	82
4.4.5	Composition Aspects	84
4.4.6	Commonalities with Hybrid Petri Nets	86
4.4.7	Relationship to DEVS	90
4.5	Summary	90
5	CREST Implementation	91
5.1	Overview	92
5.2	crestdsl – CREST’s Python Implementation	94
5.3	Simulation	99
5.3.1	Different Simulators	100
5.3.2	Calculating the Next Behaviour Change Time	102
5.3.3	Limitations	107
5.4	Tool Implementation & Architecture	109
5.4.1	Interactive Visualisation	110
5.4.2	Trace Plotting	112
5.5	Summary	113
6	Verification	115
6.1	TCTL and Timed Kripke Structures	119
6.1.1	Model Checking	122
6.1.2	Applied Model Checking	126
6.2	CREST Model Checking	126

6.2.1	CREST Kripke Construction	128
6.2.2	Ensuring Left-Total Transitions	132
6.2.3	Replacing ε -values	132
6.3	<code>crestdsl</code> Verification	133
6.3.1	Checks	133
6.3.2	Simple API	134
6.3.3	TCTL Model Checking	136
6.3.4	Limitations	138
6.4	Summary	138
7	Conclusion	139
7.1	Summary	139
7.2	Perspectives	142
A	GrowLamp Model – Function Implementations	145
B	CREST Time Base	149
C	Code listings	153
C.1	<code>crestdsl</code> – Listings	153
C.2	Simulation – Listings	155
C.3	ThreeMasses – A Non-linear System	157
D	Acronyms and Symbols	163
	Scientific Work and Publications	169
	Bibliography	173

Chapter 1

Introduction

1.1 Motivation

Cyber-physical systems (CPSs) are combinations of software programs and hardware interfaces such as sensors and actuators. From large-scale industrial applications, such as e.g. automated assembly lines and modern transport applications, to personal systems, such as health appliances and smart homes gadgets, the number of CPS installations is growing continuously. As these systems become ever more intertwined with our life, asserting their correct functionality has become indispensable.

The producers of complex, industrial and safety-critical systems developed model-driven engineering (MDE) approaches [Sch06; BCW12], formal verification techniques [CW96] and rigorous development best-practices [VB04] to prevent system failures and damage to wealth, health and human life. The caveat of these sophisticated but intricate approaches is their high cost in terms of time and money. Thus, in practice, these solutions are mainly employed by financially potent clients and in projects where the cost of failure justifies the elevated investment cost.

Creators of small or non-critical systems, such as home or office automation installations and automated farming applications, often lack the knowledge and resources to use these tools. As a result, less critical systems are often not formally verified or only unsatisfyingly tested. Despite the low risk to health, these applications can still have an enormous impact on individuals' lives. For example, a misconfigured system might experience disturbing power outages when too many devices are started at the same time. Automated plant hydration systems can damage wooden floors and electrical appliances if they spill water, or kill plants and destroy harvests. The goal of this project is to provide the means for system creators in these domains so that they can easily model and verify their CPSs.

The systems that are targeted by this research project have a strong focus on the transfer of physically measurable entities such as light, heat or water. We refer to such systems as *custom assembly systems*, since they are typically compositions of several off-the-shelf components that are connected via loosely documented or proprietary means of communication, such as informal transmission protocols or smartphone application controllers. Components within such CPSs influence one another by producing, consuming and transforming physical resources such as heat or wa-

ter. A lamp for example transforms electricity to light by continuously consuming an electrical current and creating light as output flow. Each component's created resource flow depends primarily on its state but is also affected by inputs from other components and the environment. For example, an automated light system's output depends on whether its lamps are switched on or off (i.e. the local state), the electrical current (which is provided by another component) and the information whether the sun is shining or not (i.e. environmental factors). From an abstract point of view, the CPS can be seen as a network of continuous resources flows that connect system components which act upon these flows by transforming or consuming them.

To prevent faulty system setups and reduce the risk of financial and physical harm, it is important for users to be able to simulate the evolution of their systems when facing changing system influences, and verify that undesired system states will not be reached (e.g. the lamps are on during the day and their power consumption therefore amounts to a high electricity bill). Other examples of verifiable scenarios include the discovery of electrical network overloads, simulation of soil moisture evolution to prevent drought on farmland, and the assertion that plants will receive enough sunlight before the automated sun blinds shut.

Modelling languages that are provided to the builders of such CPSs must be expressive enough to efficiently describe and model these verification scenarios. Naturally, the first research question therefore aims to discover the requirements towards modelling languages and tools so that resource flows in custom assembly CPSs can be modelled and analysed.

Research Question 1. *What are the properties required from a language or tool to model the resource flows and behaviour of custom assembly CPSs?*

In over fifty years of research, the modelling and simulation community has produced various approaches for CPS and real-time and embedded (RTE) systems [ECT03] modelling. Powerful formalisms, languages and tools have been developed, adopted and used to model concerns such as the treatment of time, the synchronism of communication and the composition of components. There is, however, a growing criticism of the complexity of these approaches. Publications such as [Fri09] critique that users require months of training to develop MDE skills, and [Hei98] demanded already over two decades ago that formal methods need to become practicable. In many cases, it is necessary to employ several modelling languages to test individual aspects of the system (e.g. architecture and behaviour) and to create separate models in each one, since they are usually not compatible. The cause for these issues often lies in the wide gap between a generic modelling language or tool and the application domain itself. This poses the question, whether existing solutions are too generic and operate on a too high level of abstraction to be useful.

The criticism is the basis for the next research question. This dissertation will evaluate existing modelling languages and examine whether these solutions offer adequate usability and sufficiently low entry barriers for novice users, provide acceptably high expressiveness for the modelling of practical systems and investigate the semantic gaps between language and model.

Research Question 2. *Are existing CPS languages suitable for the creation of useful models that remain close to their application? Do they constrain the expression of domain-specific features and thereby increase the entry barriers for novice users?*

Experience has shown that most languages and tools suffer from high complexity requirements. The authors of [Whi+13] conclude that many tools do not help application experts, but instead impose a particular way of thinking on the user, which is often very distant from the system domain. Oftentimes, the high temporal effort for the adaptation to the specific domain discourages even expert modellers from applying MDE techniques to small applications. Novice users are even more deterred by the steep learning curves and financial burden of expert tools.

To overcome the criticism of complexity, the modelling and simulation community started pushing towards the use and deployment of domain-specific languages (DSLs) [Völ09]. This trend is carried by the goal to empower domain experts with the necessary means to model, simulate and verify their applications themselves. The target audience of such DSLs is often very narrow, and the development and maintenance costs of standalone DSLs are high [DK08; vDKV00]. The development effort can be reduced, however, by building upon existing languages instead of opting for entirely new creations, as advocated in [Völ+13].

The third research question therefore aims to investigate whether it is possible to either create a new or adapt an existing modelling language to appropriately model the flows of physical resources within CPSs.

Research Question 3. *How can we create or adapt a language to fill the need of modelling domain-specific aspects such as resource flows in CPSs? How can existing language features and implementations be reused to lower development and maintenance efforts?*

Evidently, the possibility to model the domain-specific system aspects has many benefits, such as a focused system view due to the abstraction of irrelevant information, facilitated reasoning due to the modelling of influences and effects, and the improved understanding of the system. Especially in dynamic resource flow systems, however, it is of importance to execute the models and assess their evolution. Simulation is a valuable means to answer “*What happens if...*”-questions. It permits the exploration of system evolution and analysis of individual execution scenarios.

For such a simulation it is necessary to provide a well-defined model behaviour, for example in the form of a formal semantics. Commonly, such a formalisation is specified on the language level, so that all models written in that language can benefit. Therefore, the next research question aims to provide a formal syntax and semantics for the developed DSL.

Research Question 4. *How should the formal syntax and semantics of the DSL be characterised, so that the created CPS models are well-defined and can be simulated and analysed?*

In addition to the evaluation of one particular execution scenario, it is usually of high interest to verify specific properties in all possible system evolutions. Most systems expose for example behaviour that should either always or never occur. The plant growing system, for example, should always assert that the soil is humid enough for the plants and prevent dryness damage. On the other hand, it should never occur that the plants are brightly lit for more than 18 hours per day. The analysis of system models with the goal of asserting such properties is commonly performed through formal verification [DKW08]. These techniques operate on a model's formally defined syntax and semantics and analyse all possible system evolution scenarios to prove that a favourable system behaviour is attainable, or that unwanted settings can never occur. The last research question is thus:

Research Question 5. *How can we use formal verification approaches to verify system behaviour and which techniques can be used for verification of our DSL's CPS models?*

1.2 Approach and Contributions

The following section elaborates on the research approach I followed throughout the project, the results that I obtained, and the contributions made thereby.

1.2.1 Properties of a Resource Flow Model

The main purpose of this research is to enable a system's domain users (as opposed to modelling and simulation experts) with the necessary means to model, simulate and verify custom CPSs assemblies such as home automation systems. Thus, it is necessary to evaluate the systems that should be modelled, the aspects that must be represented and the properties that the modelling formalism should have. I therefore performed an evaluation on three case study systems. The first one is a smart home application that models a water boiler, a hot water shower and Internet of Things (IoT) appliances (e.g. an autonomous vacuum robot, "smart" TV, dishwasher). During the day, the system uses solar panels and a battery to generate electricity. Next, an office system with automated presence detection for energy-efficient regulation of light and temperature was developed. Lastly, an automated indoor gardening system that uses light, temperature and soil moisture readings to automatically control growing lamps and watering systems was modelled.

These systems were analysed from behavioural and architectural viewpoints. The focus lies on the representation of physical resource flows between components to assert availability/absence of physical resources (e.g. light, water, noise, electricity).

The analysis of these systems led to the discovery of six key aspects which should be supported by a CPS language. I further discovered four more properties for the evaluation of existing languages. The reason behind separating these is to help create an ordering in case several, equally good candidates are found. Thus, the usability of a language was evaluated from a domain expert's point of view, and the evaluation

of suitability was performed with resource-intensive CPSs in mind. My contributions towards Research Question 1 are summarised as follows:

Contribution 1. I analysed three custom CPSs and extracted their common property needs towards modelling languages.

C 1.1 (Analysis). I designed and analysed typical CPS case studies which can be used as references for future research. I confirmed that resource flows are essential parts of these systems [KLB18a; KLB17; KLB18b; KCB18].

C 1.2 (Key property extraction). Based on C 1.1, I extracted six key properties and four additional language features that should be supported by a modelling language to model the resource flows in CPSs [KLB18a; KLB18b].

1.2.2 Evaluation of Existing Languages

Based on the findings of the case studies, I analysed various existing languages for their suitability. The list of key aspects (C 1.2) served as a reference guide for the search of a good candidate. Next to the comparison of these properties, additional language features such as expressiveness and availability of formal semantics were examined. Further, an analysis of non-functional aspects such as simplicity, usability and target domain suitability (i.e. the complexity of expression of domain concepts) was added. The aim of the evaluation of the latter was to perform the analysis of non-functional properties from the viewpoint of non-expert modellers.

I chose twelve languages for this evaluation. The selection consists of actively used and representative solutions for modelling and simulation of CPSs, comprising different modelling paradigms. For groups of similar languages (e.g. Unified Modeling Language (UML) [UML17] and Systems Modeling Language (SysML) [SysML17]) only one representative was chosen. The evaluation provided valuable insights and can be summarised as follows: All evaluated candidates allow modelling of reactive behaviour. Almost all of them also support locality of information and provide some form component composition. When comparing other features, several drawbacks were found. Certain modelling languages do not support continuous behaviour, lack the expression of parallelism, non-determinism or synchronous means of communication. The most promising hybrid system tools lack formal semantics for verification purposes or have deficits in terms of usability. Even though some languages support all required key aspects, their complexity renders them far from simple or beginner-friendly. Further, only few languages are equipped with support of resource flow concepts (e.g. definition of resource types), or allow extension to add them.

The conclusion of the analysis is that current languages and tools are complex to use and require significant adaptation to our use cases. Formally, my contribution to Research Question 2 is:

Contribution 2 (Language Evaluation). I evaluated existing modelling languages based on the properties from C 1.2 and their non-functional aspects [KLB18a].

1.2.3 Creation of a Modelling DSL

The evaluation of features showed that existing languages and tools are not fully suitable for modelling resource flows in CPSs, especially when considering domain users as a principal audience. Therefore, the goal was set to create a DSL to fill this lack. The comparison of different languages provided a valuable overview of other languages' design choices, that could be harnessed as references.

This led to the definition of my language's requirements prior to its actual development. First, the DSL requires a graphical syntax to allow visual representation and analysis by users. Additionally, there must be a textual syntax to speed up the development process for experienced users and an interface to a general-purpose programming language (GPPL) for the creation of complex behaviours by power users.

Based on these requirements, I created the Continuous REactive SysTems language (CREST), a DSL for the modelling of resource flows of continuous-time CPSs. The language focuses on the definition of architecture and behaviour. The component behaviour is specified using finite-state machines. Within each state, *update* functions are responsible for the continuous evolution of the system. Components are modelled as *entities* with well-defined interfaces (*ports*). System composition is expressed using a strictly hierarchical entity structure, and the resource transfers are modelled using *influence* functions that connect the entities' ports. Figure 1.1 shows a CREST entity with two states, one input and one output port and update functions that modify the output port.

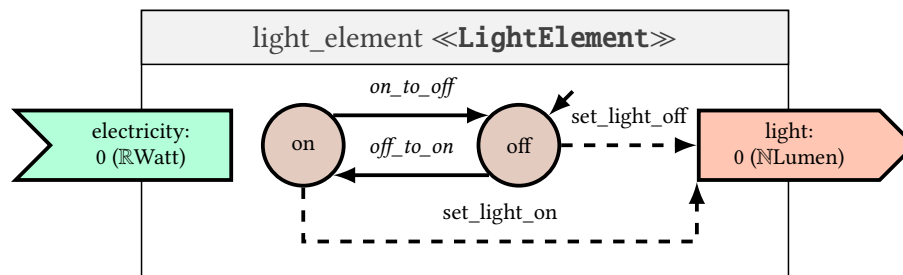


Figure 1.1 – A light element entity in CREST's graphical syntax. The figure shows states, input and output ports and update functions that modify the output.

The language is, similar to SystemC, developed as an *internal* DSL [Völ+13], using the Python GPPL as a host language. Entities are modelled as classes and instantiated into standard Python objects. Similarly, classes for resources, ports and states are provided in a library and can be used after importing it. Update functions and influence behaviour are expressed using annotated class methods in standard Python syntax. Listing 1.1 shows a brief example of a CREST entity class.

The choice of Python as a host language is founded in its simplicity, flexibility and supportive community. The use of this wide-spread programming language allows the uncomplicated extension through numerous community-provided libraries. It also permits the use of existing infrastructure such as code editors, source analysis systems and unit test tools without the need for separate developments. The contributions towards Research Question 3 are as follows:

Contribution 3. I created a graphical DSL for the specification of resource flows within a CPSs. The language and simulator are implemented in Python for the reuse of its syntax and ease of extension.

C 3.1 (DSL). I developed CREST, a graphical language that allows the expression of resource flows and component behaviour. [KLB17; KLB18b; KLB18a]

C 3.2 (Textual Syntax). I implemented CREST in the form of an *internal* DSL using Python as a host language [KLB17; KLB18a].

Listing 1.1 – Example of an entity modelled in `crestdsl`. Language elements are expressed as standard Python classes, objects and annotations.

```

1  import crestdsl.model as crest
2  class LightElement(crest.Entity):
3      # port definitions with resources and an initial value
4      electricity = crest.Input(resource=electricity, value=0)
5      light = crest.Output(resource=light, value=0)
6
7      # automaton states - one is the current (initial) state
8      on = crest.State()
9      off = current = crest.State()
10
11     # transitions and guards (as lambdas)
12     off_to_on = crest.Transition(source=off, target=on,
13                                 guard=(lambda self: self.electricity.value >= 100))
14     on_to_off = crest.Transition(source=on, target=off,
15                                 guard=(lambda self: self.electricity.value < 100))
16
17     # updates are specified using decorators
18     @crest.update(state=on, target=light)
19     def set_light_on(self, dt=0):
20         return 800
21
22     @crest.update(state=off, target=light)
23     def set_light_off(self, dt=0):
24         return 0

```

1.2.4 Simulation

Both CREST's structure and its semantics are formally specified, so that the simulation is well-defined and `crestdsl`'s simulator can be used for the study of system models with passing of time. It uses an approach that calculates the exact time at which the next state transition will occur. This strategy avoids unnecessary calculations for points in time where the system is not changing its behaviour. My approach avoids classical, step-based simulation, where guard functions and system updates are only evaluated in predefined intervals. It removes the risk of missing the correct transition time, the need for iterative recalculation, backtracking and trial-and-error step sizes. The `crestdsl` simulator's calculation of the *next transition time* (and hence step size) is based on the analysis of transition guards, updates and influence func-

tions and the subsequent expression thereof as constraint sets. By using a satisfiability modulo theories (SMT) solver (e.g. the Z3 theorem prover [DB08]) the constraints can be evaluated and the minimal time in which a transition will become enabled is found. This solution provides the precise next transition time, without the need for iterative search. The contributions towards Research Question 4 are thus:

Contribution 4. I provided a formalisation of CREST and used it for the simulation of CREST models. The module is also implemented in `crestdsl` and can therefore be used and extended easily.

C 4.1 (Formalisation). I defined the language’s syntax and semantics formally, so that CREST models can be formally analysed and the language supports well-defined simulation [KLB18b].

C 4.2 (Simulation). I developed a simulator for `crestdsl`, so that it can be used to study the evolution of CPS models. The simulator itself uses predictions of transition times to avoid fixed-step-size exploration [KLB17; KLB18a; KCB18].

1.2.5 Verification

The verification of CREST systems requires careful analysis of its temporal aspect. Due to their continuous time nature, CREST systems can advance in arbitrarily small timesteps. Despite this fact, CREST’s semantics enable the verification of timed system properties. These are translated to timed computation tree logic (TCTL) [AH92] formulas that express at what point in time the property should hold. Next, the possible system evolutions are encoded in directed graphs called timed Kripke structures [LÄÖ15]. These graphs represent the state of systems (e.g. CREST models) as graph nodes and use time-annotated transitions to model the amount of time that needs to pass before reaching the next state. Similar to the simulation, the creation of the graph structures also uses the next transition time function. Since each graph node is labelled with the set of system properties that hold within its state, the verification of TCTL formulas on timed Kripke structures is equivalent to verifying graph properties, such as the existence of paths between two nodes with a certain label. TCTL and timed Kripke structures have been shown to be useful for the verification of timed systems such as hybrid and timed automata. The thesis introduces an approach to create CREST Kripke structures, which are timed Kripke structures that are favourable for the verification of CREST models. For example, CREST Kripke structures use the next transition time function to assert that each graph node identifies a single system state.

Based on this Kripke structure, I show the use of a model checking algorithm that allows the evaluation of TCTL formulas and provides an efficient verification solution. My contributions towards Research Question 5 are listed below:

Contribution 5 (Verification). I developed and implemented a formal verification technique for CREST models.

C 5.1 (Formal Approach). I formalised a model checking approach for the verification of CREST systems. The solution is based on CREST’s formal semantics, uses adapted timed Kripke structures for the representation of the system’s state space and TCTL for the specification of system properties.

C 5.2 (Implementation). I implemented the verification approach in `crestdsl`. The implementation provides a simple application programming interface (API) for unfamiliar users and classical TCTL operators for expert users. Both APIs build upon the exploration of state spaces encoded as timed Kripke structures.

1.3 Organisation of the Dissertation

The rest of this thesis is structured as follows: Chapter 2 reviews the state of the art in the domain of CPS modelling and analyses several modelling concerns that have to be addressed. Subsequently, a discussion of different CPS modelling formalisms and languages provides a general overview over current approaches. Building upon this outline, Chapter 3 defines three example systems that are representative of the CPSs that are targeted by my research. The systems are analysed to extract a set of core features that need to be provided by a modelling language that is appropriate for our target domain and users. Chapter 3 ends with an extended analysis of the suitability of existing languages. Chapter 4 provides details of the development of CREST, a novel language for the modelling and analysis of resource-intensive CPSs, such as smart homes, office automation or plant growing systems. The chapter introduces the language’s graphical syntax using a concrete example and defines CREST’s formal syntax and semantics. Chapter 5 showcases `crestdsl`, CREST’s implementation as internal DSL, which uses the Python programming language as host. The chapter focusses on `crestdsl`’s syntax and its SMT solver-based simulation. Finally, CREST’s interactive development and simulation environment is presented. Chapter 6 provides details of the formal verification solution that is based on CREST’s semantics. It describes the use of TCTL and timed Kripke structures for the verification of CREST systems using model checking techniques, as well as their concrete implementation in `crestdsl`. Finally, Chapter 7 summarises this dissertation’s findings and provides an outlook on possible future developments.

Research Questions and Thesis Chapters Table 1.1 shows a mapping between the research questions and the dissertation chapters in which they are addressed. Note that Chapter 4 and Chapter 5 both describe the CREST language, including its formalisation and the correct simulation. The theoretic part of these subjects provides the formal foundation of the language in Chapter 4, while Chapter 5 is concerned with the language’s implementation.

Table 1.1 – Research questions and the chapters that address them

		Dissertation Chapter				
		Ch. 2	Ch. 3	Ch. 4	Ch. 5	Ch. 6
RQ 1	<i>CPS Requirements</i>		•			
RQ 2	<i>Language Evaluation</i>	•	•			
RQ 3	<i>CPS DSL</i>			•	•	
RQ 4	<i>Simulation</i>				•	
RQ 5	<i>Verification</i>					•

Chapter 2

State of the Art: Systems Modelling

In general, “systems modelling” refers to the creation of a representative *model* of a system. This model is an abstraction that can be used instead of the original system to analyse and study the latter. Abstraction refers to the simplification or disregarding of some system aspects to facilitate its analysis. Next to abstraction, models require four other key characteristics to be useful. Models must be understandable, accurate (i.e. represent the system faithfully), predictive (i.e. conclusions can be drawn based on the model) and inexpensive (i.e. significantly cheaper to create and analyse) [Sel03].

Throughout the years, many types of models have been used and introduced to serve different tasks. For example, some models can be representations of physical systems, using differential equations to describe physical phenomena. Other models hold information about the logical architecture of computer systems or facilitate the analysis of business processes. Numerous modelling languages, formalisms and aspects have been developed to support the creation and analysis of such models, each dedicated to aid a certain type of task. This chapter first reviews some basic definitions in Section 2.1 and then provides a brief overview of the systems modelling domain in Section 2.2. The section provides a more thorough introduction of different types of models and modelling approaches. The focus of this chapter, however, is put on analysing and comparing existing modelling solutions for CPSs. Section 2.3 descends deeper into the topic and introduces noteworthy languages and formalisms.

2.1 Preliminaries: The Systems Modelling World

Before reviewing the current state of system engineering, it is necessary to introduce some vocabulary that is used in the world of CPS modelling and systems engineering.

2.1.1 Viewpoints, Formalisms and Languages

This dissertation follows the terminology provided in the Object Management Group (OMG)’s model-driven architecture (MDA) guide [MDA14]. This means that models are created to serve the stakeholders’ *viewpoints* and are implemented in a language with a given syntax and semantics.

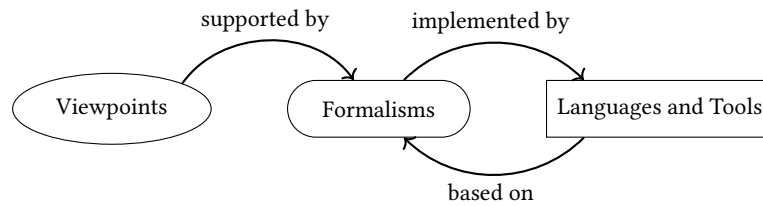


Figure 2.1 – Framework for the relation of Viewpoints, Formalisms, Languages and Tools. Reprint from [Bro+12].

Broman et al. [Bro+12] provide a more thorough structure, where they distinguish between *viewpoints*, *formalisms*, and *languages and tools*. In their framework, they additionally define that viewpoints are supported by formalisms, which are mathematical constructs and baselines, expressed through a formal syntax and semantics. These formalisms allow the description and analysis of systems. To facilitate their use, languages and tools implement these formalisms to enable the interaction with models, as shown in Figure 2.1 (originally published in [Bro+12]).

Note that these relations are not exclusive and there can be many formalisms supporting a viewpoint. Similarly, a language can implement different formalisms and thereby express many viewpoints. The final choice of an appropriate language and formalism depends on the individual stakeholders and their specific system interest.

This thesis adheres to the classification presented above. For simplicity, however, formalisms and languages are treated in the same manner, since their differences are often subtle, where formalisms operate on an abstract rather than a concrete syntax. Furthermore, many formalisms (e.g. automata and Petri nets (PNs)) developed de-facto standard notations, thereby slightly blurring the line between abstract formalism and concrete language. Nonetheless, this thesis acknowledges their difference and uses the appropriate terminology where needed.

2.1.2 The Modelling Universe

In the systems engineering domain, models appear in various forms throughout the lifetime of a system. Depending on the specific task, models can be used to validate specifications before the actual creation of the system, verify the system’s correctness during its implementation or even support the analysis of system properties after the system construction has been completed. Depending on the extent of the model’s involvement in these phases, the terms model-driven architecture (MDA), model-driven development (MDD), model-driven engineering (MDE) or model-based engineering (MBE) are used to refer to the different employment methods of models in systems engineering. The difference between the terms is often subtle and usually depends on the user’s point of view – in some literature the terms are even used interchangeably. This thesis follows the classification provided in [BCW12]. Using this framework, MBE is the most generic form of model use. It refers to using a model in the engineering process, although it might not be the driving force of the process. MDE is a subset of MBE, where models are the main artefact of the engineering process. In situations where MDE is used purely for the development of a system, rather

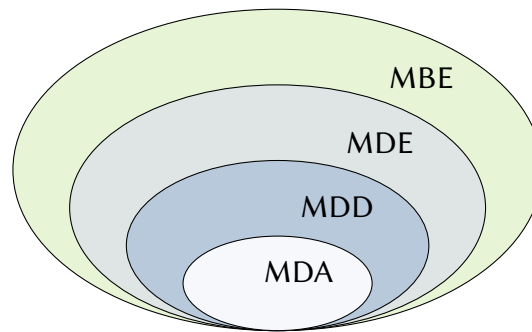


Figure 2.2 – The MD* Jungle of Acronyms. Reprint with permission from [BCW12].

than generic analysis or study, one might use the term MDD. MDA is the OMG’s specific view of this approach. Figure 2.2 depicts the relation between the four concepts.

Over the years, various publications and surveys evaluated the practicality, benefits and disadvantages of MDA, MDD, MDE and MBE. Starting points for a more profound exploration of this subject can be found in dedicated publications such as [Val15; Sel03; Est08; HWR14; Sta06; Rod15].

2.2 Overview of Systems Modelling Concerns

The success of a modelling task is heavily influenced by the choice of model type, modelling language and modelling concerns that should be represented. Sargent identifies four basic types of models: iconic, graphical, analogue and mathematical [Sar15]. Of these four types, the iconic approach (i.e. the creation of a representative miniature), is rarely used for modern, computer-driven CPS tasks. The use of analogue models, where values are represented through measurable physical phenomena (e.g. electrical voltage or a water integrator [Ken12]), became outdated with the rise of digital computers. Graphical (i.e. graph based) models are used to represent the relations between various system concepts and usually serve as high-level abstractions, which allow analysis through well-established algorithms and transformation of the models into other representation forms (e.g. code). The arguably most important model type in the CPS domain are mathematical models, where behaviour is expressed using a set of mathematical equations, e.g. as ordinary differential equations (ODEs). In most cases, specialised software is then used to execute such models and trace the behaviour by observing measurement variables.

The digital modelling of analogue behaviour is deeply intertwined with the underlying computation system. Finding precise solutions to complex equation systems is non-trivial and oftentimes relies on various numerical ODE solving techniques [But08]. There exist many other CPS modelling challenges that also have to be addressed. Examples include reliability issues of computing and communication architectures, distributed computation and concurrency concerns, and challenges of time representation. See [DLS12] and [Lee08] for a more profound discussion of additional current CPS modelling problems and potential solutions.

Due to the vastness of the CPS modelling field, it is difficult to provide a complete classification of modelling approaches. Thus, this chapter's focus is put on exploring CPS modelling concerns and the languages that allow the expression of certain stakeholder viewpoints. For a complete introduction to modelling and simulation, the reader is referred to more exhaustive reviews of the domain, such as [Nan94], [BA07] and [ZKP00]. A comparison of different modelling approaches for embedded systems is provided in [ECT03].

Note that the choice of modelling concerns for any application highly depends on the involved stakeholders, their interests in the system and the system itself [Hil99]. The rest of this section will therefore be restricted to a subset of important aspects in the domain of CPSs modelling.

Architecture Architectural concerns refer to the logical setup of a CPS. The term architecture can refer to either the *software*, i.e. the grouping of software code and packages into logical units, to *hardware*, where it describes the separation and grouping of physical components on a logical level, or to the combination of both, such as the assignment of computing tasks to available hardware. Formalisms and languages that address this concern usually also allow the modelling of system composition to larger systems-of-systems and definition of communication interfaces (although not the communication itself).

Synchronism An important aspect of the system architecture is the modelling of communication between individual parts of the system. Communication, i.e. the passing of information from one part of the system to another, can be modelled as either synchronous or asynchronous [CMT96]. For synchronous communication, sender and receiver maintain the same communication pace and the sender waits with further computation until the message has been received. Hence, it is possible to establish efficient, well-coordinated communication between components. In asynchronous communication systems, on the other hand, there is no guarantee at what point messages are sent and received. This makes systems more flexible as the individual system parts are more independent and can operate "at their own pace". Nonetheless, certain systems (e.g. physical systems) have components that need to treat messages/signals whenever they arrive. Such components/systems are often-times reactive, which means that they wait for certain messages to perform computation, rather than performing their activity in a periodic manner. The choice of communication strategy heavily influences the expressiveness of the language and thus the kind of models that can be created. This choice also has severe impacts on the system itself if the models are used as basis for the system development, such as in MDD or MDE approaches [Cri96].

Time Many models and formalisms disregard the time aspect in general, as it is not important for their purpose. For such applications the order of actions is of higher importance than the actual timing and interval information itself. When thinking of an automated transport system for instance, it is important to verify that a certain order of actions cannot occur (e.g. that a train cannot leave before its doors were shut).

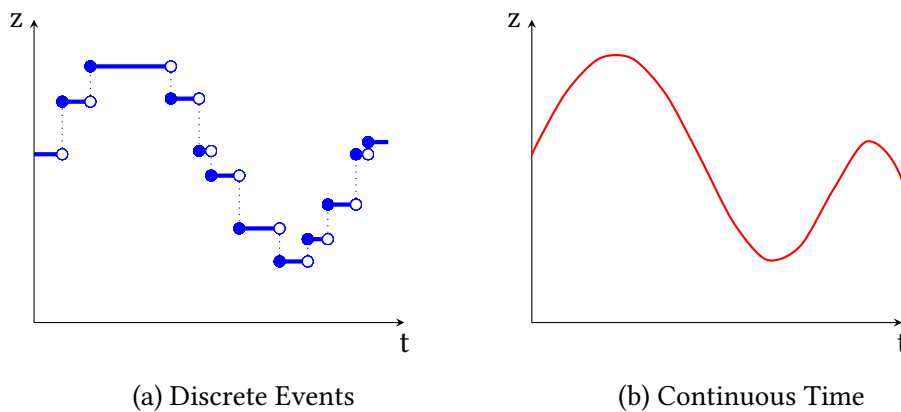


Figure 2.3 – Two plots showing observations of the function $\sin(x) - \sin(\frac{x^2}{4}) + 1$. Sub-figure (a) shows the step-function that created by sporadically measuring at discrete points in time, (b) shows its continuous evolution.

The delay between the “doors shut” signal and the train leaving is not of interest, as long as the one always precedes the other.

On the other hand, some models put high emphasis on a system’s temporal aspect. Such models deal with time and its effects on a system. Especially for safety-critical applications, time is an essential factor in the decision whether a system behaves correctly or not. Airbags need to inflate within a certain maximum time span, train crossing controllers must not open the barriers before a safety interval passes and traffic light phases need to adhere to a predefined schedule.

Time itself can be introduced in various forms, such as discrete or continuous (see [Lop15; Zei76] for exhaustive discussions). Discrete time evolution is modelled by observing discrete events and associating each with a timestamp (e.g. an integer or real value) and creating a chronological order of these events [GG15]. The passing of time is the iterative reaction to these events in the defined order. Typically, the time between events is disregarded, as it has no importance to the model and simulation. The resulting value observations can be visually represented as step functions, as shown in Figure 2.3a.

Continuous time [Reg15], on the other hand, describes continuously evolving clocks and variables whose values steadily increase according to a pre-defined specification (e.g. a set of ODEs), as shown in Figure 2.3b. Even though computation is more complex, for some system models the use of continuous time is indispensable.

Determinism The behaviour of models can be expressed in various forms using formalisms such as finite state machines (FSMs), decision trees or as rule-based systems [HW15]. However, arguably one of the most important properties is the distinction between deterministic and non-deterministic models. Determinism refers to the fact that, given it is in the same state and receiving the same input, a model will always produce the same output. Contrary to this view, non-deterministic models can expose different behaviour at each execution. In its simplest form, a non-deterministic model will randomly execute an action from a set of options (e.g. en-

abled transitions in an FSM). This means the probability of each choice is $p = \frac{1}{n}$, where n is the number of choices. More complex models can include probability theory in form of stochastic behaviour descriptions, Markov chains and similar.

Causality In most situations, the influences between individual model components are known. The model itself expresses a source’s impact onto a target, or in mathematical terms: the change of one variable causes another one to change, too.

However, sometimes it might be of greater interest to express the relation between variables. This is especially useful in situations where it is not clear which values will be known at runtime. Such a definition is referred to as *acausal*.

As an example, we might look at the calculation of the electrical resistor equation. A causal model might define that $R := \frac{v}{i}$ (where “:=” is the variable assignment). In traditional GPPLs this statement means, that the resistance value R is always calculated based on the voltage v and the current i . In modelling languages that allow acausal definitions, this statement can be used to determine the value of any variable, provided the other two are known. Hence, the language will implicitly calculate one of the following formulas, depending on which variables’ values are known: ($R = \frac{v}{i}$; $v = R \cdot i$; $i = \frac{v}{R}$). The use of acausal modelling increases the reusability of models [BF08] but also leads to unwanted and difficult to debug model behaviour.

2.3 Existing Languages and Formalisms

The rest of this chapter outlines some existing modelling approaches in the CPS field. Due to its extent, however, a complete discussion cannot be provided. A more exhaustive list of CPS formalisms, languages and tools is provided for instance in the COST Action report on “Multi-Paradigm Modelling for Cyber-Physical Systems” [Kli+19].

2.3.1 General Purpose and Software Modelling Languages

Since the late 1990s the software engineering community has thoroughly embraced the OMG’s UML [UML17] as one of the de-facto standard modelling languages for systems engineering. Its flexibility, combined with publications about the language’s benefits (e.g. [And+06]), led to UML’s high popularity¹ and replacement of other languages, such as the Integrated DEFinition (IDEF) Method family [IDEF18] and Specification and Description Language (SDL) (see below). UML also allows more flexibility in the data modelling than the simpler entity-relationship (ER) diagram modelling language [Che76]. Nowadays, UML is omni-present and included in virtually every computer science course syllabus. Its versatility manifests itself by offering over a dozen different diagram types to express various views and allowing the description of a system’s structure (e.g. class and package diagrams), behaviour (e.g. activity and state diagrams) and interactions (e.g. sequence and communication diagrams).

The Specification and Description Language (SDL) [SDL16], a predecessor of UML, provides many features for the modelling of systems. It persuades by offering a

¹To an extent, where it has even been called a disease [Bel04].

clear, hierarchical entity composition using *agents*, sequential behaviour definitions and data descriptions [Fis+00]. The language creates reactive system models whose agents perform computation upon receipt of asynchronous signals. Lastly, SDL's rigorous formal basis [Esc+01] is a compelling advantage that allows formal verification (e.g. [Vla+05; MIJ14; SS01]) and tool-independent simulation.

One often-criticised weak point of SDL and UML – next to the latter's lack of a formal semantics – is their missing support for real-time and embedded (RTE) concepts such as timing and performance. SysML [SysML17] is a language that adapts and extends a subset of UML to include embedded systems modelling capabilities. The OMG's UML Profile for Schedulability, Performance, and Time Specification (SPT) [SPTP05] is another extension of standard UML that was developed to overcome the lack of RTE concepts by providing standardised diagram annotations and quantitative analysis techniques. However, both SysML and SPT lack flexibility and require improvements [Dem+08].

The UML Profile for Modeling and Analysis of Real Time and Embedded systems (MARTE) [MARTE11] is a more recent and extensive approach to add fundamental RTE modelling capabilities, such as expression of non-functional properties and execution platform specification, to the language. It is customisable, flexible, shows promising successes [Iqb+12; Vid+09; Aul+13] and continues to follow its destiny as a replacement for SPT [Obj18]. Experience also shows that combining UML, MARTE and SysML allows merging their benefits [SSB14], provided that the semantic gap between the languages can be bridged [Iqb+12].

By taking RTE concerns such as performance, timing and architecture into account, MARTE enters the realm of architecture description languages (ADLs).

2.3.2 Architecture Description Languages

ADLs are languages that primarily focus on the structural aspect of systems. Depending on the system under study, different ADLs can be considered, each specialised to a certain type of system. An early classification and survey of ADLs is given in [Cle96]. Well-known ADLs include the Embedded Architecture Description Language (EADL) [Li+10] and EAST-ADL [EAST13]. In the CPS and RTE domain, the most widely used ADL is the Architecture Analysis & Design Language (AADL) [AADL17; FGH06], which has been successfully used for various projects (e.g. [Per+12]). It supports both textual system descriptions and an intuitive graphical syntax. It is also worth mentioning the π -ADL [Oqu04], which is based on the π -calculus and can be used for modelling of mobile and dynamic systems.

One often-criticised shortcoming of ADLs is their lack of behavioural specification. To overcome this issue, some ADLs are extended to include such behaviour descriptions. AADL's Behavioural Annex [Fra+07b], for example, enriches the formalism with hybrid automaton (HA) capabilities (see below), which are based on the language's partially formalised execution model. A similar approach is used by MontiArcAutomaton [RRW15], which extends the capabilities of the MontiArc [HRR14] ADL using finite state automata (FSAs). Both AADL and MontiArcAutomaton have shown good initial results for RTE systems modelling [Fra+07a; RRW13; RRW14].

Despite the benefits of employing ADLs in MDD engineering, the languages have only seen moderate industrial interest. This is attributed to the low number of supporting tools and restricted modelling views [WH05]. A different explanation can be found in the fact that ADLs in general and AADL specifically, can be (at least partially) replaced by UML [Pan10] and MARTE [Fau+07].

2.3.3 Hardware Description Languages

Hardware description languages (HDLs) have been extensively employed for the modelling and simulation of embedded systems, low-level hardware and system-on-a-chip designs since the early 1970s. VHDL [VHDL11; Ash08] and Verilog [Ver06; TM96] are two of the most common representatives. Aptly named, HDLs target transaction-level modeling (TLM) and register-transfer level (RTL) designs, which can be used in the verification and validation of hardware. HDLs provide built-in support for various embedded systems concepts, such as mutexes, semaphores and four-valued logic, and measure time in sub-second granularity, e.g. in picoseconds.

Nowadays, SystemC [SysC12; Bla+10], an IEEE standardised language, is a popular addition to the HDL family. Contrary to the former two languages, SystemC is not a complete language by itself but rather a set of C++ classes and macros, that allow the representation of HDL-concepts. Indeed, the use of a GPPL as foundation provides SystemC with out-of-the-box flexibility, adaptability and extensibility. SystemC programs are written as regular C++ code, the according HDL concepts are created using classes and functions provided by the library. Models are composed of modules which specify ports, communication signals and channels. Behaviour is modelled inside modules as functions, which execute at predefined events and threads.

The language's advantage is that existing integrated development environments (IDEs), compilers and tooling (e.g. for testing and analysis) can be used. This increases convenience for developers and allows the use of agile, tool-enabled development styles. Furthermore, complete beginners can first focus on learning the C++ language by using a plethora of available resource, before becoming familiar with the specific modelling concepts. Users who are already proficient programmers can skip this step and go straight to acquiring the required modelling skills.

SystemC's pragmatic approach as *internal* DSL is reflected in the absence of a formal semantics. Several proposals have been made, however, (see e.g. [Sal03; Mue+01]), and there exist approaches to formally verify (subsets of) SystemC [HG15].

One disadvantage of HDLs is their lack of continuous time concepts and analogue modelling capabilities. This issue has been addressed by providing AMS (analogue and mixed-signal) extensions to the respective languages [IEEE99; Ver14; SysC16]. However, despite these extensions, the languages' focus, as well as most tooling and verification support is geared towards the generation and verification of TLM and RTL designs. This is too low-level for larger CPS that focus on system composition.

2.3.4 Synchronous Languages

In many situations CPSs need to react to changes in their environment. Systems that perform actions based on input changes are commonly referred to as *reactive* [HP85]. The Statechart formalism [Har87] is a prominent representative of such languages. It allows the modelling of system behaviour using a graph syntax (i.e. as state automata), where each node represents a system state and directed arcs symbolise state changes. Statechart's simple, yet highly expressive syntax and semantics increased the formalism's popularity and adapted forms have therefore been included in various other languages, e.g. to UML as *state diagrams*.

Synchronous languages [Hal98] such as SIGNAL [BGJ91], Lustre [Hal+91] and Esterel [BG92] are prominently used to create and verify executable models of reactive systems. These languages are based upon a *synchrony hypothesis*, i.e. the assumption that computation can be performed infinitely fast and hence executed in zero time. Practically, the computation only needs to be finished before the next inputs arrive. Synchronous languages perform computations according to recurring logical signals (*clocks*). A base clock provides the source interval rate, other computations are based thereon. The three languages are distinguished by the fact that Esterel is an imperative language, while SIGNAL and Lustre are declarative. This means that Esterel provides calculation blocks that are executed according to the clock status. The latter two declarative languages express relations between variables in the form of simple equations. A program is valid, if there is a non-deterministic assignment that does not contradict any relation between calculation blocks. The difference between the latter two languages is that Lustre operates on sequences of inputs and outputs, while SIGNAL focuses on relating inputs and outputs of its operators, thereby constraining the program until it is deterministic. A more thorough differentiation of these languages, including an introduction of each one is provided in [Hal93].

Synchronous languages have been used extensively for the verification of RTE systems [Ray10; BKS03], but lack one important feature: the capability to model continuous time. To analyse steady system evolutions, e.g. described by ODEs, it is necessary to introduce the concept of continuous time advances. To overcome this issue, several attempts have been made to combine the advantages of continuous tools and languages with the synchronous system view [LZ07; Ben+11]. Recently Zélus [BP13], which builds upon the advances of Lustre and Esterel, was developed as a standalone solution to overcome the limitations and support continuous model behaviour.

2.3.5 Automata

The domain of automata (a.k.a. *abstract machines*) is concerned with the study of problems using system abstractions that are expressed as directed graphs. In automata theory the graph's nodes are referred to as *states*, edges as *transitions*. Classical automata types include FSMs [Gil62], pushdown automata [ABB97] and Turing machines which are capable of recognising and accepting regular, context-free and recursively enumerable languages, respectively.

The use of discrete transitions and variable modification at transition times allows automata to express complex system behaviour using a high level of abstrac-

tion. The analysis of automaton evolution is often performed by using a technique called state space exploration. Thereby, the states of an automaton are iteratively evaluated for enabled transitions, that lead to new global configurations of the automaton. Depending on the specific automaton, it is possible that its state space is very large or even infinite. The set of system states in a state space, connected by transitions, can be seen as a directed graph. For the analysis of the state space for reachability (or non-reachability) of certain states, the description of recurring patterns and similar analyses fall into the domain of temporal logic specifications, such as the computation tree logic (CTL) [CE81] and linear temporal logic (LTL) [Pnu77]. These formalisms allow the expression of properties using formulas, whose validity can be verified or disproven for state spaces using e.g. *model checking* [CGP99].

Though, the use of discrete transitions and variables leads to efficient property analyses and the reduction in the number of possible system states, it does not allow the modelling of continuous time or continuous variable evolution within a system. Timed automata (TAs) [AD94; BY03] are a method to add such continuous evolution to the automata formalism. This adaptation uses so-called *clock*-variables, which continuously grow their value. To control the automaton's evolution, transitions can be guarded by constraints (i.e. Boolean expressions) over clocks and it is also possible to reset clock values to zero upon transition triggering.

TA have also been extended to add more capabilities to the clock interactions, such as stopwatch automata [CL00] (clock advances can be paused), interrupt automata [BH09] (different interrupt levels and only one active clock per level) and hourglass automata [Osa+14] (clocks have maximum values and can run backwards). There are also techniques describing TA with independently evolving clocks [Aks+08]. Modifications of the formalism also influence its language properties. While reachability is decidable for classic TA [Hen+98], for many extensions it is not.

Most of these extensions modify the expressive power of TA. In general, these kinds of systems are commonly referred to as hybrid automata (HA) [Ras05] and larger, more complex models as hybrid systems (HSs). In fact, the TA formalism is a specialisation (subclass) of the more generic HS formalism. HSs are quite similar to TA, but they allow maximum flexibility on the interaction with clocks. Clocks – here referred to as *continuous variables* – can evolve at arbitrary rates (rather than at a constant grow rate of 1 per time-unit). These rates are usually defined using ODEs and specified by providing the derivative of the variable value at each state separately. Additionally, at state transitions, variables can be set to any value, as opposed to just being reset to zero in TA. Due to the infinite and unbounded state space of HSs, their properties (e.g. reachability or liveness) have been shown to be undecidable in the general case. Some subclasses, however, such as linear and rectangular HA, are analysable. The boundaries between decidability and undecidability of HA have been eagerly studied [Hen+98; PV94; Alu11; AHH96].

The popularity of TA and HSs has also led to the development of various tools which allow the simulation and verification of continuous-time-discrete-state systems. Well-known representatives are UPPAAL [LPY97] and Kronos [Yov97] for TA and Simulink/Stateflow [Raj+18], Modelica [FE98], and HyVisual [Bro+05] for HSs. A thorough comparison of various popular HSs tools is available in [Car+06].

2.3.6 Discrete Event System Specification

Discrete event simulation [Pag95] has been widely studied in the past for its simplicity and efficiency in model simulation. This family of formalisms is based on classic automata theory but extends the concepts and introduces additional features such as time information or logical hierarchies. One highly popular representative of these systems is the Discrete Event System Specification (DEVS) [Zei76] formalism family which allow the modelling of discrete event systems and timed system evolution. Models are specified using discrete states with continuous time advancements.

Classic DEVS specifies models, consisting of states and transitions, that define input and output events. Each state is associated with a (non-negative) real number or infinity (∞) that constitutes its lifetime. State transitions can be of two kinds, internal and external. Internal transitions are activated when the system spent enough time, i.e. the state's lifetime, within the corresponding state. External transitions are triggered upon observation of input events. The formalism distinguishes between *atomic* (single-component) systems and hierarchical compositions of DEVS components, called *coupled* DEVS. There exist simulation algorithms for both atomic and coupled DEVS [Zei84]. DEVS is based on determinism and can only specify reactive behaviour but does not allow the continuous evolution of variables.

Over the years, various extensions of DEVS have been proposed to overcome certain limitations. STDEVS [CKW10] provides the semantics of stochastic transitions and extends the formalism by adding non-determinism. [Hwa11] identifies that DEVS's determinism renders it in fact a subclass of more general (non-deterministic) timed event systems (TES). [CZ94] introduces parallel execution and simulation to the formalism and [Hon+97] proposes the addition of real-time modelling.

Hybrid DEVS is another modification of DEVS that allows continuous evolution of state variables. While there exist several ways to do so, the solutions come with drawbacks in complexity (e.g. the quantized state system (QSS) [ZS98; KJ01] approach) or require external data structures and calculation of continuous behaviour (i.e. the *wrapper* solution). The latter also needs adaptation of the simulation algorithms. A comparison of these two approaches can be found in [DP12]. The verification of DEVS (see below) has been studied in [DG05]. The reachability of general DEVS has been shown to be undecidable [HG05], but some analysis and verification techniques have been found for a few subclasses [SW09; HZ09].

Due to the popularity of DEVS, many tools have been created to help in the modelling and simulation of discrete event models. PythonDEVS [BV01] is an implementation of DEVS in Python that also offers support for parallel DEVS. Due to its GPL-basis, PythonDEVS is very flexible. PowerDEVS [BK11] is a graphical tool for the modelling of DEVS systems and QSSs, the latter allowing for seemingly continuous value evolution. ModelicaDEVS [BC06] is a reimplement of PowerDEVS in the Modelica language. SimPy [Tea18] is another Python library that focuses on the general modelling and simulation of discrete event systems [Mat08].

2.3.7 Petri Nets

Petri nets (PNs) [Pet62] are a family of formalisms which allow the efficient representation of concurrent processes within complex systems [BKL19]. The models are defined using a graphical syntax, consisting of *places* (circles) and *transitions* (rectangles) which are connected to a directed, bipartite graph using arcs. Data, resources and control flows are modelled as *tokens* (represented as dots) which are stored in places and can be produced/consumed by “firing” transitions. The operational semantics are based on one simple rule: when firing a transition, it consumes tokens from all its precondition places, i.e. the places that have an outgoing edge to the transition, and produces tokens in all postcondition places, i.e. the places that have a directly incoming edge from the transition. To specify the number of produced/consumed tokens, pre- and postcondition arcs can be annotated with natural numbers, specifying their respective *weights*. These kinds of basic PNs are often referred to as place/transition nets, to distinguish them from more complex PN types.

Such complex PN extend the base formalism by allowing the specification of different kinds of tokens. One of the most studied extensions are coloured Petri nets (CPNs) [Jen96], where each token is a value from a value type (a.k.a. “colour”). The use of colours in CPNs allows the more compact modelling and representation of systems. Various other extensions of the formalism allow the representation of complex values and data as tokens in high-level Petri nets (HLPNs). In HLPNs, transitions and arcs are extended with *guards*, which evaluate the token values and assert a certain token configuration before transition firings. Algebraic Petri nets [Vau86] are a special form of CPNs which specify token types using algebraic data types [EM85]. These special type specifications allow the efficient customisation of data and use of theorem proving techniques. Time has been added to the PNs in several forms [Pop13] such as Time Petri nets, where transitions can only fire at certain times, Timed Petri nets, where time is a token parameter, and Petri nets with Time Windows, where transitions can only fire within certain time frames.

Many algorithms exist to efficiently simulate PNs and effortlessly obtain model properties such as reachability, liveness and boundedness [JLL77]. One often-encountered issue when analysing Petri nets is state-space explosion [Pel09]. Over the years many techniques have been developed and applied to PNs to render them more easily analysable. These include for instance symmetry [CGP99] and partial order reductions [God91; KP89; Val92], the employment of Büchi automata [VW86] or the use of bounded [Bie+99], distributed [GMS01; Bar+05b; Bar+05a], parallel [SD97; BBR07] and symbolic [Bur+94; Bur+92; CBM90] model checking techniques. Another helpful technique is the use of decision diagrams (DDs) [Bry86], which allow the compact representation of states and optimised state space exploration.

Recently there is a growing trend towards the representation of continuous and hybrid behaviour in Petri nets. Continuous Petri nets [RHS07; AD98] are models where transitions consume and produce infinitesimal amounts of tokens when fired. Thus, the actual behaviour can be evaluated by the observation of the net over time. The transitions should therefore be interpreted as streams of tokens exiting and entering places, the rates are defined in the transitions. The merging of discrete

Petri nets with time and continuous Petri nets results in the creation of hybrid Petri nets [DA01; DA10]. The formalism extends classic Petri nets to achieve a powerful modelling formalism that is comparable to hybrid automata. In fact, there exist translations between these formalisms, so that the simulation and verification of hybrid Petri nets can benefit from existing HA-tools and algorithms [GAS05].

2.3.8 Bond Graphs

Bond graphs [Pay61; Bro99] are a modelling technique that focuses on the flow of energy in different forms. The formalism's foundation is the acknowledgement of the similarities of different energy domains (electrical, hydraulic, mechanical) and the dual forces of "flow" (e.g. current, velocity) and "effort" (e.g. voltage, force) that in combination create power. Bond graphs are created domain-independently, which means that the formalism uses the same notational form for all energy domains. Domain switches are trivial e.g. using *gyrator* and *transformer* elements.

The models represent idealised forms of energy influences, where nodes model idealised subsystems and edges (a.k.a. "bonds") model the energy transfers, each bond expressing effort and flow. Bond graphs use a graphical notation with directed edges. Each edge is annotated with a "harpoon" tip, which shows the positive flow of energy and a perpendicular line that indicates the causality of the bond. The formalism is actively used in engineering domains where it is used for prototyping of a system's energy flows. From an initial design it is possible to further refine the subsystems by decreasing the granularity until a precise enough model is obtained. Furthermore, bond graphs can also be used for causality determination within complex systems.

2.4 Summary

This chapter examines the current state of the art of CPS modelling. Initially, a general understanding of modelling approaches including systems concerns and aspects is provided, alongside the introduction of essential vocabulary. The second part of the chapter reviews different approaches to systems modelling and describes various formalisms and languages to highlight their respective strong points and differences. Note that this summary does not create a dedicated comparison of the presented approaches. The next chapter provides a thorough evaluation of these languages to find the most appropriate candidate for the modelling of resource flow CPSs.

Chapter 3

Resource Flow Modelling – Analysis

The previous chapter introduces numerous formalisms, languages and tools that are actively used by the modelling and simulation community for the design and analysis of CPSs. Oftentimes choosing the right one is non-trivial and requires trade-offs, as each has its own specific advantages and strengths. The choice of the most appropriate candidate for a given modelling project or domain depends on many factors. A major influence are the system aspects to be modelled (e.g. architecture, behaviour, security). Good candidate languages and tools allow the expression of these aspects for the target domain at hand and support the user during their modelling. Another factor are the individual model developers themselves, as their pre-existing knowledge or familiarity with certain tools might ease the learning phase.

One of the main goals of this research project is to enable the rapid modelling, efficient simulation and precise verification of systems that are based on the flow of physical resources. Indeed, expert modellers should be provided with the means to quickly prototype their systems and novice users with a low-entry-barrier framework¹ to experience the benefits of systems modelling. Additionally, both user groups share the common need of creating their models using language concepts that are semantically close to the target domain. To find such a formalism, it is necessary to analyse the systems that should be modelled and find the features and properties that need to be expressed and hence must be supported by a candidate language.

This chapter investigates the modelling aspects of CPSs such as inter-connected smart homes, interactive office automation applications and automated gardening installations. As each modelling language is geared towards the support of different viewpoints, it is necessary to find an appropriate language for such resource-flow CPSs. The search for appropriate candidate languages is guided by the analysis of three case study CPSs. The systems were designed so that the required properties of the target systems can be evaluated. Section 3.1 introduces the set of case studies, which comprises three sample systems from different domains:

¹ Evidently, the difficulty to learn a formalism is a soft-criterion that is hard to evaluate objectively. Nonetheless, Section 3.3 introduces properties to judge languages using published experience studies, usability reports and key metrics. More thorough analysis frameworks have been proposed by other researchers (e.g. [BAG18]). Their systematic application on the languages in this thesis is considered future work.

1. The first one, a smart home system, studies observable influences such as electrical power, hot water and noise. It features a standard electrical power mains for power supply and solar power system, an electrical hot water heater, a shower and various “smart” home appliances (such as an autonomous vacuum cleaner, IoT television and dishwasher). During the day, the system uses solar panels to produce and a battery to stock electricity.
2. The second case study, an office automation system, monitors the temperature and consumption of electricity in a part of an office building (e.g. one floor). The system controls lamps, blinds and air conditioning units to influence the light and temperature, and asserts safety in the building, based on presence and environmental sensors.
3. Lastly, an automated gardening application is under study. The application uses light, temperature and soil moisture sensors to automatically control growing lamps and watering systems to grow plants indoors. The system’s aim is to provide the best environment for plants to grow for maximum harvest.

Next to expressing the components’ state and evolution over time, each of these applications requires a modelling language or tool that is capable of representing the flow of physical resources (e.g. light, water, electricity, noise) between components.

Experience has shown that not all languages and formalisms are suitable for the modelling of systems that are defined by the flow of physical resources (“resource flow systems”). Some of them lack essential features, others are highly complex and require a lot of effort. Section 3.2 analyses the requirements on languages that are apt for the modelling of resource flow CPSs. The resulting set of criteria is, extended by other measures, used to evaluate existing languages and formalisms. Section 3.3 elaborates on this study and discusses its results.

3.1 Case Study Systems

This section introduces three case study systems to subsequently analyse their common modelling aspects and extract shared requirements. The systems are examples of custom assembly CPSs whose behaviour is built on the flow of physical resources. Before diving into the individual case studies, it is of interest to clarify the kind of systems that are targeted by our research. In general, it is hard to provide a precise description of the systems that a language or formalism serves. Especially in the domain of CPSs, the inherent variety and heterogeneity of components makes a globally valid classification difficult, if not impossible. Nevertheless, the remainder of this section will list indicators so as to more clearly judge which systems this project is intended for. As previously mentioned, the systems in our focus consist of off-the-shelf components, that are then composed to larger systems. Thus, the functionality and interfaces are defined by the chosen devices and usually cannot be modified. The three, aforementioned case studies are good representatives of such installations, but they also provide good indicators for a range of other measures.

Size The first of these factors is the systems' size. Our research targets applications that would be classified as "small" when compared to CPSs in classical engineering domains such as transportation, avionics or production line manufacturing. Both, the number of devices, and the variety of component types remains easily manageable. Typical systems are composed of a handful to a few dozen individual components, with clear information about the resources and data flows between the devices.

Developers The system size is further reflected in the number of system developers. Our target systems are usually created and modelled by individual system developers (e.g. private projects such as smart homes), or small teams of two or three modellers, for professional applications such as office automation. The target audience consists of newcomers to the domain, who are interested for instance simulating and verifying that their smart homes systems are working as intended, and professional CPS creators who are in need of a simple, easy-to-use modelling tool for the design and installation of e.g. automated control systems.

Connectivity The devices in our target systems are mostly connected by the continuous transfer of resources and simple communication interfaces. As opposed to other modelling tools, this thesis does not aim for the simulation and verification of complex communication protocols. The signals between e.g. IoT components are modelled on an abstract level. Messages are simple and trigger actions (e.g. "start" and "stop"). The focus lies clearly on the modelling of the reactive interplay between devices, rather factors such as correctness or security of digital communication.

Non-Criticality A final characteristic of the systems in our focus is their non-criticality. Specifically, this means that the systems under study do not fall into the category of "critical systems". Such installations are usually highly complex real-time systems which require special attention to not endanger human life or health, or can have significant impact on the infrastructure and financial well-being of large organisations. While our systems still might be intended to verify the correctness and safety of some systems (e.g. an backup lightning system in an office building), they are not directly responsible for human life or health. Thus, for example, the modelled backup lighting system might be triggered in case of power cuts. For emergency situations such as fires, a dedicated, safety-critical system should be installed and verified using expert tools that are intended for this specific purpose.

3.1.1 Smart Home

The smart home system is a home automation [Bru+11] (a.k.a. *domotics*) installation using IoT gadgets and similar devices. The goal is to create a model thereof, to simulate system behaviour and resource consumption. Specifically of interest is the usage of electricity and hot water within the system, and the discovery of potential for resource savings. The smart home uses a controller to schedule the execution of tasks such as hot water provision and vacuum cleaning. Additionally, the model should be

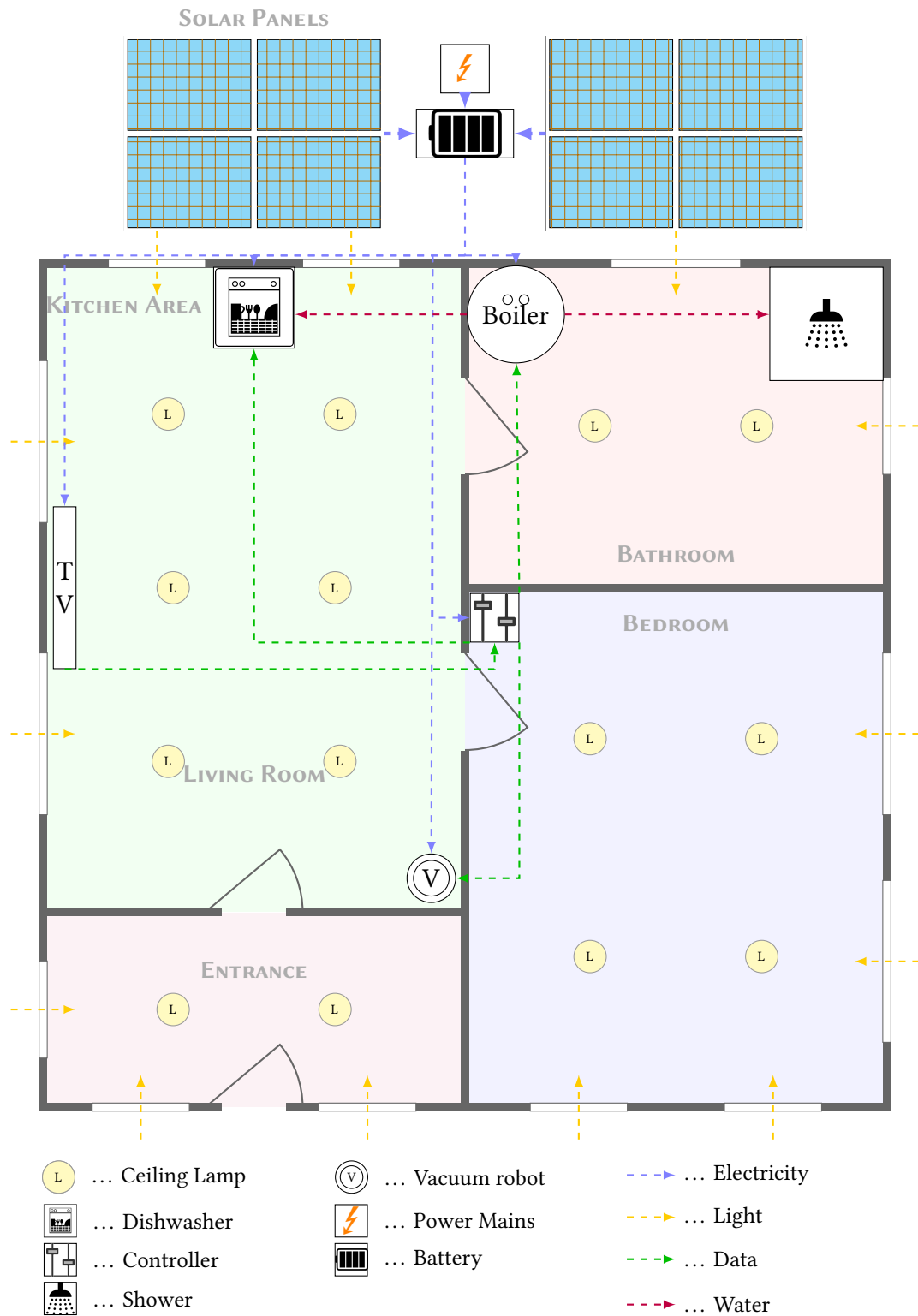


Figure 3.1 – The floorplan of a smart home system with resource flow annotations. Note that the lamps' data connection (from the controller) and electricity connection (from the battery) are not shown.

used to create execution schedules that take user comfort into account (e.g. perform noisy jobs when the user is not at home).

The assumed physical environment is a house, consisting of four rooms (entrance, living room, bedroom and bathroom) as shown in Figure 3.1. The house is connected to the central power grid and has therefore a theoretically unlimited source of electricity, except for times of power outages. To save money, there are photovoltaic power cells installed on the roof, which produce electricity on sunny days. These panels are connected to a battery that accumulates the solar power if the house's consumption is lower than the solar production. The battery has a direct pass-through connection to the power mains and switches automatically when it is depleted. For emergency purposes the battery always keeps a certain minimum charge (about 10 % of its capacity). This reserve is only used during a power outage, i.e. when power supply is shut down and therefore cannot be used.

Hot water is produced in an electrical water boiler, which has a very limited capacity (e.g. 60 litres) and is set to heat the water (e.g. to 55 °C). The hot water is used by the dishwasher and when the home-owner takes a shower. While the shower continuously draws water (e.g. 8 litres per minute), the dishwasher quickly fills its local reservoir once before and once halfway through a washing cycle.

The house possesses large windows in every room and is therefore well lit throughout the day. In the evenings and during the night there is a need for additional light sources. All lamps in the house are remote controllable through software APIs. The house further features other "smart" devices such as an autonomous vacuum cleaner, a TV and a dishwasher, which can be observed and controlled using APIs. These interfaces can be polled for the current device status (e.g. on, off or cleaning for dishwasher and vacuum) and used to execute various actions (e.g. start cleaning). The vacuum robot is assumed to be very noisy and therefore should not be running when the user is at home. Its battery is limited so that it can clean about half of the house's surface before having to return to its base station for charging. The dishwasher is less noisy and can be run while the home-owner is present. There are however certain time periods when its activation is disturbing, namely when the user is watching TV or when she is in bed. The TV is manually controlled, its API, however, can be used to discover if the dishwasher should not be running.

3.1.2 Office Automation

The office automation system's purpose is to monitor and control the light and temperature of an office building (or a part of it, such as a floor). The system consists of three offices which are all located along a hallway on the same floor. Each room has at least one window through which sunlight can enter. The lights in each office can be controlled using a switch with the three options *on*, *off* and *auto*. The hallway's switch is always set to *auto*. The first two settings turn the offices' lights on and off. *Auto* regulates the lamps depending on the amount of sunlight and guarantees a certain minimum brightness in the room. The *auto*-mode is further using a presence sensor, so that the lamps are not turned on when the workforce is not present (e.g. during the night). Upon detection of user presence, the lamps are turned on immedi-

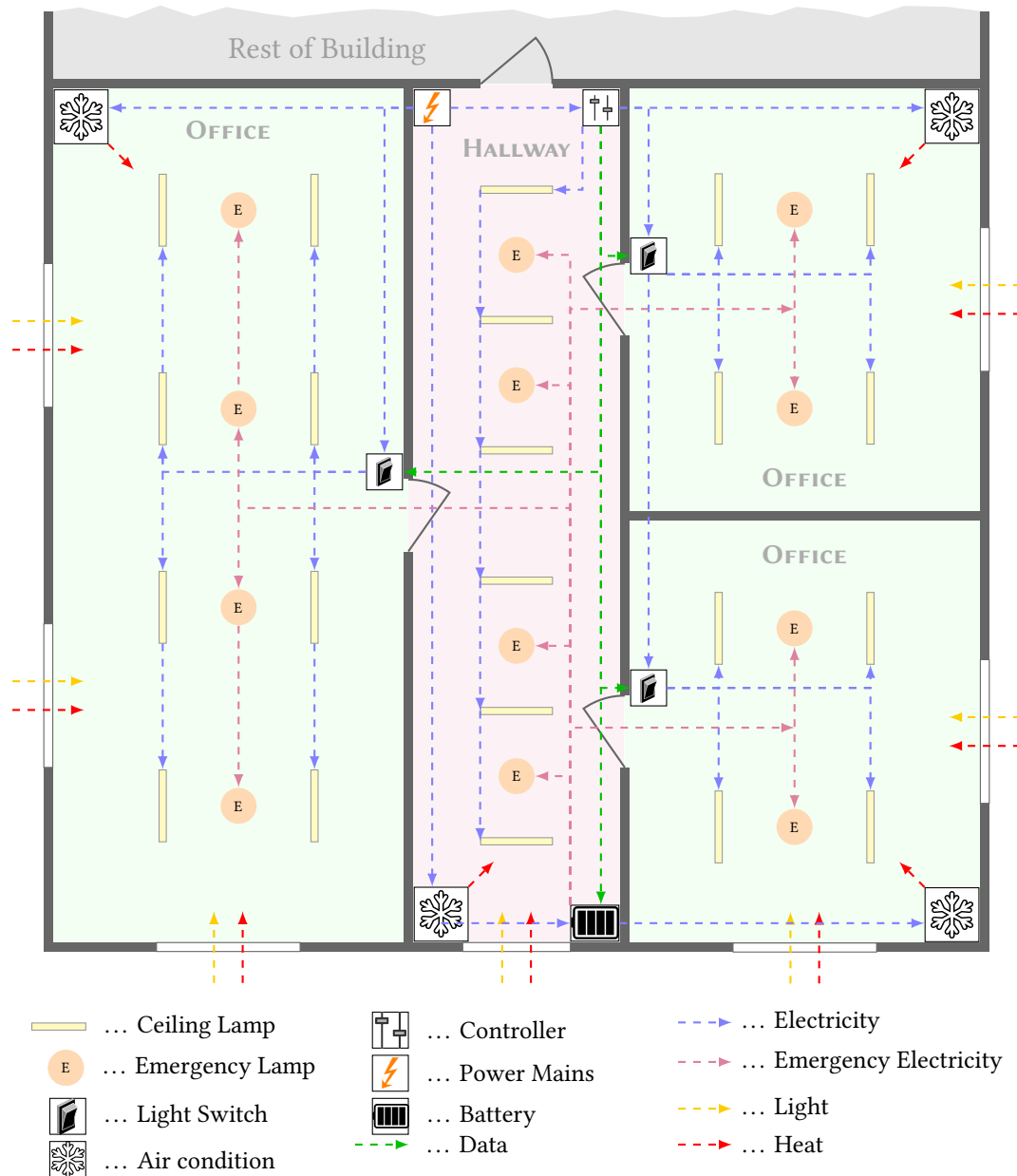


Figure 3.2 – The floorplan of the office system, annotated with the resources that flow between the system components.

ately and left on until five minutes after the user left the office, or immediately when the light switch has been set to on/off manually. For security purposes the lights in the hallway should always be on when a user is in the hallway or in an office.

Next to the lighting system, each office and the hallway are equipped with an air conditioning (A/C) unit. There are four different power settings for the A/C's fan (off, 1, 2 and 3). Depending on the setting, the device requires a different amount of electricity (0, 500, 700 and 900 watts for off, 1, 2 and 3, respectively).

The temperature inside each room of the building depends on the A/C's output, the office size (i.e. the volume of air in the office), the outside temperature and

whether the sun shines into the office. This means that during periods of sunshine a certain amount of heat energy (measured in watt) is added to the office. To avoid the sunshine, the window blinds can be regulated to open, half open and closed. No heat/sunshine is added to the office when the blinds are closed². Another influence, however, is the outside weather which interacts through the large, single glazed windows. Due to this impact the temperature rises (respectively sinks) when it is warmer (resp. colder) outside than in the office.

The office system is powered by electricity through a three kilowatt electrical power mains. If the electricity usage exceeds 3 kW, the system's fuse trips and cuts off all electricity in the system. This situation can occur during peak usage (i.e. all A/C units use maximum power, all lamps are turned on, the motors for the window blinds are activated). Occasionally, the circuit breaker also triggers when a light bulb burns out and the electricity surge spikes. In this model, we assume that this can only happen at the moment when a lamp is turned on, not during the lamp's operation.

In any way, as soon as the electricity is cut – due to a tripped fuse or a general power cut – an emergency system is activated. This backup system uses an alternative electrical circuit to run emergency lamps which point the way to the outside. The energy is drawn from a battery which is kept full by the main power supply.

The model is used to simulate the system behaviour during various scenarios such as power outages and tripped fuses. Additionally, the energy consumption from temperature control and lighting throughout normal operation can be studied and used to develop better, more energy-efficient controller configurations.

3.1.3 Automated Gardening

The last case study is an automated gardening application. Its purpose is to support the growth of fruit, vegetables and herbs indoors. The plants are grown in a south-facing room with large windows, in soil-filled trays, pots and boxes of different sizes.

As the plants' requirements differ from species to species, a controller is responsible for ensuring that each plant's growing conditions match its specific needs. This means that a plant should receive enough light and water per day, while at the same time respecting minimum and maximum constraints. Additionally, each plant type has its specific preferred temperature range. To ensure that these parameters are met for each individual plant, the system's controlling unit uses various sensor measures, including temperature, soil moisture and light intensity. To modify the plants' environment, several actuators can be triggered to provide light, shade and water. The actuators include ceiling lamps, window blinds, special growing lamps with integrated heat modules, and watering systems.

Figure 3.3 shows a schematic representation of one plant subsystem. The hydration system consists of a pump which fills a water reservoir. Small pipes are used to transport the water to the individual plants from there. At regular intervals the controller activates a relay, which triggers a small water pump to fill the reservoir.

²We assume that any increased temperature between blinds and windows is read through the temperature sensor.

The filling of the reservoir can be delayed due to the length of the inflow hose, i.e. when the hose's volume itself is large.

The automated gardening system is also equipped with a germination subsystem, that is used to grow grains into seedlings. This germination module contains several germination boxes in which batches of grains are kept moist and warm until they germinate and grow large enough to be transferred to the larger systems. Each germination box is a semi-closed container (with ventilation openings) and has a small, electrical heater built in, that can be used to regulate the temperature.

The purpose of model creation (i.e. its *modelling intent*) is to verify that the controller is correctly programmed and creates a schedule which asserts that each plant receives the optimal amount of light, water and heat.

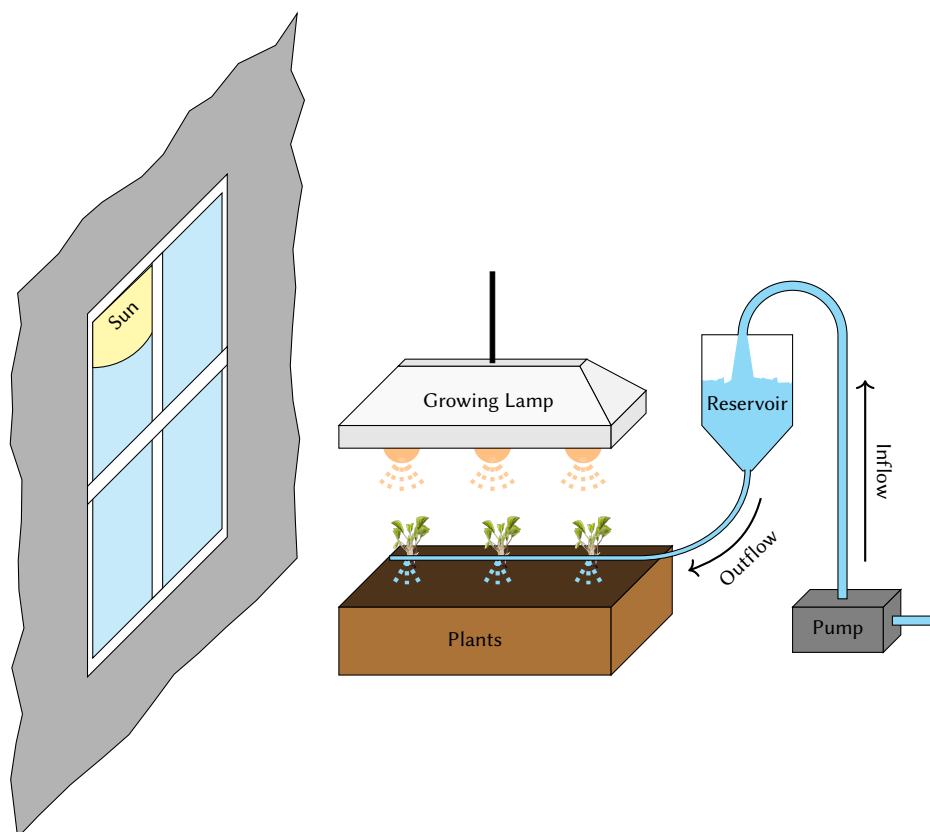


Figure 3.3 – Schema of a plant growing subsystem. The lamp produces heat and light for the plants directly underneath. The hydration system supplies the plants with water. The figure does not show the germination subsystem and the ceiling lamps.

3.2 Modelling Criteria

Even though the case studies represent different usage scenarios and serve different tasks, many of the concepts that need to be modelled are similar. All three systems' behaviour is largely based on the flow of resources such as electricity, light and water. This section describes the aspects that should be met by a language or tool to allow

the modelling of these and similar systems.

Reactivity As the main goal of CPSs is to create components and systems that adapt their behaviour according to changes in their environment, the probably most dominant feature of such systems is reactivity. For example, in all three case study systems, light sensors are used to capture the intensity of environmental light and, in case the measured value is too low (e.g. at sunset), the systems' controllers will open the window blinds or turn on lamps.

Next to observing the system's general environment, some system components modify their behaviour as a consequence of changes in other components or other parts of the system. In the smart home case study, we can observe that starting dishwasher requires a certain amount of hot water. Taking the hot water from the boiler, however, triggers its replenishing with fresh, cold water. This reduces the water temperature within the reservoir and initiates the heating process. The heating itself draws electricity from the battery, if possible, and the power mains otherwise. The example shows that a small action in a subsystem can easily cause (chained) reactions in many other parts of the system.

Synchronism While most system tasks happen over certain time periods (e.g. water heating, charging a battery), the signals within the system are propagated immediately. For example, a room is virtually instantly illuminated by a lamp. The actual time delay (i.e. the speed of light) is negligible for our target applications. Even for energy saving lamps, whose luminosity increases over time, the transition to the on-state and the start of light dissipation is immediate. The synchronism criterion requires that as soon as one value changes, the entire system is checked for possible changed influences between components. This assumption of quasi-immediate responses with negligible delays can also found in real-time systems which guarantee to respond within minimal delays (i.e. before so-called "deadlines").

Parallelism While synchronism and reactivity prescribe each individual subsystem's behaviour and its patterns of change, CPSs usually consist of combinations of many components. Such systems often execute tasks and react to inputs in parallel. In the office system for instance, a tripped fuse will shut down all electrical appliances and start the emergency system at the same time. It is important that the modelling language, as well as its underlying formalism and semantics, support the definition of parallel events and actions, such as the parallel state change of all electrical devices.

Locality, Compositionality & Architecture Despite the exchange of data and resources, the behaviour of a system component is usually based on its internal state and data that is local to the component. For instance, the internal state of the office system's air-conditioning units is not known to other components. Hence, in the model, this state and any other information, which is used to produce its output should also remain encapsulated within the component. This supports the coherent interaction with components through interfaces and enables abstraction.

Locality therefore refers to the language features that allow the clear definition of a communication interface and the “hiding” of internal data. Obviously, encapsulation can be also expressed by e.g. imposing usage conventions³. This strategy can become problematic when the guidelines are ignored (e.g. by unknowing novice users). Instead, languages should provide explicit means to control the visibility of a component features and protect data from outside modification.

The enforcement of locality also requires architectural aspects and means to allow communication and synchronisation between the system’s components. Besides the pure horizontal connection (i.e. components on the same level), it is also necessary to allow hierarchical compositions of components. This allows the creation of larger systems-of-systems but also the decomposition of components.

Continuous Time As mentioned before, many CPSs’ actions occur over time. This means that additionally to instant data transfers the jobs need to take the passing of time into account. For example, while discharging a battery, the amount of stored energy decreases over time, taking a shower continuously reduces the amount of hot water within the boiler, and plants require every day a certain a minimum light intensity for a given time period. Ideally, the chosen formalism allows arbitrarily large or small (e.g. real-valued) timesteps so that all points in time can be analysed (not just the ones that coincide with *ticks*). The time concept must further support continuous influences between components (e.g. a pump filling a water tank).

Non-determinism When it comes to real-world applications, the evolution of a system is not always predictable. The wireless communication with the smart home’s robot vacuum cleaner is not always reliable, for instance. This means that messages can be lost and need to be repeated. Potentially, the robot is not responding at all, meaning that the system has to adapt its behaviour. Similarly, when turning on the office system’s light bulbs, they can either produce light or, alternatively, burn out due to material imperfections and cause a spike in power usage. CPS modelling languages should make it possible to express such scenarios and thus the unpredictability of certain systems, as non-determinism is inherently part of the physical world.

3.3 Evaluation of Languages and Formalisms

The above list of key aspects provides a solid basis for the evaluation of modelling languages. This section explores existing solutions, in the search for an appropriate candidate language or tool for the implementation of resource flow CPSs.

³ Python’s convention for example is that class attributes starting with an underscore (“_”) are internal and not part of an object’s interface. (<https://www.python.org/dev/peps/pep-0008/#descriptive-naming-styles>)

3.3.1 Additional Selection Criteria

The six key features cover the functional aspects of the evaluation, but do not take other factors into account that could sway users. Thus, four more aspects are introduced to decide between modelling languages. Note that the definitions of these additional properties have been adapted to the types of systems we plan to model, i.e. systems with a focus on the flow of resources. An appropriate language or formalism should therefore be easy to master for novice users, but also offer quick navigation and configuration so that it can be used efficiently by professional system developers. The goal is to provide a language that can be used for the precise model analysis of small CPSs and the prototyping of larger applications likewise. Other research projects might put different emphasis on these factors if they target other user groups (e.g. experienced modellers of highly complex or dedicated analyses) or aim to model different systems (e.g. untimed systems require different expressiveness).

Usability & Simplicity The term “usability” describes the ease of a language’s or formalism’s use. In the context of non-experts, usability also describes the difficulty of starting to exploit a language’s features. Influencing factors can be e.g. the necessity of using a proprietary programming language or a lacking intuitiveness of the modelling approach. Simplicity is a specific part of the usability and refers to the number of modelling constructs that need to be learned before being able to effectively and efficiently use the language. Although these properties are often subjective, the review of existing languages is based on experience reports, the number of different language concepts (e.g. required diagram types), and similar data to perform an as objective evaluation as possible.

Expressiveness Another factor is the expressiveness of a language or formalism. It evaluates which components and systems can be modelled, and which features cannot be expressed. An example for this might be the possibility to model synchronous and asynchronous communication. Some languages and formalisms even support the expression of several views within the same model and are thus more flexible and versatile. However, the increased expressiveness comes at the cost of a larger number of language concepts and therefore a steeper learning curve and more complex simulation and verification.

Suitability Expressiveness and usability are factors that predominantly evaluate languages in general. Suitability, on the other hand, relates the modelling language to concrete target domains. It refers to the availability of required domain features (e.g. the possibility to model resource types and transformations) within the language. The aspect also includes the complexity of the extension process, i.e. how difficult it is to add new concepts if they are not provided by default. Lack of suitability results in a wide semantic gap between the model and the system that should be expressed, which in turn causes communication and interpretation issues at best, and flawed verification and wrong system behaviour in the worst case.

Formal Basis The availability of a formal syntax or semantics is of high importance, especially for the analysis and verification of CPS models. Formal verification is a powerful tool that can be used to guarantee that certain unfavourable system configurations can never be reached. An example use case is to prove that the emergency power system can always run for at least 15 minutes, irrespective of the circumstances. Thus, it can be used to assert the confidence in a system and avoid unwanted system design flaws. More formally expressed, formal verification techniques can establish the proof for reachability and liveness properties.

3.3.2 Language Evaluations

The rest of this chapter uses the above criteria to evaluate some of the languages and formalisms of Chapter 2. The focus is placed on a selection of representative candidates that are actively used for CPS modelling. The most promising candidates were chosen, based on an informal upfront evaluation.

UML & MARTE UML has been successfully used to model numerous projects. The availability of various diagram types – each dedicated to a specific purpose – renders the language powerful and versatile. UML diagrams offer features that allow the specification of reactive, synchronous, parallel and non-deterministic systems. MARTE, UML’s extension for RTE systems, adds missing features for the modelling of resource flows and the ability to express continuous actions and timing constraints.

The biggest drawback of UML and MARTE is their complexity. UML itself defines over a dozen different diagram types, each with its own syntax and abstract semantics. MARTE extends the language, but also requires the use of further languages, such as the Object Constraint Language (OCL) [WK98]. The result is a highly expressive but very generic language, whose capabilities often exceed the required features [ERG08] and lead to complexity and confusion of newcomers. An adoption of MARTE poses difficulties when integrating with UML, for example when trying to bridge the semantic gap between the two languages [Iqb+12]. In general, the use of UML, similar to SysML, is complicated and experience shows that novice users need several months to obtain even moderate modelling skills [Fri09].

Another caveat is the fact that the semantics are not formally specified [Inf06]. While this vouches for reuse of the language for various projects with different semantics (e.g. using UML profiles), it effectively makes the exact meaning of models ambiguous unless specifically clarified. The lack of formal semantics also poses problems for simulation, verification and code generation, as the tools developed for one project, might not be reusable for another without significant adaptation.

SDL SDL is a strong candidate for the modelling of the systems. Its graphical syntax is easy to learn and understand. Component behaviour is specified using extended finite state machines and encapsulated in *agents*. Systems are specified through a hierarchical architecture that is based on the composition of agents. Its design is reactive, as agents can perform their processing upon receipt of input signals. SDL’s timing constraints are modelled using *timers* that also trigger a signal upon expiration. The

language's formal basis is a significant strong point and allows simulation and formal verification [Dol03]. SDL's weak point with respect to our requirements, is that all signals are asynchronous. This contradicts our view of CPS, where influences and signals are synchronous, and messages are processed immediately. Another minor drawback is that SDL was developed for communication systems, hence its concepts are geared towards the representation of data, rather than physical units.

SDL experiences weaknesses when it comes to the expressiveness of its real time concepts and the usability in some situations. The authors of [Boz+00] highlight some issues and propose a profile-based approach which uses different semantic profiles for tasks such as simulation and formal verification. This solution, however, further adds complexity by varying SDL's semantics, which is challenging for newcomers.

Architecture Description Languages In general, ADLs are dedicated to solving a specific task. However, by adding behaviour descriptions to these architecture models the formalisms become very powerful. For example, AADL's Behavioural Annex is one such expansion which uses hybrid automata to add behaviour modelling. The extension integrates various features to the text-based language, including enhanced data type definitions and the specification of subprogram behaviour, but lacks a formal definition. The available formalised parts allow for some analysis and evaluations, but do not support any formal verification techniques. For the purpose of modelling resource flow CPSs, AADL is not an ideal fit in terms of suitability, as the language focuses on a lower abstraction level (memory, processes, hardware). The lack in suitability is emphasised in its additional shortcomings at other aspects (e.g. component compositions) [Did+07]. The authors of this publication also highlight the difficulties of starting to model with AADL.

MontiArcAutomaton is another extension of an ADL, in this case the MontiArc ADL. It too uses automata to add behaviour to the system descriptions. The language's advantage is that MontiArc itself is based on the formal semantics of FOCUS [Bro+92], which can therefore be leveraged. This means that languages which are based on MontiArc can be used for formal verification and evaluations. MontiArcAutomaton however lacks support for continuous time evolution. Instead, it is based on time-synchronous or cycle-based (*tick*) evolution and lacks support of clocks and similar time concepts. Further, it only supports MontiArc's asynchronous message passing system and hence contradicts the synchronism requirement.

SystemC SystemC is the probably most flexible representative of HDLs. It convinces through its C++ basis, which is a great advantage for users that are already familiar with programming in general and C++ in particular. One problem of using SystemC for resource flow systems is that most predefined concepts are defined for low-level electronic circuits and that the language therefore lacks support for higher-level concepts. However, using a GPL as basis has the advantage that user-defined additions can be integrated, and highly complex systems designed (even though the creation of domain-libraries might require significant effort).

The drawbacks of SystemC include the lack of true support for parallelism. In fact, its concurrency concept is based on cooperative multi-threading (a.k.a. "corou-

tines”). This means that a SystemC thread executes until it voluntarily yields control and another thread is chosen to take over. This choice is based on a reactive paradigm, where each thread specifies a time interval that has to pass or a system event that needs to be observed, at which it aims regain control. Even though no time passes during thread execution (this concept is called *delta cycles*), it is impossible to truly execute threads in parallel. The execution order of threads is under-specified by the language and varies depending on the chosen SystemC simulator and compiler. This can lead to different results when modelling dependencies between “parallel” threads. This design concept also influences the synchronism of the language. While signals are sent immediately, they are only acted upon by the receiver thread when it is triggered by the simulation kernel. As the choice of which thread to run next is not predefined by SystemC, it is possible that different simulation tools will choose different execution orders, which can potentially lead to unforeseen behaviour and race conditions. Finally, SystemC neither offers built-in support for continuous time nor explicit non-deterministic behaviour specification. While latter can be simulated using the underlying C++ mechanisms, support for continuous time concepts is difficult to add and involves extensive modification of the base language.

Synchronous Languages Esterel is a synchronous language which defines operator nodes that execute in regular intervals according to the base clock. The synchronous hypothesis (upon which the language is based) assumes that all node inputs are available at the start of the function. Operators are defined in a network which cannot contain loops (similar to Kahn networks [Kah74]), to uphold the theory of zero-time computation. Though Esterel’s syntax is imperative and should be easy-to-learn for experienced programmers, the language itself comes with a steep learning curve, due to the underlying synchronous, iteration-based paradigm. The language is powerful, but requires much experience and the learning of best practices which can be somewhat tedious for newcomers. Similar to most synchronous languages, Esterel supports reactivity and parallel programming. On the other hand, the synchronism principle voids all support for non-determinism. Its lack of an integrated concept of time further removes the support for continuous evolution. Instead, the language requires external signals to provide the computation points. Hence, periodic computations need to be triggered by an external time signal.

Lustre is another representative of the synchronous languages. The language builds on the declarative programming paradigm. Lustre provides a sound model of computation with predictable performance, property verification methods (liveness, deadlock freedom), and predictable buffering. The language’s declarative paradigm is difficult to grasp for novice users, however. Furthermore, extensions for new concepts and data types (e.g. resources in our case) are difficult to implement.

Recently, Zélus [BP13], a Lustre-inspired synchronous language, aims to overcome the limitations of classical synchronous languages by adding support for ODEs to model continuous behaviour. However, just as Lustre, Zélus too suffers from a steep learning curve, partially owed to its difficult syntax. Being a very recent development, the full scope of Zélus’ capabilities has not been entirely studied yet, and future extensions and improvements of the language are likely. At the time of writ-

ing however, Zélus lacks support for the local encapsulation of information. To the best of my knowledge, it appears that non-deterministic system models are not supported. Further, as the language is still under active development, its expressiveness is hard to estimate and is likely to change with future versions.

Hybrid Automata Tools The powerful expressiveness of hybrid automata (HA) led to the development of various tools and languages for their specification and analysis. One of the most commonly used ones is Simulink. Simulink is a graphical programming environment for the modelling and analysis of dynamic systems. After an initial effort to become familiar with Simulink, the tool convinces through large libraries of predefined components, powerful numerical solvers and industry-proven simulation capabilities. The tool is extended by Stateflow, a plugin that allows the definition of discrete automata based on non-deterministic Harel Statecharts [Har87]. In combination, Simulink/Stateflow can be used to model HA. The tool's language allows reactive and parallel modelling and supports system compositions with data encapsulation. The biggest drawbacks of Simulink/Stateflow are the lack of formal execution semantics (although proposals exist; e.g. [HR04]). Depending on the choice of numerical solver, the tool can produce different and, in corner cases, unexpected results. This also impacts the model's synchronism, as sent messages are relayed by the simulation engine which can lead to unforeseen behaviour.

HyVisual is a hybrid systems (HSs) application based on the Ptolemy II multi-formalism simulation and verification platform. It allows the definition of HSs with causal influences. Its academic development background led to high flexibility, so that it supports all key aspects described in the previous section. It further convinces through its formal semantics, which can be leveraged for simulation and verification. However, it also resulted in a lack of usability. The exclusively graphical modelling environment is difficult to understand for newcomers, while at the same time being a tedious burden for advanced power-users, as pointed out in [Car+06].

Modelica is a textual language specification for the creation of multi-domain systems. It supports the acausal definition of complex models. The language's popularity led to the creation of a large standard library, rendering Modelica suitable for many domains such as mechanical, thermal and electrical applications. A caveat however is that the creation of user-defined libraries is non-trivial and requires proficiency in the language. Another drawback of Modelica is that it is a pure language specification and does not offer a reference implementation or formal semantics. Tools such as OpenModelica [Fri+06] and Dymola [Brü+02] can therefore produce different results, despite using the same syntax and model. This can become problematic when it comes to the formal verification of models that require numerical equation solving. In terms of usability the language suffers from the fact that its target audience are modelling experts with good domain knowledge. This leads to a steep learning curve, requiring the learning of many particular language features.

DEVS PowerDEVS is one of the many tools and libraries that have been developed to support the creation, simulation and analysis of DEVS models. It implements the classic DEVS algorithm, which supports atomic and coupled models and thereby

keeps information local. The tool’s graphical user interface (GUI) is based on the graphical DEVS representation and thus easy to learn and use. Further, the tool offers the programmatic extension of functionality, even if it requires strong knowledge of the tool’s internal processes. PowerDEVS prominently features the use of the QSS method, which approximates continuous variable evolution and support for ODEs. On the negative side, important missing features include the support of parallel and non-deterministic models, as neither is part of the classic DEVS formalism. There are extensions to support parallel and stochastic DEVS, and several approaches try to add model checking and formal verification (e.g. DEv-PROMELA [YHF16]). Despite the research in these directions and attempts to approximate continuous evolution using time and state discretisation (see [Van00]), the application of DEVS to modelling continuous resource flows requires a good grasp of the formalism’s details. This increases complexity and lowers the usability for modelling newcomers. There exist other, easy-to-use DEVS tools and libraries that make up for some of these shortcomings (e.g. PythonDEVS), but they do not support the continuous time and ODEs.

3.3.3 Discussion

The results of the individual languages’ analyses are summarised in Table 3.1. Since the evaluation only considered the most promising candidates for CPSs modelling, most languages support reactivity and, except for Zélus, some form of local encapsulation of information. However, when comparing other features several drawbacks can be found: Certain languages do not support continuous behaviour (MontiArcAutomaton, SystemC, Esterel, Lustre), lack expression of parallelism (AADL, SystemC, PowerDEVS), non-determinism (Esterel, PowerDEVS) or synchronous communication (SDL, MontiArcAutomaton, Simulink). The most promising hybrid system candidates lack formal semantics (Simulink, Modelica) or usability (e.g. HyVisual’s tedious graphical editor) and often require large efforts to add domain-concepts.

Even though UML/MARTE and HyVisual tick all boxes for the key aspects, UML’s complex web of diagrams, MARTE’s OCL-based annotations and HyVisual’s complex adaptation procedure are far from simple or beginner-friendly. The table also highlights that few languages are equipped with or allow user-friendly extension to support resource flow concepts (e.g. definition of own resource types).

3.4 Summary

Custom assembly CPSs such as smart homes, office automation systems and automated gardening installations require modelling languages/tools that are capable of representing the flow of physical influences (e.g. light, water, electricity) between components, in addition to expressing the component’s state and evolution over time. This section describes three case study systems and extracts similarities and basic modelling aspects. The analysis of the three systems led to the discovery of six key concerns whose representation should be supported by a CPS language:

1. *Reactivity*; to allow system models to react to environment changes.

2. *Parallelism*; for the modelling of concurrent phenomena and behaviour.
3. *Synchronism*; since CPS behaviour changes are primarily instantaneous.
4. *Locality & Architecture*; for component separation and system composition.
5. *Continuous Time*; to model continuous resource flows and behaviour over time.
6. *Non-determinism*; as it is inherent to many physical systems.

The above list, extended by the factors *simplicity*, *expressiveness* and availability *formal semantics*, as well as *usability* and *suitability*, served as a reference guide for the search of a suitable, existing modelling language. The analysis of several existing candidates, representing various modelling paradigms, revealed that very few languages support all six key features, and that the ones that do show severe lacks in terms of usability and suitability for our target domain.

Table 3.1 – Evaluation of several modelling languages for resource flow intensive CPSs such as home and office automation or plant growing systems.

Formalism/Tool	Reactivity	Synchronism	Parallelism	Locality	Continuous	Non-determinism	Usability	Expressiveness	Suitability	Formal
UML / MARTE	✓	✓	✓	✓	✓	✓	×	✓	~	~
SDL	✓	×	✓	✓	✓	✓	✓	✓	~	✓
AADL + Beh. Ann	✓	✓	×	✓	✓	✓	×	~	×	~
MontiArcAutomaton	✓	×	✓	✓	×	✓	✓	~	~	✓
SystemC	✓	×	~	✓	×	~	~	✓	~	×
Esterel	✓	✓	✓	~	×	×	×	~	×	✓
Lustre	✓	✓	✓	✓	×	×	×	~	×	✓
Zélus	✓	✓	✓	×	✓	?	×	?	×	×
Simulink/Stateflow	✓	×	✓	✓	✓	✓	~	✓	✓	×
HyVisual	✓	✓	✓	✓	✓	✓	×	✓	~	✓
Modelica	✓	~	✓	✓	✓	✓	×	✓	~	×
PowerDEVS	✓	✓	×	✓	~	×	✓	~	×	✓

Key aspects **Add. criteria**
 Symbol meaning: ✓ (Yes) × (No) ~ (to some extent) ? (unknown)

Chapter 4

The CREST Language

The previous chapter highlights the need for a modelling language that allows the efficient description of resource flows within a cyber-physical system (CPS). The analysis also revealed that only few modelling solutions exist for CPSs within our specific application area. The latter discovery suggests the use of DSLs [Völ+13] to overcome the distance between tool and system domain. DSLs are “*dedicated to a particular domain or problem*” [Rev+00] and aim to facilitate expression of target domain concepts and problems by providing specialised model and language constructs. A well-known example of a DSL is the Structured Query Language (SQL). Its purpose is to simplify the interaction with relational database systems by offering dedicated features for the efficient insertion and querying of database entries.

Following the DSL methodology, the modelling and simulation community embraces the approach by creating domain-specific modelling languages (DSMLs). These languages are, opposed to GPPLs, dedicated to modelling of domain-specific concepts and usually designed by modelling and language design experts in combination with specialists from the target domain. The goal is to create a modelling language or tool that helps domain users to learn and use modelling concepts easily, and ideally to adapt the DSML to the users’ existing knowledge.

This chapter introduces the Continuous REactive SysTEms language (CREST). CREST was developed for the modelling of resource flow intensive CPSs such as smart homes, office automation and intelligent gardening systems. Its design focuses on the support of the six core-principles (reactivity, parallelism, synchronism, locality, continuous time and non-determinism), as identified in the previous chapter. The language’s purpose is the modelling of the reactive behaviour of CPS components and the transfer of physical resources and signals between those components. It enforces the distinction of resources by their type (e.g. light, electricity, heat or switch position) and modelling of their transfer as influences between system components (“entities”). Entities are structured in a strictly hierarchical system view which encourages composition and system-of-systems designs. Their behaviour is modelled using automata, continuous value updates take real-valued time into account. The language semantics preserve dynamic behaviour with arbitrary time granularity, to allow the modelling, precise representation and faithful verification of CPS.

This chapter is structured as follows: Section 4.1 introduces CREST’s graphical syntax, formal language structure and modelling restrictions. Section 4.2 provides details of the language’s semantics, first informally then using structured operational semantics (SOS) rules. Section 4.3 describes the formal basis of syntactic extensions, which increase CREST’s usability without altering its expressiveness. Finally, Section 4.4 analyses CREST’s properties and compares the DSL to related formalisms.

Related Publications The content of this chapter, especially the syntax, semantics and formalisation has been the subject of previous publications, which were composed primarily by the thesis author [KLB18b; KLB18a; KLB17]. This chapter revisits these early works and builds upon them to provide a completed and extended language description.

4.1 Syntax

CREST is a modelling DSL that combines a system’s architectural and behavioural aspects in a coherent language. Its graphical concrete syntax, the *CREST diagram*, was developed to facilitate legibility and allow design discussions. CREST diagrams aim to highlight the transfer of resources between components. The language was created by following well-established design practices such as the *Physics of Notation* [Moo09]. In particular, CREST diagrams establish a one-to-one mapping between semantic concepts and their graphical representations. The choice of symbols asserts that the displayed concepts are easily visually distinguishable by their shape and colour, as suggested by the authors. The reuse of well-known notation forms (e.g. circles and arrows for states and transitions) further increases legibility. The positioning of the symbols within the diagram should highlight the flow of resources throughout the system. Hence, the ordering of elements should “flow” from one side to the other (e.g. left to right). Tool implementations that offer automatic layouting should also follow this strategy. Figure 4.1 displays the complete CREST diagram of a growing lamp, as used by the automated plant growing system (see Section 3.1 – Case Study Systems). It serves as a running example throughout this chapter.

List of Symbols The specification and formalisation of a language’s syntax and semantics usually require the definition of numerous language concepts. CREST is no exception to this rule. To avoid confusion or getting lost in the jungle of symbols and definitions, the first-time reader is strongly advised to keep the list of symbols (Appendix D) close at hand for lookup purposes.

As shown in Figure 4.1, CREST components clearly define their scope. Visually this is represented by a black border, indicating the scope’s limits. A component’s communication interface is drawn on the edge of this scope, while its behaviour and internal structure are placed on the inside.

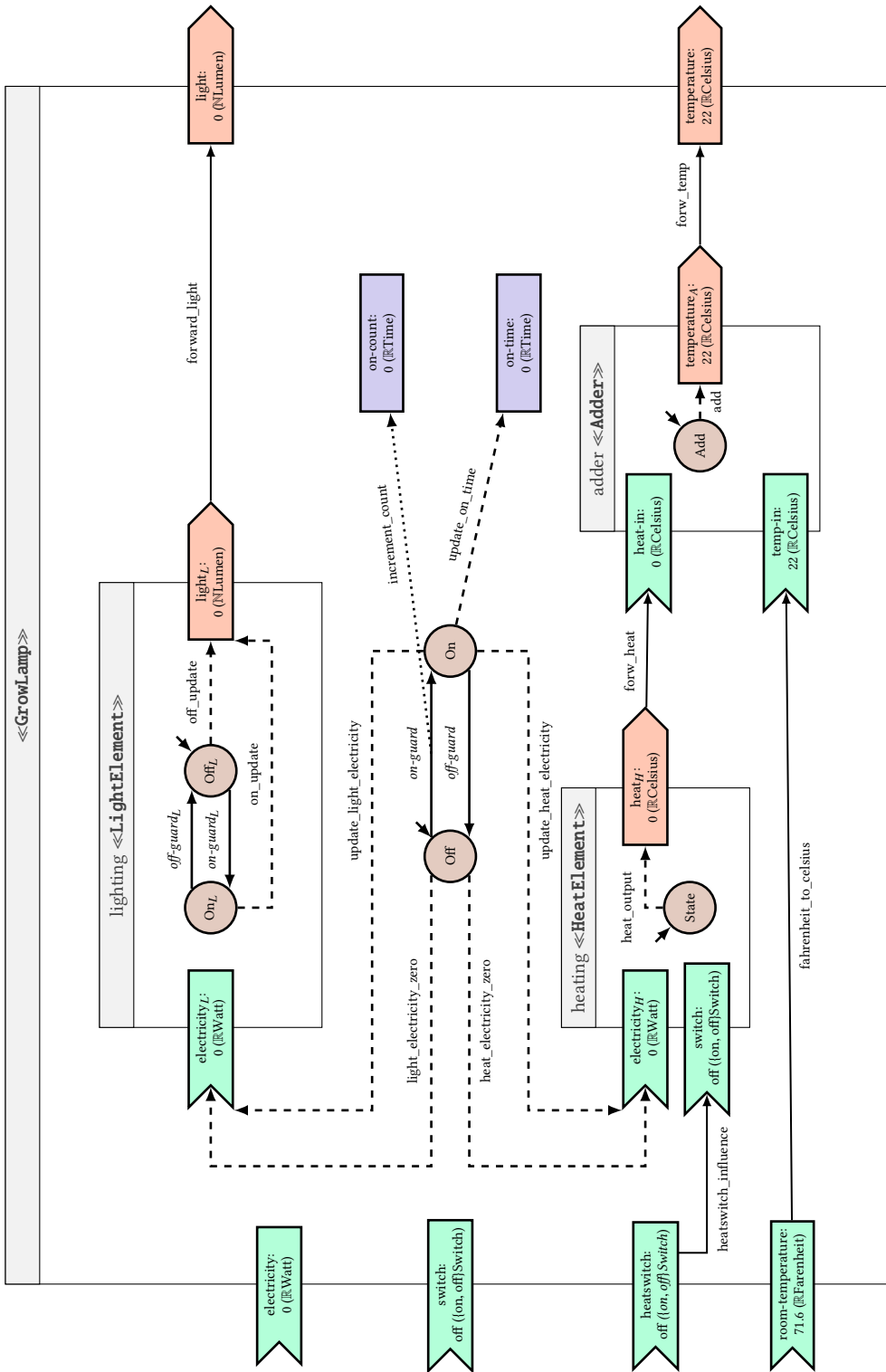





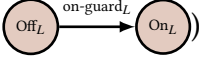
Figure 4.1 – A CREST diagram of a growing lamp entity with subentities. Updates, transitions and influences are annotated with their respective function names. The behaviour of these functions is provided in Appendix A.

System Structure Most non-trivial CPSs are physically and logically structured into subsystems, which can be further divided into components and subcomponents, and so on. Each individual part belongs to exactly one bigger component. CREST follows this system view where CPS are defined as hierarchical compositions. This concept is expressed through the definition of components (called “**entities**”) in a nested tree-structure. Each CREST system model contains one single *root* entity, which represents the “entire system”. This entity can define arbitrarily many subentities, which can contain child entities themselves, etc. The growing lamp for example, contains three subentities: one for light production (**LightElement**), one for heating (**HeatElement**) and an **Adder**, which calculates the sum of two input values.

The strict hierarchy asserts a simplified, localised view on the entity level. Each entity encapsulates its internal structure and can hence be treated as a *black box*. This view facilitates composition, as the entity’s *parent* can treat it as coherent instance, disregarding the entity’s internal structure.

The black box view is completed by the definition of the entity’s communication interface, which consists of **input** () and **output** () ports. Ports are required for the modelling of the resource flow within the system. CREST offers a third kind of port: **locals** (). This port type is not part of the interface, but rather serves as internal storage of data. In the example **on-time** and **on-count** are internal ports. All instances of the three port types are associated with a specific **resource**.

Resources combine the information of a port’s unit and domain, such that any port clearly states which values it can be assigned. The growing lamp specifies units such as watt or lumen. Domains are sets of values, e.g. the natural numbers \mathbb{N} , reals \mathbb{R} or a set of discrete values such as {on, off} (for **Switch** in the example). Thus, a port that specifies **NWatt** as its resource, can be assigned the value **3Watt**, but not **3.14Watt** or **2Lumen**. Next to the resource itself, each port specifies a value from its respective resource as its current value binding.

Entity Behaviour CREST uses FSMs to specify behaviour. Each entity defines a set of **states** and guarded **transitions** between them. Transitions relate *source* states with *target* states and the names of transition *guards* (e.g. ). The transition guard implementations are functions over an entity’s set of port value bindings *bind* and previous port bindings *pre*. The functions’ codomains are Boolean values (**True**, **False**), indicating whether the transition is enabled or not. CREST does not provide a syntax for the definition of guard functions. Instead, the language prescribes the functions’ signature, leaving users to choose a suitable language for the guard specification. Evidently, the choice of transition guard language impacts the expressiveness of the overall CREST models. Chapter 5 presents a Python-based CREST implementation, where transition guards and other language constructs are expressed using program code. It is also possible to use mathematical notation for the guard specification. The following formula shows growing lamp’s **on-guard_L**, for instance.

$$\text{on-guard}_L(\text{bind}, \text{pre}) = \begin{cases} \text{False} & \text{if } \text{bind}(\text{electricity}_L) < 100\text{Watt} \\ \text{True} & \text{if } \text{bind}(\text{electricity}_L) \geq 100\text{Watt} \end{cases}$$

In most situations *pre* is explicitly not used. However, there are particular scenarios in which access to these values is necessary. For instance, they have to be used to discover and analyse port value changes (i.e. $bind(port) \neq pre(port)$). Second, in certain situations they are also required to resolve cyclic dependencies between components (so-called *algebraic loops*), which otherwise could not be supported in CREST. The concept of supporting previous values is present in other languages such as Modelica, Lustre and Esterel through the *pre* operator. CREST’s implementation automatically manages *pre* for the user and in most cases uses it automatically when necessary.

Resource flow The transfer of resources between entity ports can be modelled using **updates** ($- \rightarrow$). Updates are defined for a specific state and target port, relating them to an update function name. If the automaton is in the given state, the update’s function (identified by its name) is permanently executed, and models continuous change. The update’s function implementation operates on the entity’s port values to calculate its target port’s new value. The function has to be written so that the returned value matches the target port’s resource domain. Conceptually, updates are continuously executed, so that the system’s ports always hold the correct values. This means that theoretically, an infinitesimal amount of time passes between two executions of an update. Practically, CREST’s operational semantics assert that the evaluations are performed as often as necessary, as detailed later on in this chapter. The growing lamp defines several updates, such as `update_on_time`, `update_light_electricity` (both in `GrowLamp`) or `heat_output` (in `HeatElement`). Updates further enforce CREST’s *synchronism* principle. Provided the automaton is in the update’s state, the continuous evaluation of update functions always guarantees that the target port’s value is the result of the update function execution, without delay or explicit message passing.

Similar to transition guards, the update functions’ syntax is not strictly defined but constrained by a predefined signature. This means users are free to choose any suitable language to define the functionality. Update functions are executed with the current and previous port bindings *bind* and *pre*, such that *pre* holds the port values before the execution of the update. Additionally, the functions also have access to another parameter δt . This is a value of the system’s time base \mathbb{T} (see Appendix B for details on the time base) and holds the information about the amount of time that has passed since last executing the update function. Hence, update functions can be used to model continuous behaviour and value updates. However, it is practically impossible to simulate systems with infinitesimal time increments. Thus, in implementations of CREST the simulator decides on appropriate δt -intervals, as will be explained in the semantics section. In the growing lamp’s example, the time base is the domain of non-negative real values (i.e. $\mathbb{T} = \mathbb{R}_{\geq 0}$). As an example for an update function, we look at the mathematical definition of `update_on_time`, which continuously accumulates the amount of time that the automaton spent in state on:

$$\text{update_on_time}(bind, pre, \delta t) = pre(\text{on-time}) + \delta t$$

In CPSs, resources are often continuously transferred from one port to another, independent of their entity’s state or the time that has passed. In the example above, the value of growing lamp’s `heat switch` is transferred to the `HeatElement`’s `switch` input port, disregarding whether the lamp is turned on or off. To avoid repeating the specification of the same update function for every state of the entity, CREST offers **influences** (\rightarrow) as a syntactic shortcut. Influences relate a source port to a target port and a function name. The behaviour of influences is similar to updates, with the difference that only one source’s value is used for calculation of the target port value. Neither δt nor any other port values are considered for the calculation. In the growing lamp the influence `fahrenheit_to_celsius` is defined as follows:

$$\text{fahrenheit_to_celsius}(\text{bind}) = (\text{bind}(\text{room-temperature}) - 32) \times 5/9$$

Finally, a third type of resource flow is available in CREST: **actions** ($\cdots\blacktriangleright$). Actions define update functions that are executed during the triggering of transitions. Similar to influences, actions are not allowed to access the δt parameter of the related update functions. The growing lamp scenario defines one action (`increment_count`) that is executed every time the transition from `Off` to `On` is triggered. It is used to count the number of times the lamp has been switched on.

Entity Abstraction CREST diagrams allow complexity management via masking of component internals. This means that an entity can be drawn by displaying only its communication interface. In this case, the masked component should “hide” any subcomponents, states, transitions, local ports, updates and actions. Instead, an icon is shown to indicate that some information is masked. Figure 4.2 shows the CREST diagram of the growing lamp, where information has been hidden. Note that this feature is mostly of notational importance and intended to simplify the legibility of complex system diagrams. It does not carry any particular semantic meaning, except that it potentially abstracts over some underlying behaviour. Thus, no formal syntax and semantics will be defined.

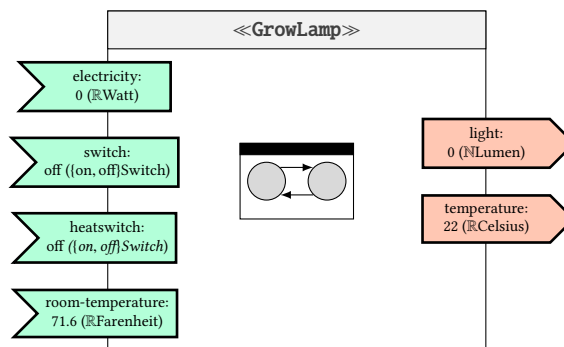


Figure 4.2 – Alternative, masked CREST diagram of the growing lamp, which only shows its communication interface, but masks its internals (states, transitions, subentities, etc.).

4.1.1 Formal Language Structure

Based on the above introduction of the CREST diagram syntax, we will subsequently explore the formal definition of a CREST system's structure and the definition of the sets and functions it is constructed of. To make the concepts easier to grasp, the GrowLamp system (Figure 4.1) is used to explain the concepts. As a complete description would exceed the purpose of illustration, only a few representative examples are presented for each concept instead. It is also important to keep in mind that CREST's formalisation (structure and semantics) is defined on a system-global level. This means that states, transitions, ports, etc. are first defined as a system-wide set and then divided into mutually exclusive sets for each entity to preserve locality.

Notation (Notational convention). To increase the legibility of the various notions, the rest of this chapter uses the following styling convention and notations for mathematical concepts:

- Sets are *Capitalised*;
- Functions are *lowercased*;
- Sets of function names are *Calligraphed*.

This chapter also uses the notion of non-overlapping set partitions (denoted by the \sqcup operator) to define mutually exclusive set partitions.

Notation (Partition of sets). Given a set S , the subsets S_1, \dots, S_n are defined to be a partition of $S = \sqcup_i S_i$ or $S = S_1 \sqcup \dots \sqcup S_n$ iff $\forall i, j, i \neq j \implies S_i \cap S_j = \emptyset$ and $S = \bigcup_{1 \leq i \leq n} S_i$.

Definition 1 (Time Base). CREST is a language that offers the continuous evolution of variables according to time advances. Thus, it is necessary to define the time base \mathbb{T} that a system operates in. Note that \mathbb{T} is not a part of the CREST system itself, but rather an orthogonally defined concept.

Formally, a time base \mathbb{T} is required to satisfy the theory $\text{TIME}_\epsilon^\infty$ (see Appendix B). Accordingly, \mathbb{T} is a linearly ordered commutative monoid ($\text{Time}, +, 0, <$) with infinitesimal element ϵ , infinity element ∞ and monus operator $\dot{-}^1$. Depending on the specific use of the CREST system and the variables, it might be of interest to define further operations (e.g. multiplication, division) on the time base.

Common time bases are the set of positive real numbers $\mathbb{R}_{>0}$ or rational numbers $\mathbb{Q}_{>0}$, extended by ϵ and ∞ . In this dissertation the time base is assumed be the set $\mathbb{R}_{>0} \cup \{\epsilon, \infty\}$, unless specified otherwise. We will also make use of common mathematical operations of this value domain such as multiplication, division, and similar².

¹Monus “ $\dot{-}$ ” is an operator defined for some commutative monoids. In most time bases (e.g. $\mathbb{R}, \mathbb{Q}, \mathbb{N}$) the monus operator is the subtraction “ $-$ ”.

²These operations are assumed to be common knowledge and are thus not be formally defined.

Definition 2 (Types and Values). Given *Units*, a set of resource units, and *Domains*, a set of value domains, the set of resource types is defined as $Types \subseteq Domains \times Units$. The values of a resource type *type* are $\{\langle v, unit \rangle \mid v \in domain, \langle domain, unit \rangle \in Types\}$, where $type = \langle domain, unit \rangle$. The set of all resource values is defined as $Resources = \{\langle v, unit \rangle \mid \exists \langle domain, unit \rangle \in Types \wedge v \in domain\}$. It contains all possible couples of values and units.

For legibility, CREST diagrams often use the simplified notations *domain unit* and *v unit* for a resource type and value. It is therefore possible write e.g. $\mathbb{N}Watt$ and $3Watt$ for $\langle \mathbb{N}, Watt \rangle$ and $\langle 3, Watt \rangle$. Figure 4.1 shows the use of the following resource types:

$$\begin{aligned} Units &= \{Watt, Switch, Celsius, \dots\} \\ Domains &= \{\mathbb{R}, \mathbb{N}, \{on, off\}\} \\ Types &= \{\mathbb{R}Watt, \{on, off\}Switch, \mathbb{R}Celsius, \dots\} \\ Resources &= \{0Watt, onSwitch, 22Celsius, \dots\} \end{aligned}$$

We also define the \in operator on resource values and resource types, to test for compatibility between a value and a type. The compatibility check is used to verify that only appropriate values can be written to a port of a certain resource type. Formally we define that a value is part of a resource type, iff the value is an element of the type's domain, and the value's unit and the type's unit match:

$$\begin{aligned} \forall res = \langle value, unit_1 \rangle \in Resources, \forall type = \langle domain, unit_2 \rangle \in Types, \\ res \in type \Leftrightarrow value \in domain \wedge unit_1 = unit_2 \end{aligned}$$

Hence, we see that e.g. $3Watt \in \mathbb{N}Watt$ and $3Watt \in \mathbb{R}Watt$, but $3Watt \notin \mathbb{N}Lumen$.

Familiar readers might discover similarities to the way SysML 1.4 implements units. In contrast to SysML, however, CREST explicitly states the value domain, but does not define the “quantityKind” to specify what the unit is used for (e.g. “mass”). This means that values can be formally checked for validity but translations between different resource types (e.g. different temperature measures such as Fahrenheit and Celsius) are not directly possible without e.g. the use of updates or influences. Nevertheless, its support would significantly increase usability and is currently being investigated. Our planned approach is built “on top” of the existing language structure and uses ontologies (e.g. of the SI units) for the specification of types and domains. This should allow for instance the possibility for extended checks such as e.g. resource type compatibility and translation correctness. The research results are still preliminary at the time of writing.

Definition 3 (Hierarchy of Entities). CREST components are modelled as *entities*. Each entity can contain other entities, which are referred to as *children* or *subentities*. An entire CREST system's structure forms a rooted tree. The system's entity tree is defined by a set of entity names *Entities*, and a function $parent : Entities \rightarrow Entities \cup \{\perp\}$, which returns the parent of an entity or \perp if it has no parent. The

function $children : Entities \rightarrow \mathcal{P}(Entities)$ returns the direct children of any entity, and the constant $root : Entities$ provides the system's root entity.

$$\begin{aligned} children(e) &= \{e' \mid e' \in Entities \wedge parent(e') = e\} \\ root &= e \text{ s.t. } e \in Entities \wedge parent(e) = \perp \end{aligned}$$

CREST's strict entity structure asserts that there is exactly one entity without parent, which we refer to as the system's *root*. Formally, the constraint is expressed as: $|\{e \mid e \in Entities \wedge parent(e) = \perp\}| = 1$.

Furthermore, we use the terms *ancestors* to refer to the set that is composed of an entity's parent, its parent's parent, etc. Thus, it is inductively defined as

$$\begin{aligned} ancestors(root) &= \emptyset \\ ancestors(e) &= \{parent(e)\} \cup ancestors(parent(e)) \quad \forall e \in Entities \setminus \{root\} \end{aligned}$$

Similarly, the term *descendants* identifies the set of entities that includes an entity's children and its children's descendants.

$$descendants(e) = children(e) \cup \bigcup_{e' \in children(e)} descendants(e') \quad \forall e \in Entities$$

These two definitions permit the assertion that the hierarchy forms a tree, where the root is an ancestor of all nodes (except the root itself):

$$\forall e \in Entities \setminus \{root\}, root \in ancestors(e)$$

Inversely, it would also be possible to define a similar constraint so that all entities (except root) are descendants of root:

$$descendants(root) = Entities \setminus \{root\}$$

The growing lamp example consists of one root entity that defines three sub-entities for lighting, heating and an adder. The functions thus return, for instance:

$$\begin{aligned} Entities &= \{GrowLamp, LightElement, HeatElement, Adder\} \\ root &= GrowLamp \\ children(GrowLamp) &= \{LightElement, HeatElement, Adder\} \\ ancestors(GrowLamp) &= \emptyset \\ descendants(GrowLamp) &= \{LightElement, HeatElement, Adder\} \end{aligned}$$

Definition 4 (Ports). CREST systems use *ports* for the transfer of resources and storage of values and information. These ports are defined by a set of port names *Ports*, and a function $type : Ports \rightarrow Types$ that assigns the resource type of each port.

In the example above, the port names of the growing lamp system are:

$$Ports = \{\text{electricity, switch, on-time, light, temperature, ...}\}$$

The types associated with these ports are:

$$\begin{aligned} \text{type}(\text{electricity}) &= \mathbb{R}\text{Watt} \\ \text{type}(\text{switch}) &= \{\text{on}, \text{off}\}\text{Switch} \\ \text{type}(\text{light}) &= \mathbb{N}\text{Lumen} \end{aligned}$$

The system's port names are partitioned into *inputs*, *outputs* and *local* ports: $\text{Ports} = \text{Ports}^I \sqcup \text{Ports}^L \sqcup \text{Ports}^O$. Each port is also assigned to exactly one entity:

$$\text{Ports} = \bigsqcup_{e \in \text{Entities}} \text{Ports}_e$$

The intersection of these partitions defines an entity's inputs, outputs and locals:

$$\forall e \in \text{Entities} \begin{cases} \text{Ports}_e^I &= \text{Ports}^I \cap \text{Ports}_e \\ \text{Ports}_e^O &= \text{Ports}^O \cap \text{Ports}_e \\ \text{Ports}_e^L &= \text{Ports}^L \cap \text{Ports}_e \end{cases}$$

To enforce the locality principle, CREST allows only a subset of ports to be used in transition guards and update functions. Thus, an entity can only read certain ports called *sources*. Further, an entity's updates can only write to specific *targets* ports.

The function $\text{sources} : \text{Entities} \rightarrow \mathcal{P}(\text{Ports})$ provides the ports of an entity, that can be used to calculate transition guards or the value of update functions. The set is composed of an entity's inputs, the entity's local ports and, to access data from subentities, the direct subentities' output ports.

$$\forall e \in \text{Entities}, \text{sources}(e) = \text{Ports}_e^I \cup \text{Ports}_e^L \cup \bigcup_{e' \in \text{children}(e)} \text{Ports}_{e'}^O$$

In the growing lamp we find for example

$$\begin{aligned} \text{sources}(\text{GrowLamp}) &= \{\text{electricity}, \text{switch}, \text{heatswitch}, \text{room-temperature} \\ &\quad \text{on-count}, \text{on-time}, \text{light}_L, \text{heat}_H, \text{temperature}_A\} \\ \text{sources}(\text{Adder}) &= \{\text{heat-in}, \text{temp-in}\} \end{aligned}$$

$\text{targets} : \text{Entities} \rightarrow \mathcal{P}(\text{Ports})$ is a function that returns the set of possible targets of update functions for an entity. Targets can be an entity's local ports, outputs and all direct subentities' input ports.

$$\forall e \in \text{Entities}, \text{targets}(e) = \text{Ports}_e^O \cup \text{Ports}_e^L \cup \bigcup_{e' \in \text{children}(e)} \text{Ports}_{e'}^I$$

Definition 5 (Bindings). During the execution of a CREST system, each port is associated with a value of its respective resource *type*. The mappings from ports to values are defined by the set $\text{Bindings} = \{b : \text{Ports} \rightarrow \text{Resources} \mid \forall p \in \text{Ports}, b(p) \in \text{type}(p)\}$. The initial port bindings for some ports in Figure 4.1 are for instance:

$$\begin{aligned} b(\text{electricity}) &= 0\text{Watt} & b(\text{switch}) &= \text{offSwitch} \\ b(\text{on-time}) &= 0\text{Time} & b(\text{light}) &= 0\text{Lumen} \end{aligned}$$

Definition 6 (States and Transitions). The behaviour of a CREST entity e is defined by an automaton consisting of $States_e$ and a global, guarded transition relation. The set of all states is partitioned into distinct subsets for each entity:

$$States = \bigsqcup_{e \in Entities} States_e$$

Each entity further has to have at least one state: $\forall e \in Entities, States_e \neq \emptyset$.

In Figure 4.1 we find the following states: $States = \{\text{On}, \text{Off}, \text{On}_L, \text{Off}_L, \text{Add}, \text{State}\}$. These states are split up for each individual entity as follows:

$$\begin{aligned} States_{\text{GrowLamp}} &= \{\text{On}, \text{Off}\} \\ States_{\text{LightElement}} &= \{\text{On}_L, \text{Off}_L\} \\ States_{\text{HeatElement}} &= \{\text{State}\} \\ States_{\text{Adder}} &= \{\text{Add}\} \end{aligned}$$

The *Transitions* relation associates a source state to a target state and a guard function name. CREST requires a transition's source and target states to be part of the same entity. \mathcal{T} is the set of all guard function names. The global set of transitions is accordingly defined by:

$$Transitions \subseteq \bigcup_{e \in Entities} (States_e \times States_e \times \mathcal{T})$$

In Figure 4.1 the following transitions are defined:

$$\begin{aligned} Transitions &= \{\langle \text{On}, \text{Off}, \text{off-guard} \rangle, \langle \text{Off}, \text{On}, \text{on-guard} \rangle, \dots\} \\ \mathcal{T} &= \{\text{on-guard}, \text{off-guard}, \text{on-guard}_L, \text{off-guard}_L\} \end{aligned}$$

The function $\tau : \mathcal{T} \rightarrow (Bindings \times Bindings \rightarrow \mathbb{B})$ maps the guard function names to guard function implementations. Guard implementations use a current port bindings $binding$ and a previous port bindings pre ($binding, pre \in Bindings$) to calculate a Boolean codomain value (True/False). This value states whether a transition is enabled. The guard function must only use the values of the entities' *sources* ports to compute its result. (See Section 4.1.5 – Semantic Constraints for details.)

The growing lamp example's $\tau(\text{on-guard})$ points to the following guard function implementation, for instance:

$$\tau(\text{on-guard})(binding, pre) = \begin{cases} \text{False} & \text{if } binding(\text{electricity}) < 100\text{Watt} \\ \text{True} & \text{if } binding(\text{electricity}) \geq 100\text{Watt} \end{cases}$$

Definition 7 (Updates). Updates are CREST's means to perform modification of port values. Each update specifies an automaton state and evaluates continuously while that state is active. Updates specify functions that calculate the values that are assigned to their target ports. These functions use the current and previous port bindings as domain, which makes it possible to modify a port value based on other ports' values. In the example above, we see that the *Adder* defines an update *add*.

add reads the values of *Adder*'s two input ports, sums them up and writes them to the output port.

Updates can also be used to model variable evolution over time, since update functions have access to $\delta t \in \mathbb{T}$, the amount of time that passed since they were last executed. Hence, this value can be used to model timing aspects and time-based changes. Formally, *Updates* associates states, ports and update function names \mathcal{U} :

$$Updates \subseteq \bigcup_{e \in Entities} (States_e \times targets(e) \times \mathcal{U})$$

Only one update definition is allowed for each combination of target port and state, to avoid write-conflicts when two updates try to write to the same port:

$$\forall p \in Ports, s \in States, |\{(s, p, u) \in Updates\}| \leq 1$$

The function $\nu : \mathcal{U} \rightarrow (Bindings \times Bindings \times \mathbb{T} \rightarrow Resources)$ maps the update names to their implementations. Applied to port bindings *bind*, the previous port bindings *pre* ($bind, pre \in Bindings$) and a passed time span $\delta t \in \mathbb{T}$, they provide a new value for the specified target port.

The execution of the update function (identified by ν) can only change ports that are *targets* of the entity which contains the associated state. Further, only *sources* ports are allowed for the calculation of the returned value (see Section 4.1.5 – Semantic Constraints). The growing lamp model defines the following updates:

$$\begin{aligned} Updates &= \{\langle \text{On}, \text{electricity}_L, \text{update_light_electricity} \rangle, \\ &\quad \langle \text{On}, \text{on-time}, \text{update_on_time} \rangle, \dots\} \\ \mathcal{U} &= \{\text{update_on_time}, \text{update_light_electricity}, \dots\} \end{aligned}$$

In some formalisms (e.g. DEVS) the outputs of a system or component are only updated after certain actions (e.g. after “internal” transitions in DEVS). Since CREST's update functions model the continuous propagation of values independent of time advances or state automaton changes, updates have to be executed after any modification to any entity port, automaton transition or time advance to assert a correct and coherent system state. The details of this process are described in CREST's semantics in Section 4.2.

The CREST diagram also displays a special kind of update functions: *influences*. Influences are static updates that connect two ports unconditionally, i.e. independent of an entity's automaton state and the time that passed. Since such influences are a purely syntactic addition (they can be expressed through a set of “normal” updates), they are not directly part of the structure and semantics of a CREST system. See Section 4.3.1 – Influences for the formal introduction of influences.

Definition 8 (dependencies). The function $dependencies : \mathcal{U} \rightarrow Ports$ returns a set of ports for each update function name. We add a constraint that an update's dependencies can only be source-ports of the update's entity.

$$\begin{aligned} \forall \langle s, p, u \rangle \in Updates, \forall e \in Entities, s \in States_e, p \in targets(e), \\ dependencies(u) \subseteq sources(e), p \notin dependencies(u) \end{aligned}$$

The dependencies function is used to determine the execution order of updates within the operational semantics.

Note, that CREST entities are not allowed to specify circular dependencies between ports. This means that if a dependency e.g. reads a port A and writes B, then there cannot be an update reading B and write A. Such behaviour has to be resolved using a port's previous values *pre* as explained in Section 4.2³

Resolving Circular Dependencies CREST's semantics (see Section 4.2), define the execution order of subentities, based on the information of the *dependencies* function. The locality principle defined by CREST treats subentities as *black boxes* that read their input ports and write values to their respective outputs. However, this black box view might mislead to the naive conclusion that all of a subentity's output ports depend on all its inputs.

When looking at Figure 4.3, it becomes evident that this assumption can easily produce false circular dependencies which cannot be simulated or verified in CREST. The displayed system is composed of a power source and an electric heating device, each modelled as separate subentity. The heater is controlled by a switch input, which decides how much electricity it needs to operate (e.g. zero in case of *off*, 1000 watt if it is *on*). Accordingly, this information is propagated to the power source using the *electricity_draw* influence. The power source provides the requested electricity to its power output, which is transferred to the heater's *electricity* input. The power supply can also switch to an error state, if *draw_p* exceeds a certain threshold. In this case no electricity is provided.

It is immediately visible that the heater's *electricity* input depends on the power supply's *output*, but the power supply's *draw_p* input in turn depends on the heater's *draw_H* output. The above assumption – all entity outputs depend on all entity inputs – results in a circularly dependent system where the heater depends on the power supply and the power supply depends on the heater. Such a system cannot be simulated. One solution to this issue would be to remove CREST's black box view of subentities, which prevents the analysis of subentity internals. This approach clearly violates the *locality* principle, which is one of our key requirements. Another possibility is to structurally change the subentities to avoid such dependencies (e.g. by splitting the heater). The problem is that the entity, which is supposed to model real-world components, is replaced by artificial components where neither maps to a real system part. It further creates additional subentities and thereby weakens the model's maintainability, since more entities have to be considered, replaced and maintained. Thus, we dismiss these approaches and require more information to be provided instead. CREST therefore uses a function that explicitly specifies such dependencies inside subentities.

Definition 9 (io-dependencies). *io-dependencies* : $Ports \rightarrow \mathcal{P}(Ports)$ is a function that specifies the dependencies of an output port on its input ports. Clearly, it is necessary to restrict these dependencies, so that any output port can only depend

³Other modelling languages and tools use time delays to resolve such situations in a similar fashion. The use of *pre* makes this choice explicit.

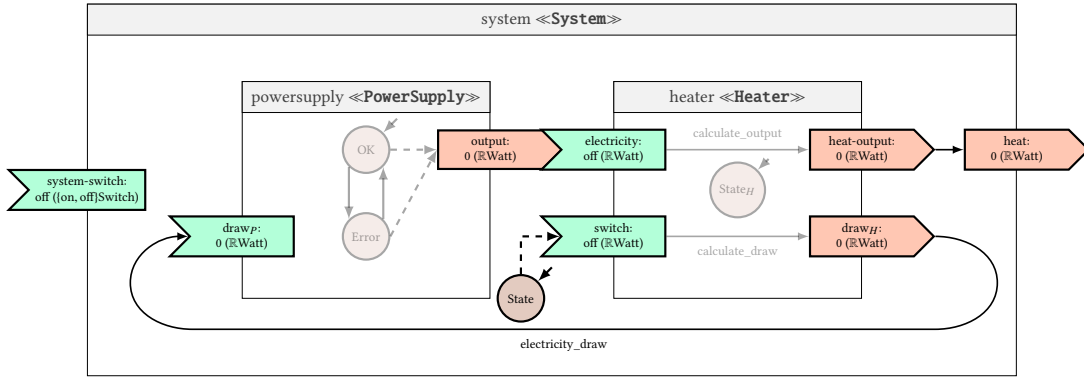


Figure 4.3 – Example of a circular dependency between subentities. The subentities’ internal structure and behaviour (drawn in semi-opaque) reveal however, that the circular dependency does not exist in the real system.

on a subset of input ports of the same entity. Thus, the function is implemented according to the following criterion:

$$\forall e \in Entities, \forall p \in Ports_e^O, io-dependencies(p) \subseteq Ports_e^I$$

In the example system in Figure 4.3, we can observe the following io-dependencies:

$$\begin{aligned} io-dependencies(draw_H) &= \{switch\} \\ io-dependencies(heat-output) &= \{electricity\} \\ io-dependencies(output) &= \{draw_P\} \\ io-dependencies(heat) &= \{system-switch\} \end{aligned}$$

In situations where the original, uninformed black box view is required, one could easily define that each of an entity e ’s outputs depends on all its inputs:

$$\forall p \in Ports_e^O, io-dependencies(p) = Ports_e^I$$

This assumption can be of useful when interacting with models that cannot be analysed by CREST, e.g. due to their complexity, their use of subroutines or proprietary libraries or are implemented e.g. as Functional Mock-up Units [Blo+12]. Evidently, the modelling of such “uninformed black boxes” might cause difficulties to the simulation and verification, due to semantic differences. The *io-dependencies* solution is thus only from a syntactic viewpoint. Developers are still required to analyse the component thoroughly and assert that the created model corresponds to the underlying system model (e.g. using a data-driven approach).

4.1.2 Global State of a CREST System

Definition 10 (State of the system). The global state $w \in W$ of an entire CREST system is a combination of the current states of all entity automata, the bindings of all ports, the previous bindings of the ports, and a global time.

$$W = Currents \times Bindings \times Bindings \times \mathbb{T}$$

Each CREST system further needs to define its initial state $w_0 \in W$.

The set of current automata states (not to be confused with the global system state) is given by $Currents = \{f : Entities \rightarrow States \mid \forall e \in Entities, f(e) \in States_e\}$. In the growing lamp example of Figure 4.1 $current \in Currents$ is initially defined as

$$\begin{aligned} current(\text{GrowLamp}) &= \text{Off} \\ current(\text{LightElement}) &= \text{Off}_L \\ current(\text{HeatElement}) &= \text{State} \\ current(\text{Adder}) &= \text{Add} \end{aligned}$$

The need for *pre* The *pre* binding stores all ports' previous value bindings. It can be used for various functionality in CREST systems where knowledge of the previous value is required. Without access to the ports' *pre* values, implementation of such behaviour would not be possible.

The most basic example for the use of *pre* is the implementation of an update function that does not alter a port's value. For example, an update $\langle s, p, u \rangle$'s function could be implemented as

$$v(u)(b, pre, \delta t) = pre(p)$$

Another update could calculate the change rate of another port p_2 's value using an implementation as follows:

$$v(u)(b, pre, \delta t) = \begin{cases} \frac{p_2 - pre(p_2)}{\delta t} & \text{if } \delta t \neq 0 \\ pre(p) & \text{otherwise} \end{cases}$$

Assuming that p is the port whose value is assigned by the update, the function calculates p_2 's linear change rate if δt is not zero, otherwise it leaves the value unchanged (i.e. writes its own *pre*-value).

4.1.3 CREST Syntactic Structure

Based on the previous definitions, a CREST system is specified as a structure S containing information about the resources (data types), entity hierarchy, ports, states and transitions, updates, dependencies, io-dependencies, and the initial global state:

$$S = \langle Units, Domains, Entities, parent, Ports, type, States, Transitions, \mathcal{T}, \tau, Updates, \mathcal{U}, v, dependencies, io-dependencies, w_0 \rangle$$

4.1.4 Changes to the System State

Definition 11 (Change of automata states). The state transition of an entity e to a state s is represented by $w[e \mapsto s]$. This change within one entity creates a new

(global) system state w' where the *current* automaton state of all entities remains the same, except for e (the entity to be updated), which now maps to s .

$$\begin{aligned} \forall w \in W, w &= \langle \text{current}, \text{bind}, \text{pre}, t \rangle, \\ \forall e \in \text{Entities}, \forall s \in \text{States}_e, w[e \mapsto s] &= \langle \text{current}', \text{bind}, \text{pre}, t \rangle \\ \text{where } \forall e' \in \text{Entities}, \text{current}'(e') &= \begin{cases} s & \text{if } e' = e \\ \text{current}(e') & \text{otherwise} \end{cases} \end{aligned}$$

Definition 12 (Change of port values). Changes to port bindings are denoted by $w[ps]$, where ps is a set of port-value mappings ($p \mapsto r$) such that there is at most one mapping for each p ⁴. We define the value assignment to be the creation of the global state where the bindings for all ports p appearing within ps are the new values and all ports not specified within ps remain unchanged.

$$\begin{aligned} \forall w \in W, w &= \langle \text{current}, \text{bind}, \text{pre}, t \rangle, \\ \forall ps \in \{f : P' \rightarrow \text{Resources} \mid P' \subseteq \text{Ports} \wedge f(p) \in \text{type}(p)\}, \\ w[ps] &= \langle \text{current}, \text{bind}', \text{pre}', t \rangle, \text{ where} \\ \forall p \in \text{Ports}, &\begin{cases} \text{bind}'(p) = r \wedge \text{pre}'(p) = \text{bind}(p) & \text{if } \exists p \mapsto r \in ps \\ \text{bind}'(p) = \text{bind}(p) \wedge \text{pre}'(p) = \text{pre}(p) & \text{otherwise} \end{cases} \end{aligned}$$

Note that the previous port values *pre* of the ports in ps have to be updated, so that efficient dataflow modelling is possible and value changes can be observed.

To modify the GrowLamp's inputs we could for example call

$$w[\{\text{electricity} \mapsto 500\text{Watt}, \text{switch} \mapsto \text{onSwitch}\}]$$

The definitions above specify the modification of individual automata and port values. The effects of such changes on a CREST system and the upkeep of a well-formed system state require more complex behavioural routines that are defined as CREST's semantics in the next section.

4.1.5 Semantic Constraints

As briefly outlined before, CREST systems have certain constraints on update functions and transition guards. These constraints limit the implementations of update functions and transition guards. As CREST neither prescribes the syntax nor semantics for the implementation of these functions, it is important that these constraints are upheld, so that the semantic correctness of the system can be preserved. The constraints, that are imposed on function implementations, are presented below.

⁴i.e. there cannot be a mapping $ps = \{p \mapsto r, p \mapsto s\}$

Transition Guard Locality This constraint states that the guard conditions can only use ports that are part of an entity e 's source ports $sources(e)$ for evaluation. The requirement formalisation below expresses this using the condition that the application of a guard function onto two bindings b_1 and b_2 produces the same result, when b_1 and b_2 are equal for all $sources$ ports.

$\forall e \in Entities, \forall \langle s, t, g \rangle \in Transitions, s, t \in States_e, \forall b_1, b_2, pre_1, pre_2 \in Bindings,$

$$\left(\begin{array}{l} \forall p_1 \in sources(e), \\ b_1(p_1) = b_2(p_1), pre_1(p_1) = pre_2(p_1) \end{array} \right) \implies \tau(g)(b_1, pre_1) = \tau(g)(b_2, pre_2)$$

This means, that transition guards also have to evaluate to the same result, even if there are non-*sources* ports whose bindings do not match, i.e.

$$\exists p_2 \in Ports, p_2 \notin sources(e), b_1(p_2) \neq b_2(p_2) \vee pre_1(p_2) \neq pre_2(p_2)$$

When looking at the growing lamp example of Figure 4.1, we observe that any transition defined for the GrowLamp entity can only read the following ports:

$$sources(\text{GrowLamp}) = \{\text{electricity, switch, heatswitch, room-temperature, on-count, on-time, light}_L, \text{heat}_H, \text{temperature}_A\}$$

Update Function Locality Similarly to the transition guards, update functions also can only use ports within an entity e 's source ports $sources(e)$.

The constraint below expresses that given any entity and update writing to a port p , the update's function implementation $v(u)$ has to produce the same result when applied onto a binding that is equal in the entity's $sources$ port bindings.

$\forall e \in Entities, \forall \langle s, p, u \rangle \in Updates, p \in targets(e), \forall b_1, b_2, pre_1, pre_2 \in Bindings, \forall \delta t \in \mathbb{T}$

$$\left(\begin{array}{l} \forall p_1 \in sources(e), \\ b_1(p_1) = b_2(p_1), pre_1(p_1) = pre_2(p_1) \end{array} \right) \implies v(u)(b_1, pre_1, \delta t) = v(u)(b_2, pre_2, \delta t)$$

This also implies that the update functions have to produce the same result, even if there exist non-*sources* ports whose value bindings differ, i.e. if

$$\exists p_2 \in Ports, p_2 \notin sources(e), b_1(p_2) \neq b_2(p_2) \vee pre_1(p_2) \neq pre_2(p_2)$$

Update Resource Type Finally, CREST requires that update functions always have to produce a value of the update's target's resource type.

$$\forall \langle s, p, u \rangle \in Updates, \forall \delta t \in \mathbb{T}, \forall b, pre \in Bindings, v(u)(b, pre, \delta t) \in type(p)$$

4.2 CREST Semantics

CREST’s semantics allow two basic forms of interaction with the system: setting the root entity’s input values and advancing the system time. After either of these is performed, the system might be in an *unstable* state. The term “unstable” refers to a system state, where, due to the interaction, a transition might have become enabled⁵ or an update’s target port value outdated. To correct this situation, the system must be *stabilised*. Stabilisation is hence the process of bringing a system into a state where all updates have been executed, and no transitions are enabled.

In the following, we describe the stabilisation process after changing port values and advancing time. To facilitate the understanding of the processes and their interconnections, Figure 4.4 presents the semantic processes and the individual steps taken as UML 2.5 activity diagrams [UML17]. The figures were collectively placed on a single page, so that the relations between diagrams can be followed more easily. For facilitated understanding of the semantics, the reader is advised to keep Figure 4.4 at hand while navigating through this section.

Setting Values The most basic form of interaction with a CREST system is the modification of its input port values. As stated, any external modification of input values requires a subsequent stabilisation phase to assert that the system is in a stable state. All value modifications have to be propagated to dependent ports through updates. The `ad:SetValues` activity diagram (Figure 4.4a) visualises this process. For an example, we can imagine a modification of the `electricity` value in the growing lamp model. This value change has to be propagated to the corresponding inputs of the lighting and heating components’ input ports. These modules will in turn modify their respective internal behaviour and update output port values, which will trigger further propagation of the values.

The stabilisation process (`ad:Stabilise`, Figure 4.4c) outlines the tasks performed to assert correct value propagation and subsequent triggering of enabled transitions. As even a minimal change of one value can have a significant impact on the entire system’s behaviour, the execution of updates and stabilisation of subentities has to be performed in the correct order. For instance, stabilising the `GrowLamp` entity involves the orchestrated triggering of the stabilisation process in each subentity. From the (partial) CREST diagram in Figure 4.5, we see that the growing lamp’s `temperature` output is modified by `forw_temp`. Since `forw_temp` accesses the output of the `Adder` entity, it is necessary that this subentity is stabilised before `forw_temp` is executed. However, stabilisation of `Adder` requires that its inputs have been set correctly, etc⁶. We see, that a complex network of dependencies must be established and considered so that the system’s outputs are correctly calculated.

In general, the simulation of a system requires that the *modifiers* (updates and

⁵CREST uses “must”-semantics, which means that a transition must be triggered when it is enabled.

⁶Note that the `GrowLamp` uses influences for the propagation of port values. Since influences can be translated into update functions though, the underlying ordering concept is the same. See Section 4.3.

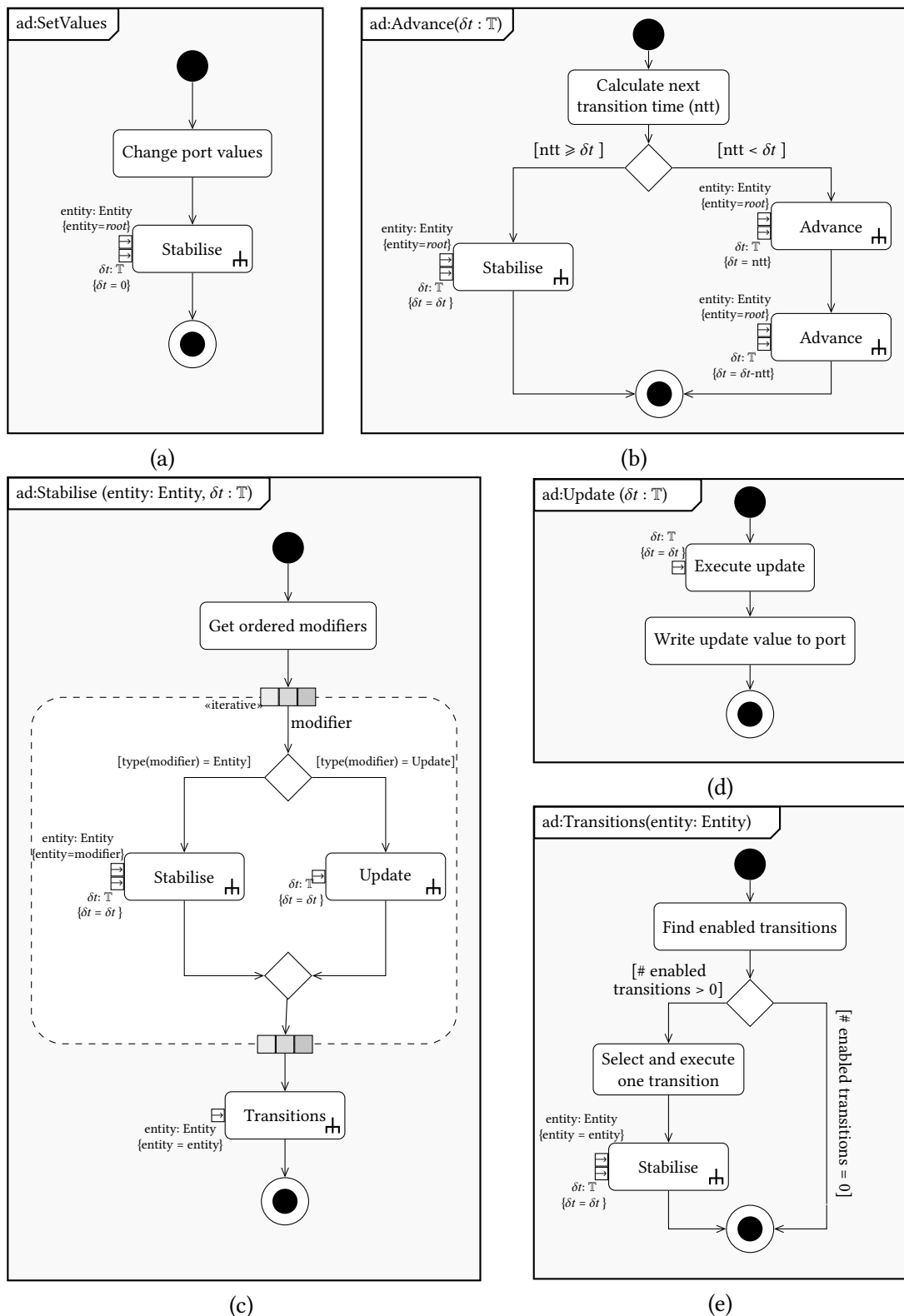


Figure 4.4 – Informal, schematic representation of the semantics as UML activity diagrams. Activities represent semantic processes and arrows indicate triggering of other processes. Note that not all parameters (UML object pins) that are passed between activities are displayed.

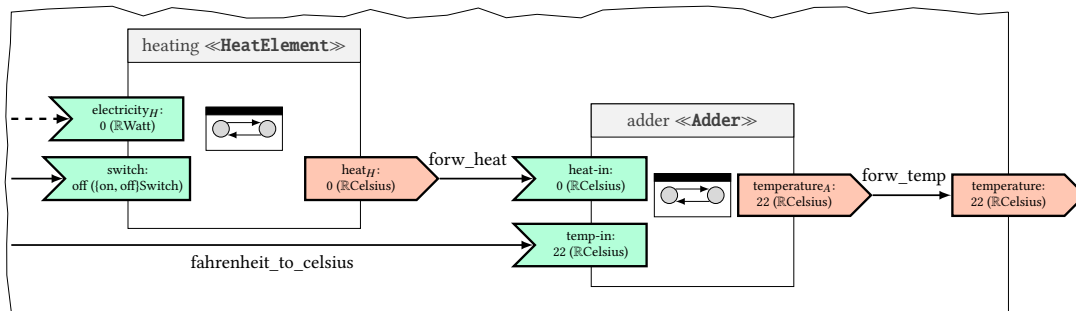


Figure 4.5 – Excerpt of the growing lamp CREST diagram highlighting dependencies between outputs and subentities. Note that some parts of the GrowLamp have been omitted and subentity behaviour has been masked.

subentities⁷) which read a certain port have to be executed after the modifiers that write to that port. The creation of this execution order is closely related to the concepts of dataflow modelling and Kahn process networks [Kah74]. Synchronous languages (e.g. Lustre and Esterel), similarly rely on the appropriate ordering of their operator nodes for a correct behaviour simulation.

Due to the required total order of modifiers, CREST systems must not define circular dependencies. If there are interdependencies between values, they cannot all reference their current values. Instead, the circular dependencies have to be broken using a port's *pre* value. This means that at least one of the dependencies is on a port's previous value, instead of it's current, thereby stopping the cycle. The solution is inspired by other languages, e.g. Lustre's *pre* operator, Simulink's *Unit Delay* blocks and Modelica's *pre* instruction.

Once the modifiers have been arranged, they are triggered in this order. The specific execution is based on the modifier's type. If the modifier is an update, the specified function is executed and the value written to the target port (`ad:Update`, Figure 4.4d). In case it is a subentity, the stabilisation is recursively performed inside that subentity, before continuing with the rest of the modifier list. In the activity diagram, the choice of the correct modifier treatment is visualised in the UML *expansion region*, which iterates over the ordered modifiers and selects the correct action based on the modifier's type.

If after the execution of all modifiers any FSM transitions are enabled, one of them is chosen to be executed (`ad:Transitions`, Figure 4.4e). CREST does not prescribe a selection procedure in case multiple transitions are enabled, meaning that *non-determinism* may occur. If a transition was enabled and executed, another stabilisation is started to execute the updates that are related to the new FSM state. In the end, this stabilisation will again look for enabled transitions, potentially trigger one and stabilise once more, until no enabled transitions are found.

The stabilisation process operates recursively. This means that if an entity trig-

⁷When subentities are treated as black boxes, they can be seen as an abstract form of update functions that read their input ports and write to their outputs.

gers a subentity stabilisation, the subentity’s modifiers are executed in the correct order and the enabled transitions within that subentity are triggered (followed by stabilisations) until no transitions are enabled. Only then, the control is returned back to the parent entity to continue. As a result of this *locality*, modifiers of the same entity could be safely executed in parallel, provided that they are independent, i.e. if they do not have dependencies between their inputs and outputs. For simplicity purposes, this feature will not be considered by the semantics described in this section, but can be considered in a tool implementation for performance reasons⁸.

It is important to remember that CREST operates synchronously and no time passes between the update of port values and the end of the stabilisation process, whereas some other languages (e.g. Simulink) introduce a small time delay at every modification. CREST’s *synchronism* concept can also be found in languages such as Esterel. CREST’s semantics differ from Esterel’s, however, in that CREST always stabilises the entire system instead of just the affected subset.

Hierarchical Encapsulation CREST’s system view, where an entity is responsible for the execution of modifiers in the correct order supports the locality principle. However, this local encapsulation of data and control, in combination with a hierarchical system view is also well-known in the modelling and simulation community. In last decade, this design pattern was used for co-simulation [Gom+17] and in standards such as FMI [Blo+12]. FMI imposes the use of explicit *orchestrators*. These are components trigger computation in subcomponents and are responsible for calculating the ideal step size of time advances. They are also in charge of relaying the communication between the individual system components so that the required data is available when and where it is needed. It is obvious that this pattern is related to the way CREST implements the stabilisation of system states. Each CREST entity can be seen as the orchestrator of its direct subentities. Similar to FMI orchestrators, a CREST entity is charged with triggering the propagation of values and asserts correct execution order so that the system’s state remains valid.

Advancing Time The prior part of this section states that updates allow the modification of a system over time using a δt parameter. In fact, the semantics of time advances are based on the same stabilisation process as `SetValues`, except that `SetValues` triggers stabilisation with $\delta t = 0$, while `Advance` specifies a $\delta t > 0$.

There is one particularity of time advances that has to be considered though: CREST implements eager transition triggering (“must” semantics). This means that a transition has to be executed as soon as it becomes enabled. Hence, CREST can only advance to the next point in time when a transition becomes enabled, before stabilisation is needed.

When trying to advance time further than that, CREST needs to find the next transition time first, advance to it, stabilise and then advance the remaining time. CREST’s semantics adheres to a *continuous time* concept, that does not require “ticks”

⁸A discussion of the parallel execution of modifiers is presented in Section 4.4.

as synchronisation points at which transition guards are evaluated. In order not to “miss” the exact moment when a transition becomes enabled, CREST makes use of a function that calculates the precise duration ntt that has to pass until any transition will be enabled. ntt is in the range $[0, \dots, \infty]$, where 0 states that a transition is currently enabled (and that the system is not stable), and ∞ means that no transition can become enabled by just advancing time. Note that the implementation of a next-transition-time function is non-trivial and depends on the concrete implementation of updates and guards. It’s functionality involves complex tasks such as the creation of inverse functions or the expression of the functionality as sets of constraints.

As depicted in the activity diagram (ad:Advance, Figure 4.4b), the information of the next transition time ntt creates two possible scenarios:

1. $ntt \geq \delta t$: This means that the next transition time is further away than (or at least as far away as) the time we plan to advance δt . CREST advances δt and calls the stabilisation action to execute updates and transitions until the system reaches a stable state.
2. $ntt < \delta t$. In this case, we plan to advance past the point where a transition becomes enabled. CREST divides such an advance into two steps: First a recursive call to $Advance(ntt)$ advances time to the point where a transition becomes enabled (i.e. triggering scenario 1 above). All updates and transitions are triggered, followed by stabilisation until the system is stable. Next, CREST recurses on the remainder of the time, by triggering $Advance(\delta t - ntt)$. Depending on the next transition time of the new system state and the remaining time to advance, CREST will again trigger one of these two scenarios.

CREST’s time semantics allow the simulation and verification based on real-valued clocks with arbitrary time advances. This is essential for the precise simulation of CPSs without the need for an artificial base-clock. The continuous time-based enabling of transitions extends the language and adds a *continuous behaviour* to the otherwise purely reactive system.

4.2.1 Modifiers and Precedence – Formalisation

Before delving into the actual semantics provided with SOS rules, we first have to formalise vital concepts, such as the modifier ordering and the discovery of enabled transitions, as discussed above.

To establish the modifier precedence, the formal semantics make use of the precedence operator $<$. This operator expresses an order between ports, updates and child entities that arises from the updates’ dependencies. Further, a function *active-modifiers* identifies the set of updates and subentities that have to be executed to stabilise a CREST system and to recursively propagate time advances throughout an entity. Due to potential interdependencies between update functions, i.e. one update might read a port which is written by another update, it is necessary to execute the stabilisation in a specific order. This subsection defines functions and operators that are used in the semantic rules for this stabilisation process.

Port Precedence The $<$ operator defines a partial order between ports, based on the *dependencies*-function (introduced in Definition 8), and the input-output dependency function *io-dependencies* (Definition 9). We say that for any two ports $p_1, p_2 \in Ports$ $p_1 < p_2$ iff one of the following cases applies:

1. there exists an update whose target is p_2 and p_1 is a dependency of that update (i.e. the update function reads the binding of p_1 to calculate the value of p_2);
2. there exists an entity, and p_1 is an input, p_2 is an output and there exists an io-dependency between the two;
3. there exists a port p' so that $p_1 < p'$ and $p' < p_2$ (i.e. $p_1 < p_2$ by transitivity).

Formally this operator is expressed as follows:

$$\forall p_1, p_2 \in Ports, p_1 < p_2 \text{ iff } \begin{cases} \exists \langle s, p_2, u \rangle \in Updates, p_1 \in dependencies(u) \\ p_1 \in io-dependencies(p_2) \\ \exists p' \in Ports, p_1 < p' \wedge p' < p_2 \end{cases}$$

The partial order operator also satisfies the anti-symmetry constraint which requires that $\forall p_1 \in Ports, \nexists p_2 \in Ports, p_1 < p_2 \wedge p_2 < p_1$. Hence, circular dependencies between ports, subentities and updates are impossible. This constraint is further important so that at runtime all subentities and updates respect the notion of order.

active-modifiers Modifiers are those parts of the system that have the capability of altering a port's value. From an entity's point of view, they are either updates, defined to change the entity's ports, or subentities, because they alter the subentity output ports (which can also be read by the entity). To facilitate the subsequent definitions, we define the set of all modifiers to be the union of all entities and updates.

$$Modifiers = Entities \cup Updates$$

When performing changes within CREST entities, it is important to execute all "active" updates (the ones linked to an entity's current automaton state) and to propagate the action "down-stream" towards the entity's children. The function *active-modifiers* returns all such updates and child entities as a set.

$$\begin{aligned} active-modifiers &: W \times Entities \rightarrow \mathcal{P}(Modifiers) \\ active-modifiers(\langle current, bind, pre, time \rangle, e) &= \\ &\{ \langle s, p, u \rangle \in Update \mid s = current(e) \} \cup children(e) \end{aligned}$$

The list below shows the active modifiers of the *GrowLamp* entity in Figure 4.1. Note that some updates of the list are modelled as influences in the growing lamp. As influences are a special kind updates, they are indirectly also considered as modifiers. See Section 4.3 for details on the syntax and semantics of influences.

$$\begin{aligned}
\text{active-modifiers}(w_0, \text{GrowLamp}) = \{ \\
& \text{lighting}, \quad \text{adder}, \quad \text{heating}, \\
& \langle \text{Off}, \text{electricity}_L, \text{light_electricity_zero} \rangle, \quad \langle \text{Off}, \text{electricity}_H, \text{heat_electricity_zero} \rangle, \\
& \langle \text{Off}, \text{light}, \text{forward_light} \rangle, \quad \langle \text{Off}, \text{heat-in}, \text{forw_heat} \rangle, \\
& \langle \text{Off}, \text{temperature}, \text{forw_temp} \rangle \quad \langle \text{Off}, \text{temp-in}, \text{fahrenheit_to_celsius} \rangle \\
& \}
\end{aligned}$$

The function *modified-ports* returns the list of ports that are actively being written in an entity. This means, it consists of all ports that are the targets of the update functions of the current automaton state or outputs of a subentity.

$$\begin{aligned}
& \text{modified-ports} : W \times \text{Entities} \rightarrow \text{Ports} \\
& \text{modified-ports}(\langle \text{current}, \text{bind}, \text{pre}, \text{time} \rangle, e) = \\
& \quad \{p \mid \forall \langle s, p, u \rangle \in \text{Updates}, s = \text{current}(e)\} \cup \{p \mid \forall e' \in \text{children}(e), p \in \text{Ports}_e^O\}
\end{aligned}$$

For example, the modified ports of the GrowLamp in Figure 4.1 are:

$$\begin{aligned}
\text{modified-ports}(w_0, \text{GrowLamp}) = \{ \\
& \text{light}_L, \text{temperature}_A, \text{heat}_H, \text{electricity}_L, \text{electricity}_H, \\
& \text{light}, \quad \text{heat-in}, \text{temperature}, \text{temp-in} \\
& \}
\end{aligned}$$

ordered-ports The function *ordered-ports* creates a total order of ports that are modified according to their precedence.

$$\begin{aligned}
& \text{ordered-ports} : W \times \text{Entities} \rightarrow \text{PortLists} \\
& \text{ordered-ports}(w, e) : [p_0, p_1, \dots, p_n] \text{ s. t. } \forall p_i, p_j, i < j, p_i < p_j
\end{aligned}$$

The list of ports is defined by the *PortLists* type:

$$\text{PortLists} = \begin{cases} \emptyset \\ \langle \text{Ports}, \text{PortLists} \rangle \end{cases}$$

For clarity reasons, the common notation $[p_0, p_1, p_2, p_3, p_4]$ is used to denote a list of modifiers instead of the formal notation $\langle p_0, \langle p_1, \langle p_2, \langle p_3, \langle p_4, \emptyset \rangle \rangle \rangle \rangle \rangle$, where $i \in \mathbb{N}$ is a port's list index.

Further, an operator “:” can be used to separate a list's *head* (its first element) from its *tail* (the rest of the list) as follows: $[p_0 : \text{tail}]$ such that p_0 is the first element of the list and $\text{tail} = [p_1, p_2, \dots, p_n]$ is the rest.

Note that CREST does not prescribe an algorithm for creating this total ordering of ports, but instead only provides the constraints above to assert that any CREST

port is updated after the ports it depends on. For unrelated ports, i.e. ports where neither one specifies a precedence over the other, the order of modification is irrelevant. Without provision of a formal proof, one can easily see that order relevance is only given in situations of port dependencies. Thus, in these situations there must be a modifier (or a sequence of modifications) that creates a dependence from one port to another, and hence a precedence order would have to be defined.

ordered-modifiers The list of ordered ports is used for the creation of a list of modifiers (updates and subentities) that specifies their correct execution order, so that all dependencies are satisfied and no port value is assigned a wrong or outdated value. The type *ModifierLists* is used to describe such lists. Its formal signature is

$$\text{ModifierLists} = \begin{cases} \emptyset \\ \langle \text{Modifiers}, \text{ModifierLists} \rangle \end{cases}$$

We define the notational form $[m_0, m_1, m_2, m_3, m_4]$ and the “head-tail” operator “:” for lists of modifiers, similar to the *PortLists* type above.

The function *ordered-modifiers* returns a list of modifiers so that for each port in the ordered-ports list there is a modifier that updates it.

$$\text{ordered-modifiers} : W \times \text{Entities} \rightarrow \text{ModifierLists}$$

$\text{ordered-modifiers}(w, e) : [m_0, m_1, \dots, m_n, m_{n+1}, \dots, m_{n+k}]$ such that for each port p_i , there is an entry in the modifier-list with the same index that alters this port’s value

$$\forall p_i \in \text{ordered-ports}(w, e), \exists m_i \in \text{active-modifiers}(w, e) \wedge \begin{cases} m_i = \langle s, p_i, u \rangle \in \text{Updates} \\ m_i = e' \in \text{children}(e), p_i \in \text{Ports}_e^O \end{cases}$$

and that all additional active modifiers are appended at the end of the list.

$$\forall m_{n+j} \in \text{active-modifiers}(w, e), 0 < j \leq k, \nexists p \in \text{ordered-ports}(w, e), \\ m_{n+j} = \langle s, p, u \rangle \in \text{Updates} \wedge m_{n+j} = e' \in \text{children}(e), p \in \text{Ports}_e^O$$

Note that *ordered-modifiers* is based on the (non-deterministically defined) *ordered-ports* function. This means, that *ordered-modifiers* provides one of possibly many orderings, whose exact list of modifiers is not precisely specified but constrained to assert a correct system evolution.

Modifiers without Precedence Information Additional active modifiers are such modifiers for which no precedence information is available. Specifically, they are subentities whose inputs are not written by any update or which do not provide output ports. Two examples of such subentities are shown in Figure 4.6. Assuming that the update *up* does not read any ports, there is no dependency and hence no precedence. *no_input* also does not specify any dependency, as there is only one output port and no input ports. Hence, from the system entity’s point of view, there are no dependencies defined.

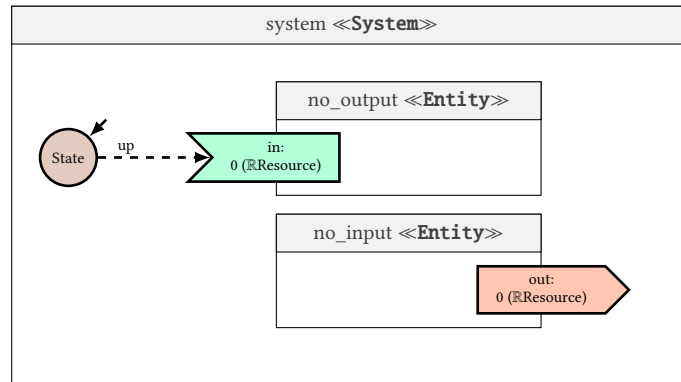


Figure 4.6 – Example of a subentities without precedence specification.

Even though the effects of such subentities are not observable (e.g. due to lack of output ports or updates that read them), it is important that they are still executed to preserve the system’s complete correctness. However, the execution order of such modifiers does not impact the further behaviour of the system.

enabled-transitions Each CREST entity defines guarded transitions between its automata states. To discover all currently enabled transitions of an entity, the function *enabled-transitions* is used. Given a current state and an entity, the function returns the set of transitions whose guard functions evaluate to **True**.

enabled-transitions : $W \times Entities \rightarrow \mathcal{P}(Transitions)$

enabled-transitions($\langle curr, bind, pre, time \rangle, e$) =

$$\{ \langle s, t, g \rangle \in Transitions \mid s, t \in States_e \wedge s = curr(e) \wedge \tau(g)(bind, pre) \mapsto \mathbf{True} \}$$

4.2.2 Formal Operational Semantics

This section takes a closer look at the formal aspect of CREST’s operational semantics. CREST’s semantics describe modifications of the global system state ($w \in W$). All system modification information has to be propagated from the *root* entity towards the leaves of the CREST model’s entity tree. The locality principle further demands that an entity is responsible for the upkeep of its own state and can only trigger stabilisation of its direct subentities. Thus, an entity does not directly influence the stabilisation process of its ancestor entities. This behaviour is important for the locality and black box view of entities, as it supports the clean composition and decomposition of entities.

The global semantics of a CREST system are based on the concept of reaching a fixed point (“fixpoint”) after each system modification. Fixpoints are states in which the system is stable, i.e. the system does not change unless time passes or external factors modify the system inputs.

set-values The fixpoint concept is applied for example when modifying the value of ports, as shown in Rule 4.1 below. The change of the port value bindings of a CREST system’s global state $w \in W$ is defined according to the *set-values* SOS rule: $\xrightarrow{\text{set-values}} \subseteq W \times \{Ports \rightarrow Resources\} \times W$. The relation operates on a system state w and updates it according to a port-value mapping $vs = \{Ports \rightarrow Resources\}$. This operation requires the subsequent application of the *stabilise* rule on the system’s *root* after the modification ($\xrightarrow{\text{stabilise}} \subseteq W \times Entities \times \mathbb{T} \times W$). The routine triggers an entity’s updates and automaton transitions until a fixpoint (stable state) is reached. In the process it also recursively propagates the modifications to its child entities.

$$\frac{w_1 = w[vs], \quad \langle w_1, root, 0 \rangle \xrightarrow{\text{stabilise}} w_2}{\langle w, vs \rangle \xrightarrow{\text{set-values}} w_2} \quad (4.1)$$

stabilise As previously introduced, the propagation of system changes (port value alteration or time advance) is based on the concept that after the modification, all updates of an entity should be executed, and stabilisation should be triggered in child entities. Rule 4.2 is called on a specific entity e whose stabilisation is to be triggered, and a timestep size δt that should be executed. For stabilisation of the system without time advance, such as after the setting of port values (see *set-values* above), this rule is initially called on the *root* entity with $\delta t = 0$, which will propagate values but not execute any time advance of the system.

In detail, the rule first obtains the ordered list of modifiers and triggers their execution using the *apply-all* rule ($\xrightarrow{\text{apply-all}} \subseteq W \times ModifierLists \times \mathbb{T} \times W$). This is followed by *transitions* ($\xrightarrow{\text{transitions}} \subseteq W \times Entities \times W$), which executes the state automaton’s transitions until no further ones are enabled.

$$\frac{\begin{array}{c} mods = ordered-modifiers(w, e), \quad \langle set-pre(w, e), e, mods, \delta t \rangle \xrightarrow{\text{apply-all}} w_1 \\ \langle w_1, e \rangle \xrightarrow{\text{transitions}} w_2 \end{array}}{\langle w, e, \delta t \rangle \xrightarrow{\text{stabilise}} w_2} \quad (4.2)$$

The above rule uses a function *set-pre*, that, given $w = \langle curr, bind, pre, time \rangle$ (the global system state) and an entity e , creates a new state, where the ports’ previous value binding *pre* is updated. It returns a global system state whose *pre* bindings are set to *bind* for all ports in *targets(e)*. This function is used before executing any modifiers of this entity, to assert that the modifiers inside the entity have access to the port’s previous values (i.e. the values before the updates). Note that *set-pre* only modifies the *pre*-bindings of an entity’s target ports, to maintain consistency with the

rest of the formalisation. This means that each entity is also responsible for setting the *pre* values of its subentities' inputs.

$$\begin{aligned} \text{set-pre}(\langle \text{curr}, \text{bind}, \text{pre}, \text{time} \rangle, e) &= \langle \text{curr}, \text{bind}, \text{pre}', \text{time} \rangle \\ \text{where } \text{pre}'(p) &= \begin{cases} \text{bind}(p) & \text{if } p \in \text{targets}(e) \\ \text{pre}(p) & \text{otherwise} \end{cases} \end{aligned}$$

apply-all This rule is responsible for identifying the first modifier m_0 within the ordered modifier list, and then executing it. After the execution of the first modifier m_0 (see **apply-one** below), the rule recursively executes on the rest of the modifier list mods . Using this approach, CREST propagates value changes iteratively, taking the dependencies between ports into account.

Rule 4.4 is the break-condition of the recursion. It is called when the list of modifiers is the empty list \emptyset and all applicable modifiers have been executed and removed. In this case, no action is taken and the system state remains unchanged.

$$\frac{\langle w, m_0, \delta t \rangle \xrightarrow{\text{apply-one}} w_1, \quad \langle w_1, \text{mods}, \delta t \rangle \xrightarrow{\text{apply-all}} w_2}{\langle w, [m_0 : \text{mods}], \delta t \rangle \xrightarrow{\text{apply-all}} w_2} \quad (4.3)$$

$$\frac{}{\langle w, \emptyset, \delta t \rangle \xrightarrow{\text{apply-all}} w} \quad (4.4)$$

apply-one ($\xrightarrow{\text{apply-one}} \subseteq W \times \text{Modifiers} \times \mathbb{T} \times W$) The two **apply-one** functions (Rules 4.5, 4.6) execute the modifier (update or subentity) based on their type. If the modifier is an update, the execution of the update is handed over to the specific update rule ($\xrightarrow{\text{update}} \subseteq W \times \text{Updates} \times \mathbb{T} \times W$). Otherwise (if it is a child entity), it calls **stabilise** on the child entity to propagate the changed system state and triggering updates within the children.

$$\frac{\text{mod} \in \text{Update}, \quad \langle w, \text{mod}, \delta t \rangle \xrightarrow{\text{update}} w_1}{\langle w, \text{mod}, \delta t \rangle \xrightarrow{\text{apply-one}} w_1} \quad (4.5)$$

$$\frac{\text{mod} \in \text{Entities}, \quad \langle w, \text{mod}, \delta t \rangle \xrightarrow{\text{stabilise}} w_1}{\langle w, \text{mod}, \delta t \rangle \xrightarrow{\text{apply-one}} w_1} \quad (4.6)$$

It is important to understand that the similar treatment of updates and child entities is a significant feature of CREST. CREST sees a child entity as a complex form of update which reads input values and writes output values. This *black-box* concept encapsulates all child behaviour and allows every CREST entity to always rely on the fact that its children are in a stable state.

update The execution of an update is relatively simple. The premises of Rule 4.7 extract the update function's name u , current port bindings $bind$ and previous port bindings pre . The new system state created by this rule is the old state where the update's target port p is set to the value returned by the update function implementation $v(u)$ executed with the parameters $bind$, pre and δt .

$$\frac{\langle s, p, u \rangle = mod, \quad w = \langle curr, bind, pre, time \rangle,}{\langle w, mod, \delta t \rangle \xrightarrow{\text{update}} w[p \mapsto v(u)(bind, pre, \delta t)]} \quad (4.7)$$

Effects of Continuous Time Update functions have access to δt , the time that has passed since the current state was activated. This allows the modification of port values over time and thereby potentially enabling transitions. Figure 4.7 shows such an enabling over time. It displays a water tank system that contains a pump. When the pump is turned on, the volume of water is calculated by an update function which continuously evaluates as follows: $volume = pre(volume) + in \cdot \delta t - out \cdot \delta t$, where in is the amount of water being pumped into the tank per time unit and out the water leaving the tank. The pump transitions to Off when the water volume exceeds 75. In Off the update subtracts $out \cdot \delta t$ from the $volume$. The pump starts as soon as the volume drops below 25. Assuming reasonable in and out flows – the pump will alternate between the On and Off states. Note that for simplicity reasons in this example we chose to annotate the updates and transitions with their implementations, rather than their names. While this “shortcut” allows for more expressive diagrams, it is important to understand that these annotations are mathematical syntax, and not part of CREST.

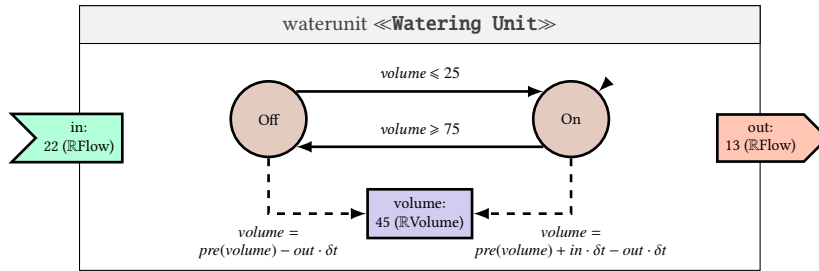


Figure 4.7 – A water tank that can alternate between filling and emptying.

transitions The transitions rules are responsible for triggering transitions and deciding whether further stabilisation is required. If there are transitions enabled (Rule 4.8), the system state is modified by $w[e \mapsto t]$ and the semantics recurse by triggering the `stabilise` rule. In case no transition was enabled (and thus none was executed), no further action is taken (Rule 4.9).

$$\frac{\langle s, t, g \rangle \in \text{enabled-transitions}(w, e), \quad w_1 = w[e \mapsto t], \quad \langle w_1, e, 0 \rangle \xrightarrow{\text{stabilise}} w_2}{\langle w, e \rangle \xrightarrow{\text{transitions}} w_2} \quad (4.8)$$

$$\frac{\text{enabled-transitions}(w, e) = \emptyset}{\langle w, e \rangle \xrightarrow{\text{transitions}} w} \quad (4.9)$$

CREST implements eager transition evaluation. This means that a transition must be fired if (at least) one is enabled. It is essential that update functions are triggered immediately after the transition phase (using *stabilise*), as they otherwise risk being executed at the wrong moments or not at all. This behaviour also forms the basis for CREST’s implementation of transitions actions (See Section 4.3).

Non-Determinism It is also noteworthy that CREST does not prescribe a strategy in the event where more than one transition is enabled. Thus, this is the place where non-determinism is possible, e.g. if guard conditions “overlap”. Figure 4.8 for example displays the state automaton of a non-deterministic entity. The system is assumed to be a watering unit which is connected to two plants. If a plant’s soil is dry, the watering unit automatically waters that plant (**Water Plant X**). We can easily imagine a scenario where both plants are dry at the same time, for instance just after starting the system. Since CREST does not dictate any strategy, the decision which plant to water first is non-deterministic. This non-determinism is an important and required to model many software systems.

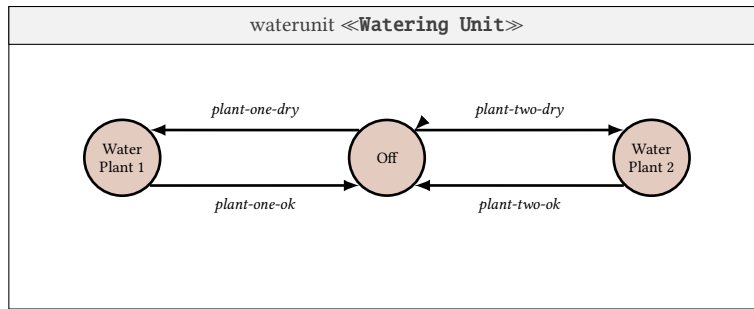


Figure 4.8 – Example of a non-deterministic state automaton. In case both plants are dry at the same time (and hence both transitions are enabled), the CREST implementation will choose one of the transitions.

Time Advance We see from the above rules that updates allow the modification of a system over time. The semantics of these time advances are defined below by the relation $\text{advance} (\xrightarrow{\text{advance}} \subseteq W \times \mathbb{T} \times W)$. The decision of which specific rule to apply depends on the amount of time to advance $\delta t \in \mathbb{T}^9$. CREST only supports positive δt -values, meaning that it is not possible to “step back in time”. In the semantics, triggering advance with a $\delta t < 0$ has no effect, as shown in Rule 4.10.

$$\frac{\delta t < 0}{\langle w, \delta t \rangle \xrightarrow{\text{advance}} w} \quad (4.10)$$

⁹See Appendix B for details of CREST’s time base.

An advance of $\delta t = 0$ triggers a system stabilisation. Since the stabilisation process searches for fixpoints, this means that the state w is only modified iff the system was not stable.

$$\frac{\delta t = 0, \langle w, root, 0 \rangle \xrightarrow{\text{stabilise}} w'}{\langle w, \delta t \rangle \xrightarrow{\text{advance}} w'} \quad (4.11)$$

The advance of time requires the availability of $next_transition_time : W \rightarrow \mathbb{T}$. Given a CREST system's current state w this function tries to calculate the precise amount of time δt that has to pass until updates enable any transition's guard condition. It returns ∞ in case it is not possible. CREST's semantics do not prescribe an implementation of this function. Depending on the available computation resources, many ways to implementing $next_transition_time$ are possible, such as a search approach that naively tries to find values until it is "close enough", or more complex analysis techniques that include symbolic reasoning on the system. CREST's implementation for example transpiles the transition guards and update functions' source codes into constraints, that are solved using an SMT prover, to find a minimum next transition time. A more detailed discussion of this implementation of $next_transition_time$ is provided in Chapter 5 – CREST Implementation.

Time advances with $\delta t > 0$ can be split into two possible scenarios, depending on the amount of time to advance (δt) and the time ntt until a transition triggers a discrete behaviour change. A visual aid is presented in Figure 4.9.

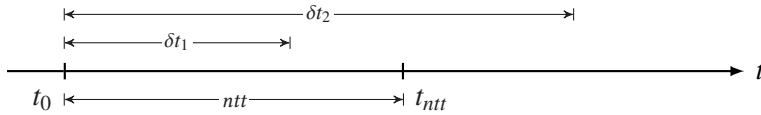


Figure 4.9 – Distinction of the two advance scenarios. Depending on δt and the point where the next transition happens (i.e. $t_0 + ntt = t_{ntt}$), different actions are taken. If $\delta t < ntt$ (e.g. δt_1) then scenario 1 is chosen, otherwise scenario 2 is executed (e.g. δt_2).

1. If the time we plan to advance (δt) is less than or equal to the next transition time (ntt), Rule 4.12 applies. Since no transition can become enabled before the δt timestep, CREST can safely advance to that point. Before the advance, the rule sets the *pre* values of the current binding.

Subsequently `stabilise` is called on the `root` entity with parameter δt , to trigger all system updates and stabilisation of entities (including potential transition triggering). Lastly (in the rule's conclusion) the system's global time is updated.

$$\frac{\delta t \leq next_transition_time(w), \langle set_pre(w, e), root, \delta t \rangle \xrightarrow{\text{stabilise}} \langle curr, bind, pre, time \rangle}{\langle w, \delta t \rangle \xrightarrow{\text{advance}} \langle curr, bind, pre, time + \delta t \rangle} \quad (4.12)$$

2. If δt is bigger than the next transition time ntt , Rule 4.13 will split the advance into two steps: First, it will trigger *advance* with the value ntt , which activates Rule 4.12, thereby triggering a stabilisation (including transition firing). Next, CREST recursively advances the remaining time (i.e. $\delta t - ntt$).

$$\frac{\delta t > ntt, \quad ntt = \text{next_transition_time}(w), \quad \langle w, ntt \rangle \xrightarrow{\text{advance}} w_1, \quad \langle w_1, \delta t - ntt \rangle \xrightarrow{\text{advance}} w_2}{\langle w, \delta t \rangle \xrightarrow{\text{advance}} w_2} \quad (4.13)$$

CREST's time semantics allow for the simulation and verification based on real-valued clocks with arbitrarily small time advances. This feature is essential for the precise simulation of CPSs and avoids the need for an artificial base-clock.

4.3 Language Extensions

The avid reader will have noticed that CREST's formalised syntax and semantics only describe the use of transitions and update functions for dynamic behaviour, but omit the influence and action concepts that were informally introduced at the beginning of Section 4.1. This is due to the fact, that CREST was designed to be easily customisable and extensible. In this section we introduce the formal basis for these language extensions.

In some situations, several design patterns occur repeatedly and it might be of interest to add a dedicated modelling concept to express them. One example is the static linking of two ports' values by an update function. The basic idea of update functions is the continuous modification of a port value if the automaton is in a certain state. In many systems, however, some update functions should be triggered in every automaton state, to represent constant, state- and time-independent behaviour. Such a link can be for instance the conversion of resource values, e.g. from degrees Fahrenheit to Celsius, which should always be executed, independent of the entity's state. Another pattern that one repeatedly comes across is the need to execute update functions when a transition is triggered. Although these patterns can be expressed using the concepts introduced before, they lead to unnecessary repetition.

In this section, the two additional syntactic concepts of *influences* and *transitions with actions* are formalised. Both build upon CREST's syntax and semantics and naturally translate into CREST's formal system structure. Hence, these extensions are purely syntactic and do not add further modelling power or expressivity¹⁰. For each of these concepts, the formal definition (and translation to basic CREST) is presented and eventual constraints are specified.

¹⁰In the modelling community, such concepts are commonly referred to as "syntactic sugar".

4.3.1 Influences

As outlined, CREST uses the notion of influences to statically link the values of two ports. In the growing lamp system, `fahrenheit_to_celsius` is an example for such an influence. This influence continuously reads the room temperature, executes a transformation and writes the calculated result to its target port, independently of the current automaton state. In basic CREST the update would have to be specified for each state. Instead, an influence is used to replace its equivalent set of updates (that all refer to the same function) for each of the entity's states. The benefit of influences is that diagrams are more legible and an overload of updates is avoided.

Syntax Formally, an entity's influences are defined as follows:

$$\forall e \in \text{Entities}, \text{Influences}_e \subseteq \text{sources}(e) \times \text{targets}(e) \times \mathcal{U}$$

Similar to updates, all of a CREST system's influences are defined as the distinct union of the entities' influences.

$$\text{Influences} = \bigsqcup_{e \in \text{Entities}} \text{Influences}_e$$

The definition of influences requires that for each influence there is an update related to each state of the influence's entity. Formally, the constraint is expressed by the following implication:

$$\forall e \in \text{Entities}, \langle p_1, p_2, u \rangle \in \text{Influences}_e \implies \forall s \in \text{States}_e, \exists \langle s, p_2, u \rangle \in \text{Updates}$$

For the calculation of the modifier precedence, we have to provide the function's dependencies. An influence's dependency is only its source port.

$$\forall \langle p_1, p_2, u \rangle \in \text{Influences} \implies \text{dependencies}(u) = \{p_1\}$$

Semantic Constraint On the semantic side, an influence function's output is required to be only influenced by the source port's value. Thus, for any two bindings and *pre*-bindings whose values are equal for the influence's source port, the following condition must hold:

$$\forall e \in \text{Entities}, \forall \langle p_1, p_2, u \rangle \in \text{Influences}, \forall b_1, b_2, pre_1, pre_2 \in \text{Bindings},$$

$$b_1(p_1) = b_2(p_1) \wedge pre_1(p_1) = pre_2(p_1) \implies v(u)(b_1, pre_1, \delta t) = v(u)(b_2, pre_2, \delta t)$$

The returned value of the influence's function needs to be the same, even if there exists a port where binding or *pre*-binding values differ, i.e. if

$$\exists p \in \text{Ports}, p \neq p_1, b_1(p) \neq b_2(p) \vee pre_1(p) \neq pre_2(p)$$

4.3.2 Transition Actions

In some situations it is convenient to execute an update every time a certain transition is triggered. The concepts that were introduced before suffice to express such behaviour. An example for such a situation is a counter that increments a value every time a specific transition is fired, as modelled in Figure 4.10a. Instead of passing directly from *Off* to *On*, an intermediate state *Count* is added. After transitioning to this intermediate state, the *plus_one* update performs the value increment. The *Count*-state is left immediately upon completion of the updates, since the next transition specifies a guard that always evaluates to *True* (here modelled by the function *true*) and hence is triggered immediately.

To alleviate the burden of having to define an intermediate state for each such action-update, the CREST syntax is extended to include transition-actions. The graphical syntax of such an action is shown in Figure 4.10b. Actions are defined as updates connected to a transition, rather than a state.

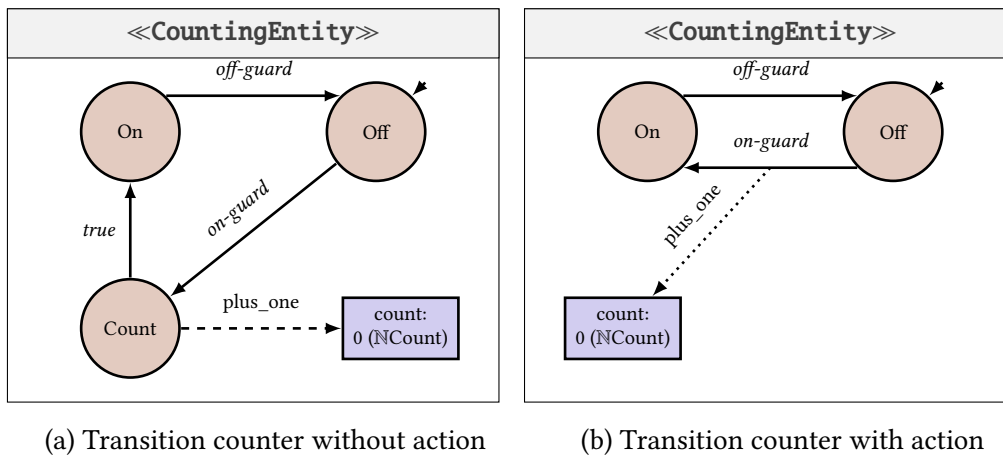


Figure 4.10 – Example of a counting entity with and without transition action. The models increment count each time the *Off*-to-*On* transition is triggered.

Note that this feature strongly builds upon CREST’s semantics, which triggers updates right after entering a state. If CREST were to first execute all enabled transitions and only trigger the updates when the automaton cannot not advance any further, *plus_one* would never be executed and it would be impossible to implement a counter such as this one.

Syntax Structurally, each transition with actions is defined as a guarded transition (using two states and a transition guard name) that is extended by a set of tuples of the form $\langle \text{target-port}, \text{update function name} \rangle$. The tuples each map a target port to one update function name.

$$Transition_{SAct} \subseteq \bigcup_{e \in Entities} (States_e \times States_e \times \mathcal{T} \times \mathcal{P}(\text{targets}(e) \times \mathcal{U}))$$

As explained above, the introduction of actions is syntactic sugar, added to the language to increase usability. This means that each transition with actions can be

also expressed through an extra state, two transitions and a set of updates (one update per action), as shown in Figure 4.10a.

This translation is formalised by the following implication: For each defined action-transition quadruple, there exists an extra (intermediate) state t' , a transition from the source-state s to t' that is guarded by the original transition guard and another transition from t' to the target-state t that is guarded by *true*, a guard function that always returns True.

For each target port-action pair, a new update is introduced that is linked to t' so that the update is executed right after the automaton transitions to t' .

$$\forall e \in Entities, \forall s, t \in States_e$$

$$\langle s, t, g, pas \rangle \in Transitions_{Act} \implies \left(\begin{array}{c} \exists t' \in States_e, \langle s, t', g \rangle, \langle t', t, true \rangle \in Transitions \\ \wedge \\ \forall \langle p, a \rangle \in pas, \langle t', p, a \rangle \in Updates \end{array} \right)$$

Semantic Constraint Note that transition actions are always called with $\delta t = 0$. This is because transitions are instantaneous and hence time can neither pass during the execution of the transition nor the update. Therefore, actions should behave the same, independent of the δt value.

Formally we require that an action always returns the same value, independent of the timestep δt it is called with. We assume that for any two timespans $\delta t_1, \delta t_2, \in \mathbb{T}$ the action's return value is the same.

$$\forall \langle s, t, g, pas \rangle \in Transitions_{Act}, \forall \langle p, a \rangle \in pas,$$

$$\forall bind, pre \in Bindings, \quad \forall \delta t_1, \delta t_2, \in \mathbb{T}, v(a)(bind, pre, \delta t_1) = v(a)(bind, pre, \delta t_2)$$

4.4 Language Analysis

CREST aims to provide a powerful language for the dedicated purpose of CPS modelling. Before implementing the details of the language, however, many other languages were evaluated for their fitness to model resource flow CPSs. This knowledge influenced the design of CREST, as existing languages' strengths inspired CREST's features, while weaknesses of other DSMLs were avoided. CREST's principle structure, including its entity hierarchy and port system, for instance, is inspired by architecture description languages (ADLs) and the specification of continuous behaviour and variable evolutions is a key feature of hybrid automata (HA). However, similar to these systems, CREST also imposes constraints.

This section analyses CREST and describes its handling of some well-known modelling problems such as *Zeno* behaviour and parallel computation. Furthermore, it outlines the relationship to three other formalisms, namely hybrid automata, Petri nets and DEVS, by pointing out commonalities and differences between these formalisms and CREST.

4.4.1 Language Design and Modelling Considerations

CREST is based on several, deliberate design choices that might seem counter-intuitive when regarding the general trends of modelling languages. One criticism could be that it merges architecture and behaviour within the same language concepts and diagrams. Usually, modelling languages use viewpoints, to allow engineers to develop individual system aspects independently. The separation of a system's structure and behaviour increases the complexity of the DSL, however, since it introduces new needs for viewpoint synchronisation and the verification of system coherence. In regard of CREST's target users, which might not be familiar with such processes, this increases complexity and lowers the language's usability and suitability. Nonetheless, CREST offers a practical means to separate an entity's architecture from its behaviour. Using *entity abstraction* (see Section 4.1), it is possible to model a component by its scope and communication interface only. This behaviour-masking allows the entity's use for structural composition even before its dynamics have been defined. Based thereon, the models can already be used for various analyses such as the discovery of dependency cycles or resource incompatibility between ports.

The decision to merge structure and behaviour renders CREST a very pragmatic modelling language. The DSL's purpose is to help the design and creation of CPSs such as smart homes. In most cases, such systems are built "bottom-up", where users select available devices according to certain requirements. For instance, if a system should have automatic light controls, a smart home builder will purchase one of the available IoT light bulbs. The system itself is then composed from the chosen devices. This approach to CPS creation is reflected in CREST's modelling process, which encourages the composition of entities to larger systems. As a result, the architectural aspect of CREST models is imposed by the represented system components, while the behaviour emerges from entity composition.

A final, generic point of discussion is that CREST is a DSL that aims to provide a coherent means to CPS modelling. Based on this concept, the language enforces the use of automata for behaviour specification and the use of a hierarchical entity structure, encapsulation and ports for the structural aspect. Some researchers might criticise that these means are not generically "the right" choices. Indeed, the multi-paradigm modelling community embraces the view that in all situations the "most appropriate" formalism or language should be used. The problem with this choice is evidently the highly increased complexity of the modelling approach, that comes with the expression of different components in different ways. To avoid confusion or erroneous behaviour, modellers that work with such systems have to pay close attention to the exact syntax and semantics of each component. As this reduces usability and significantly steepens the modelling learning-curve, CREST avoids this concept in favour of a streamlined, clear and unambiguous formalism.

Nevertheless, CREST embraces a limited multi-paradigm view to increase usability. As it neither prescribes a syntax nor semantics for the definition of transition guards and update functions, modellers can use any appropriate means to express such behaviour. Thus, expert engineers might use differential equations, while users of CREST's Python DSL will use source code, for instance.

4.4.2 Zeno Behaviour

One of CREST's most vital constraints is the exclusion of Zeno-behaviour. Formally, a system execution is called Zeno, if there is an infinite number of transitions that are executed in a finite amount of time. In CREST for example, there could be a cycle of transitions, where every transition is always enabled.

Figure 4.11 for example shows two entities that have Zeno behaviour. Figure 4.11a exposes an entity whose transitions are both permanently enabled if `port`'s value is below five. Stabilisation of this system would force CREST to repeatedly take any enabled transition, and thus the model would transition infinitely often between the two states A and B. Figure 4.11b shows a different kind of Zeno behaviour, where infinitely many transitions occur in a finite amount of time. The entity iteratively halves the amount of time it spends in state `Wait`. After an initial waiting time of one time unit, the entity transitions to `reset` and back and uses the action `halve_wait_time` to decrease `wait-time` to 0.5. After 0.5 time units the transitions are activated again and `wait-time` is set to 0.25. This cycle continues, setting `wait-time`'s values to 0.125, 0.0625, 0.03125, and so forth. It is however well-known that for a finite number of iterations, the sum of the sequence $1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots$ approaches, but never reaches 2. Therefore, it is impossible to simulate an advance of two time units, since an infinite number of transitions would have to be executed before. Similar situations occur in loops of infinitesimal-transitions.

The modelling of Zeno behaviour has been studied previously in many publications (e.g. in [Mos07]). For the simulation and verification of CREST systems, we forbid Zeno behaviour for the above reasons. Developers must pay attention to not accidentally model such behaviour.

CREST's implementation aims to help developers identify Zeno behaviour in their models by analysing system traces for recurring patterns. When discovered, an implementation should offer the possibility to break this cyclic execution of transitions (e.g. by manual suppression of transitions) or by defining thresholds for modification of values. In the example of Figure 4.11b this might be implemented by ceasing to decrease `wait-time` in `halve_wait_time` if its value drops below a certain minimum. Despite these naive attempts, the discovery of modelled Zeno behaviour significantly more difficult for complex systems and is unsolvable in general.

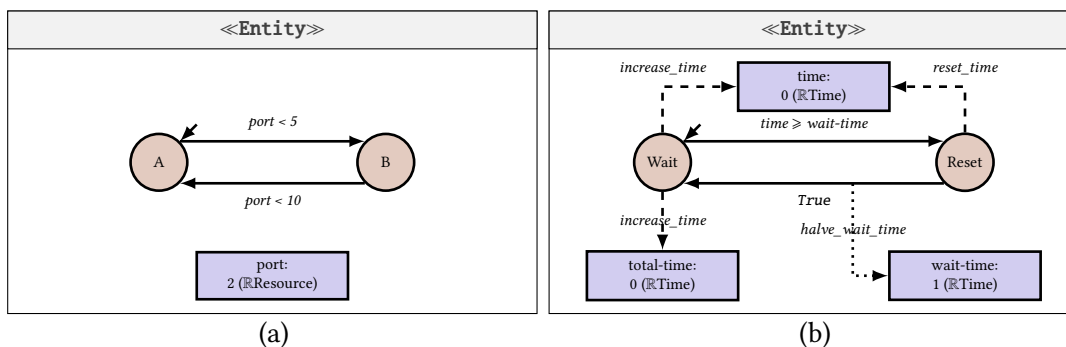


Figure 4.11 – Zeno behaviour. Left: Both transitions are permanently enabled. Right: `total-time` cannot reach 2 without infinitely many transitions.

4.4.3 Modifier Execution Order and Parallel Computation

The syntax and semantics sections introduce several concepts to define dependencies between a system's ports and to create an execution order for CREST's modifiers (i.e. its updates, influences and subentities). Although a complete proof of validity exceeds the scope of this discussion, this section briefly revisits the subject and provides the reasoning behind the concept.

The need for definition of an execution order is easily explained using the example shown in Figure 4.12. The figure depicts a sequence of ports that are connected by influences¹¹. CREST's semantics define, that all modifiers within an entity need to be executed, so that no port value update remains pending.

In the example below, a modification of the `input` port's value should be directly reflected by the update of the `local` and `output` ports. It is however necessary that `propagate_A` is obligatorily executed before `propagate_B`. If these two were to be triggered in the wrong order, `propagate_B` would first write the (outdated) value to the output port, before receiving its new value by `propagate_A`. Thus, `output` would carry the wrong value, at least until the next stabilisation. Such situations can be avoided by the use of the ordering created by *ordered-modifiers*.

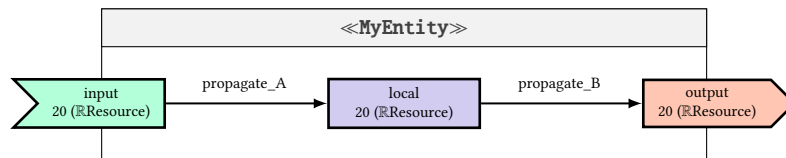


Figure 4.12 – A (partial) CREST diagram with a sequence of ports connected by influences. Note that the entity is not complete as it misses e.g. a state automaton.

Parallel Computation & Concurrency Evidently, the stabilisation and advance of time in large systems can be computationally demanding and require significant resources quantities. Next to the modification of the model itself, e.g. by increasing the level of abstraction and thereby removing unnecessary details, computation can be sped up by performing certain calculations in parallel. CREST's semantics define that the stabilisation process executes an entity's modifiers in a certain order. This order is based on port dependencies, such that a modifier m_1 (an update or subentity) which writes to a certain port p is executed before any m_2 that reads p . As a result, the semantics assert that behaviour is computed and executed in a logically correct order. The formal semantics however do not elaborate on the fact that the computation of such two components can be effectively calculated in parallel.

In case two modifiers m and m' exist that do not have any dependencies between their ports, they can be triggered in parallel. This is expressed by requiring that there does not exist any port read by m that depends on any port read by m' (thus also all ports written by m') and vice versa. Formally, $p \not\prec p' \wedge p' \not\prec p$ for all ports p read

¹¹Note that influences were chosen for graphical simplicity. These dependencies could equally be modelled using updates that read one port and write another, or subentities where the source port is an input and the target is an output.

by m and all ports p' read by m' . We call these two modifiers m and m' *independent*. The theory of this parallel execution is closely related to Kahn process networks and operator networks found in synchronous languages such as Esterel and Lustre.

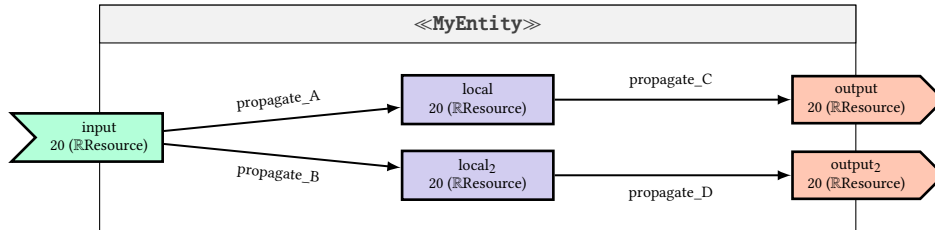


Figure 4.13 – A (partial) CREST diagram with independent influences that can be executed in parallel.

Figure 4.13 shows an example system that offers parallel execution possibilities. The system consists of an entity with five ports that are connected by four influences. As can be seen from the diagram, there exist no dependencies between the upper two influences and the lower ones. This means that the execution of these influences can occur in any order, as long as *propagate_A* is executed before *propagate_C*, and *propagate_B* before *propagate_D*. However, provided that enough computation resources are available, *propagate_A* and *propagate_B* could be executed in parallel and thereby reduce the time needed for the entity’s stabilisation process.

In fact, as any two modifiers without dependencies can be executed in parallel, the following pairs of influences can be safely executed at the same time:

- *propagate_A* and *propagate_B*,
- *propagate_B* and *propagate_C*,
- *propagate_A* and *propagate_D*, and
- *propagate_C* and *propagate_D*.

Race Conditions The parallelisation of computation is a complex subject for most tools and languages. Guaranteeing semantic correctness usually requires significant additional effort. Nonetheless, race conditions and so-called “dirty-reads” (i.e. use of outdated values) and “dirty-writes” (i.e. overwriting values) remain problematic and their avoidance is of high interest.

The ordering of modifiers is presented above as a means to avoid dirty-read scenarios, and to ensure that all updates and influences always access the latest value binding of a port. As for dirty-writes, the problematic is completely resolved by CREST itself. CREST’s formal system structure forbids that two updates that are related to the same automaton state write to the same target port. This constraint also extends to influences, since they are syntactic sugar for a set of updates. Hence, a model as shown in Figure 4.14 is not allowed. The two influences *propagate_C* and *propagate_D* cannot both write to the output port, since CREST cannot decide which influence’s modification is more important and CREST does not provide a dedicated

concept to resolve such write conflicts¹². Instead, CREST encourages the use of update functions that read the values of `local` and `local2` and explicitly define which value to write to `output`. Thus, the language is kept simple and easy to learn.

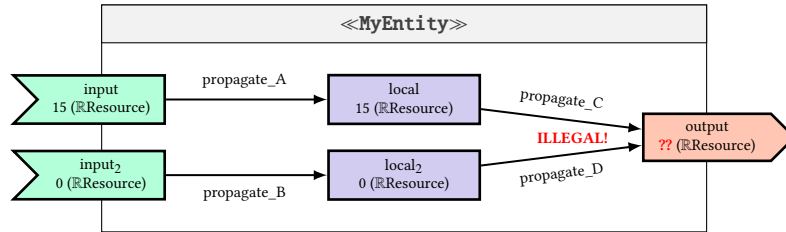


Figure 4.14 – CREST forbids that two modifiers concurrently write to the same port.

4.4.4 Structural vs. Temporal Non-Determinism

CREST’s semantics describe the support of non-determinism by allowing two transitions to be enabled at the same time. In such situations any one of the transitions can be chosen to be executed – no selection strategy is imposed by the formalism. Such constellations are referred to as *structural* non-determinism, since it is rooted in the model’s structure. This means that if a CREST model does not have any reachable state where two transitions are concurrently enabled, the model’s evolution is completely deterministic. This is due to the fact that CREST implements *must*-semantics, where a transition has to be triggered if it is enabled.

When comparing CREST with hybrid automata (HA) however, it becomes evident that the latter supports another kind of non-determinism. In their most general form, HA implement *may*-semantics, where an automaton does not have to trigger the transition although it is enabled. Instead, the automaton may remain in its current state and activate the transition at a later point. This is known as *temporal* non-determinism, since it creates diverging execution traces based on the time that a transition is executed. Depending on the time base \mathbb{T} , there can be infinitely many execution traces (e.g. if $\mathbb{T} = \mathbb{R}_{\geq 0}$).

A May-Semantics Automaton An example for a system with may-semantics is shown in Figure 4.15. The model represents an automatic heater module and the evolution of some temperature value. The HA consists of two states *on* and *off* (called *locations*) and manages one continuous variable x . Each location specifies the evolution of x ’s value via the \dot{x} variable. For example, *on* defines $\dot{x} = 0.5$, which means that the variable’s value grows by 0.5 per time unit. Finally, locations are annotated using *invariants*, which dictate how long an automaton can remain in a certain location. For instance, *on* specifies that x has to be below or equal to 40 in this location.

Transitions allow the switching of locations and are labelled with transition events, guard conditions and variable updates. Events specify signals that is emitted when the transition is taken (*ON* or *OFF*) and can be used to synchronise the transitions

¹²Many HDLs overcome this problem by using “resolved” signals to make this logic explicit

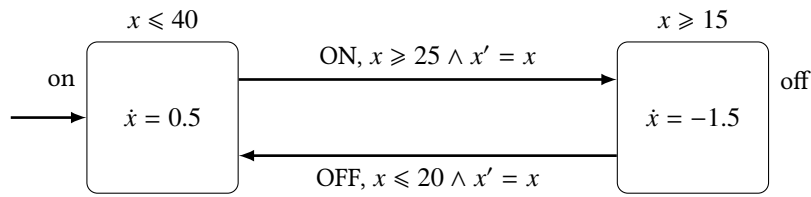


Figure 4.15 – Example of a HA that models a very simple heater module using *may*-semantics. x represents the temperature value and \dot{x} the rate at which x changes.

of several automata. *Guards* are invariants that specify when the transition can be taken (e.g. $x \geq 25$). Variable updates are statements that discretely modify the values when a transition is taken (similar to CREST’s actions). In the heater model, both transitions define $x' = x$, i.e. that the variable’s value after the transition x' is equal to its current value x (before the transition). Thus, no discrete “jumps” are modelled.

This simple HA highlights the effect of using *may*-semantics. For example, the transition from *on* to *off* becomes enabled when $x = 25$. *on*’s invariant, however, defines that the automaton may remain in the location until x reaches the value 40 (i.e. $x \leq 40$). This creates a certain time interval in which the automaton’s evolution is non-deterministic, as the transition can be taken at any point.

For example, if the variable’s initial value is $x = 15$, the transition will become enabled after spending 20 time units in state *on*. The location’s invariant states that the location must be left after 50 time units, when $x = 40$. This means, that the HA can transition to *off* at any of the infinitely many times in the interval $[20, 50]$.

Approximation of *may*-Semantics CREST’s use of *must*-semantics does not permit temporal non-determinism. This property renders simulation and verification manageable, since the number of evolutions is limited by the model’s structure. Thus, it is impossible to model a transition that executes “at some point” in an interval.

However, CREST can be used to implement an approximation of this behaviour. The general idea of this workaround is to create the possibility to delay a given transition by introducing another transition to an alternative state with the same guard as the original transition. An example is presented in Figure 4.16. The model on the left displays the CREST diagram of a heater model. It triggers transitions as soon as they become enabled (i.e. when x reaches 25). To create an approximation of *may*-semantics, an additional state *on'* and a transition to *on'*, which becomes enabled at the same time as the transition to *off*, is introduced. Thus, when $x = 25$, both transitions are enabled and the automaton can transition to *on'* instead of *off*. After a pre-defined time τ , the automaton transitions back to *on*, where it can (again) either switch to *off* or delay further by transitioning to *on'*. The τ -delay is modelled using a local port *time*, whose value is reset to zero when transitioning to *on'*, and continuously increased while in that state.

To avoid infinite delay possibilities, the invariant condition (e.g. $x \leq 40$) can be added to the transition guard, thereby limiting the transitions to *on'*. To guarantee the semantic correctness, *on'* also has to define the same update functions as *on*.

This example shows that using the presented approach, a delay of a transition

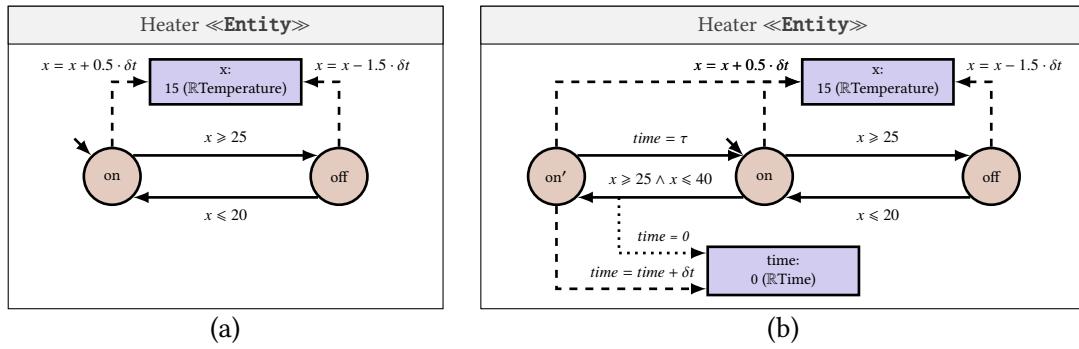


Figure 4.16 – Approximation of temporary non-determinism and may-semantics. The CREST model on the left uses classical may-semantics. The new version on the right allows to delay the transition to *off* by switching to a *on'* for τ time units.

can be simulated. The solution requires, however, a predefined delay granularity τ . Depending on how small the value τ is chosen, this solution can severely impact the simulation and verification performance of CREST. It also has to be pointed out that τ needs to be a value from the time base \mathbb{T} but cannot be zero or the infinitesimal value ε , as this would introduce Zeno behaviour.

Modelling Must-Semantics with May-HA It is also worth mentioning that any CREST system can always be modelled as a HA, even if the underlying semantics expose may-behaviour. To do so, it is necessary to define an invariant for each location that limits the possible time spent to the point at which a transition is enabled. Thus, each location's invariant must be the conjunction of all negated transition guards. As soon as any transition guard is enabled, the invariant is not satisfied any longer and the newly enabled transition has to be taken. In the example of Figure 4.15, we might replace the invariant for *on* with $\neg(x \geq 25)$ and *off*'s invariant with $\neg(x \leq 20)$ and thereby model must-behaviour. Formally, such invariants can be expressed as follows:

$$\forall l \in Loc, \exists i \in Inv, i = \bigwedge \{ \neg g \mid \langle l, l', g \rangle \in Trans \}$$

where *Loc* are the HA's locations, *Inv*, its location invariants and *Trans* the transitions between locations such that $\langle l, l', g \rangle \in Trans$ is the transition from location *l* to *l'* and *g* is the transitions guard¹³.

Thus, CREST system models can be potentially be expressed as HA with must- as well as may-semantics, which enables modellers to reuse simulation and verification tools that were developed for HSs.

4.4.5 Composition Aspects

As shown at the beginning of this chapter, CREST uses a hierarchical approach to assemble large systems. Hybrid systems (HSs), on the other hand, use a different

¹³A complete formalisation of HA can be found in dedicated publications such as [Ras05]

composition approach. Since they do not introduce hierarchy, they require a different means to synchronise the discrete and continuous advances within their components. The parallel execution of transitions in multiple automata is orchestrated using events (a.k.a. *synchronisation labels*). Transitions that are labelled with the same event (e.g. ON and OFF in Figure 4.15) are executed in parallel and thereby advance concurrently. The continuous evolution of an automaton’s state, i.e. its set of variables, is usually modelled via shared variables [AHH96]. This means, that several automata can reference the same variable and use it e.g. in their respective transition guards. However, it also means that all locations of all automata need to be synchronised on the variable’s rate. If two concurrent locations specify contradictory rates, the system is invalid and cannot be simulated. Resolving this issue often requires some form of “ownership” concept or priority for variables.

CREST refrains from sharing variables to preserve its locality principle and instead chooses to follow an alternative approach inspired by ADLs and dataflow languages. Each entity modifies only its own local ports and outputs and defines updates and influences to propagate information. From a practical point of view, this means that CREST remains closer to a CPS setup, where components encapsulate their internal values and only expose their output interface ports.

Hierarchical HA Composition Another approach is the use of hierarchical hybrid automata [Jie+99]. This formalism proposes that a HA can encapsulate other HA in its locations. Hence, the evolution of continuous variables in a location is not described by ODEs, but by the embedded automata, which allows more complex behaviour. In CREST’s domain, this would mean that each automaton state could embed subentities that are activated when their parent-automaton state is active.

The problem of this approach is the mixing of architectural and behavioural aspects. CREST’s methodology to entity composition treats subentities as explicit parts of the hierarchy, similar to e.g. AADL. Thus, it logically separates the static architecture (i.e. subentities) from the behavioural aspect (i.e. its state automaton).

If the subentities were to be modelled inside automata states, the system’s structure could change after every transition, since the new state might specify a different embedded subentity structure. In the growing lamp system, for example, the light module and heat modules always remain active parts of the system, independently of the growing lamp’s FSM-state. When modelling this system with the hierarchical HA approach, the light and heat module can be specified within the growing lamp’s automaton states. In states where only the light module is used, the heat module’s specification would even be omitted. This removal however, distorts the architectural view of the system, since the actual heat module, including its ports and connections, remains part of the system even if it is not active. Another problematic is the replication of information, as common functionality has to be replicated in each automaton state. For example, the light module would have to be specified inside every individual state, since its functionality is required in all situations.

The problems of placing structural concepts inside the behaviour automaton states increase further when large, nested automata with many states are required or when multiple hierarchical layers have to be modelled.

4.4.6 Commonalities with Hybrid Petri Nets

Petri nets (PNs) are a family of formalisms that are well-suited for the modelling of resource and control flows, especially in parallel and concurrent systems. One of the main advantages of these formalisms stems from their simplicity. System models are described as bipartite graphs using a set of *places* and *transitions*, which are connected using weight-annotated arcs. Places can represent resources or control and store *tokens* which e.g. specify the amount of the available resource. System evolution is modelled via transition “firings” that consume tokens from the transition’s *precondition* places (i.e. nodes connected by incoming arcs) and produce new ones in all its *postcondition* places (i.e. outgoing arcs). The number of tokens produced (resp. consumed) is defined by the arcs’ weights annotations¹⁴.

Figure 4.17 shows a PN that models the state automaton of an electrical device. Initially, there is one token in place *OFF*. Firing transition T_0 causes this token to be consumed and another token to be produced in place *ON*¹⁵. From place *ON*, the PN can either transition back to *OFF* or to *ERROR*. By default, PN’s do not hold information about which transition is taken if more than one is enabled. This means that the treatment of non-determinism is similar to CREST.

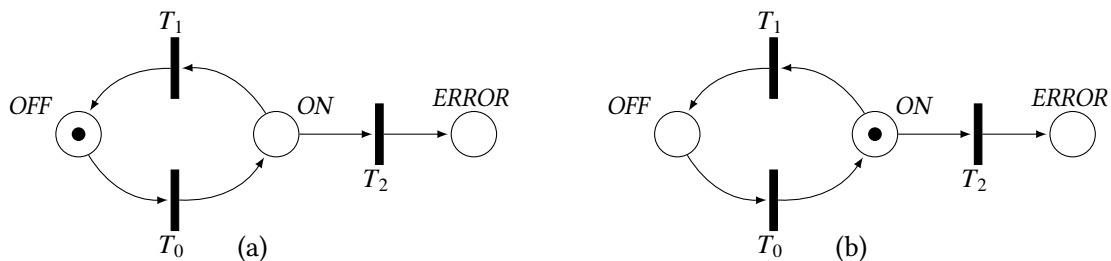


Figure 4.17 – A Petri net modelling a state automaton. The nets show the state before firing transition T_0 (left) and after (right).

Additionally, a PN can have several places with tokens at the same time. This means that concurrency can be modelled more easily. In CREST, for comparison, only one automaton state can be active at a time. To model the activation of two CREST states in parallel, it is necessary to hierarchically separate these automaton states and move them into different subentities. The result is that each subentity possesses its own automaton. This design choice can be seen as a trade-off, since CREST’s intuitiveness, where an automaton is always in one single state, comes at the cost of having to introduce additional entity hierarchy levels to model concurrency.

Next to the representation of control flows, PN’s can also be used to model the movement of resources inside a system. For example, Figure 4.18 displays a very abstract view of battery-powered growing lamp. The battery charge is modelled by the number of tokens in the *BATTERY* place. Firing of T_2 represents the operation of the lamp, which consumes one battery token and creates three tokens for heat

¹⁴By convention, the annotation is omitted for arcs with weight 1

¹⁵To avoid confusion, the reader is reminded that *tokens do not move* in a PN. Transitions consume tokens and produce new (different) ones.

and fifteen for light. In this model, the *HEAT* and *LIGHT* places measure the total amount of light and heat produced by this device. T_2 also consumes the token from *ON*, but immediately creates a new one. This is a common pattern in PNs to limit the firability of T_2 . To fire a transition, enough tokens are needed in all its precondition places. Thus, T_2 can only be fired with a token in *ON* and it is disabled if the token is in *OFF*. This small example visualises two things. First, PNs allow the expression of both control and resource flow within the same net, using the same syntax and semantics. Second, it highlights the activation of certain parts of the model (the resource flow) based on the configuration of other parts (i.e. the control flow). CREST uses a similar technique where updates are only active if the automaton is in their specified state.

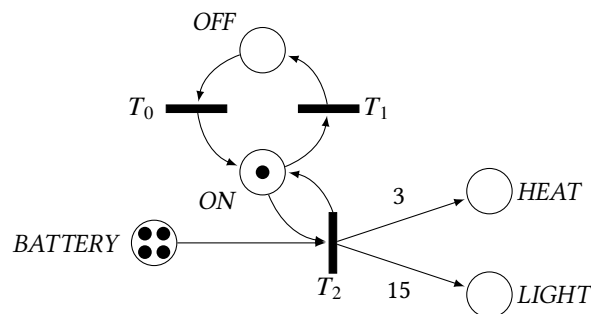


Figure 4.18 – A PN as resource flow model.

Timed Petri Nets When taking a closer look at this model, we see that there is no information about the frequency in which T_2 can be activated and how long this transition takes. For instance, the consumption and production of energy takes time, which needs to be modelled. There are discretisation techniques that express time as tokens, although this solution only works well for trivial models. For more complex cases, PNs have been extended in various ways to include this concept of time [Pop13]. In this thesis, we focus on PNs with timed transitions, even though other solutions could be used. This extension allows the annotation of each transition with a duration value. Setting T_2 's duration to 60, for instance, means that T_2 takes 60 minutes to execute and can only be fired once per hour.

The use of time in PNs allows us to specify how long a transition firing takes, but it also introduces a new issue. The example PN only allows us to turn the lamp on in one-hour intervals and does not allow for any finer time granularity. A naive solution would be to “split” each token into smaller parts and thereby create a finer-grained execution. For example, we could split each token in *BATTERY* into four “smaller” tokens. By modifying T_2 's duration equally, the transformation permits to use the lamp in 15-minute intervals. Note that even though the arc weights remain the same, the semantic “value” of the tokens produced in *HEAT* and *LIGHT* also has to be adapted to a quarter of its original value. Based on this principle, we could split tokens into smaller and smaller chunks until the required granularity is reached. In the growing lamp example, a precision of one second times should suffice for most application. However, we can easily imagine systems that require even more fine-grained execution times, and therefore token values.

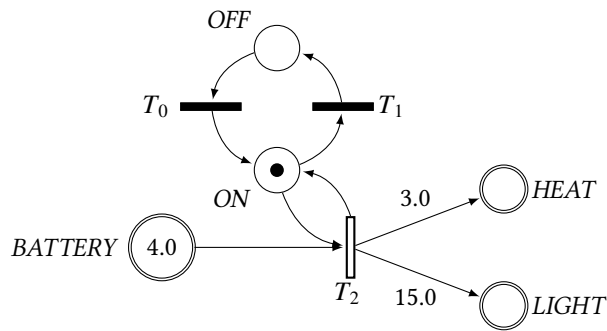


Figure 4.19 – A Continuous PN as model for the growing lamp.

To avoid dealing with large numbers of tokens and the need to split token values until a certain discrete granularity is found, Continuous Petri nets have been defined¹⁶. Continuous PNs are the result of “splitting a token value infinitely often”. In this situation, a place’s *marking* (the number of tokens inside) is represented using a real number, rather than an integer token count. The resulting continuous places and transition are shown in the lower part of Figure 4.19. Note that continuous (real-valued) places and transitions use a slightly altered graphical syntax to visually differentiate them. When triggering a transition in a continuous PN, it is necessary to define a *firing quantity*. It is a real value in the interval $[0, 1]$ and defines “how often the transition is fired”. For example, firing T_2 with a quantity of 0.25, consumes $1 \times 0.25 = 0.25$ tokens from *BATTERY*. It further produces $3 \times 0.25 = 0.75$ tokens in *HEAT* and $15 \times 0.25 = 3.75$ tokens in *LIGHT*.

Hybrid Petri Nets In fact, the PN shown in Figure 4.19 is a Hybrid Petri net, as it mixes both continuous and discrete parts. Again, it is possible to highlight the commonalities between Hybrid PNs and CREST, where a discrete formalism is used to model the control part (CREST’s state automaton) and the continuous aspect (i.e. CREST’s update mechanism) is used to model the flow of resources.

The flow of resources such as water and electricity within a PN can be easily modelled as shown above. In fact, by adding a transition T_3 to the growing lamp system which reduces the tokens in *HEAT* at a constant rate, it is possible to model a cooldown¹⁷. In this scenario, *HEAT* does not show the total amount of heat created by the growing lamp, but the net added energy (i.e. the difference between heat added and lost). Figure 4.20 displays the model of such a system.

However, we observe that with the means that were introduced so far, it is impossible to create a place that represents the amount of light that is currently being produced. The reason is that, in contrast to heat, light disappears immediately when the transition is not triggered. In the example, *LIGHT* holds the total energy that was provided to the lighting function. A place that shows the current amount of light being produced (measured e.g. in watt) should hold a zero-marking at any time T_2 is not executed and the light bulb’s wattage value during T_2 firings. With the presented

¹⁶For a more thorough introduction of the concept, the reader is referred to [AD98].

¹⁷This behaviour is expressed as *sink* transition, i.e. a transition without a postcondition.

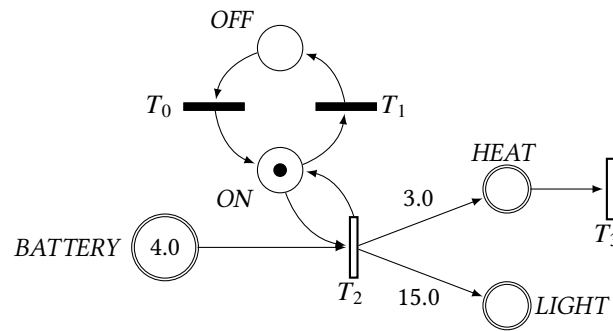


Figure 4.20 – Adding a sink-transition T_3 so *HEAT* always stores the net added heat.

Petri net formalisms, it is not (or only with highly complex nets) possible to model this behaviour. PNs however have been extended to allow the expression of this behaviour in formalisms such as Petri nets with Inhibitor Arcs and Higher-Level Petri nets [JR12]. The former allows the disabling of transitions based on precondition markings and the latter the use of data and variables inside tokens.

Another problematic is the modelling of non-numeric data inside a PN. In the models above, we saw that tokens represent Boolean and numeric values. A black token can indicate a certain state by being present or not (see *ON* and *OFF*, above). Other places count the number of resources available, to e.g. model the *BATTERY* charge. CREST systems, however, also model more complex data and enumeration types. The CREST growing lamp system introduced at the beginning of this chapter uses an $\{on, off\}Switch$ resource, for instance. Additionally, CREST’s updates execute function calls, its transitions compare port values to evaluate whether they are enabled. The modelling of these features in classic PNs is far from trivial, if not impossible. CPNs [Jen96] however introduce types into the PN formalism family, High-level Petri nets permit the implementation of complex transition guards and evaluate functions, and CO-OPN [BG00] allow the expression of object-oriented aspects inside PN. Some of these formalisms have already used within Hybrid Petri nets as documented in [Her+17] and [GU96].

Outlook In general, the integration of PNs for modelling is usually limited to a lower abstraction level, where models are automatically generated. Since PN models often contain hundreds of places and transitions a manual creation is difficult and error-prone, extensions of the base-formalism help this process only in a limited way. The advantage of PNs, however, lies in their simplicity and the use of very few syntactic and semantic concepts. This means that various system aspects (e.g. flow of control and data) can be efficiently expressed within the same model, and that tools can be implemented with fewer rules.

PNs are a tempting translation target for CREST models, since their popularity led to the creation of numerous analysis methods and simulation implementations that could be reused for the verification of CREST model properties.

4.4.7 Relationship to DEVS

CREST is also loosely related to other formalisms that combine discrete and continuous behaviour, such as hybrid DEVS [BK11]. Conceptually, it is easy to find the resemblance between DEVS and CREST. DEVS, by default, is a formalism that combines time-based (*internal*) transitions with event-based (*external*) transitions. Similarly, CREST continuously advances time between points where port values are externally modified (set-values). However, while CREST uses continuous semantics for variable values, DEVS is bound to discrete transitions between states. Quantized state systems (QSSs) have been introduced to overcome this issue, where variable values are discretised into “*quantized states*”. Each quantized state thus represents an interval of variable valuations. This allows the approximated simulation of hybrid behaviour within the discrete formalism. Tools such as PowerDEVS provide capabilities to automatically create QSSs based on atomic or coupled DEVS specifications and use a block diagram-based GUI to simplify modelling. Each block represents an atomic DEVS model within a coupled DEVS system. ProDEVS, is another hybrid modelling tool that supports state-machines [VFA15].

There are however certain other limitations to the representation of CREST models in the DEVS formalism. For example, non-determinism is neither allowed in DEVS nor QSSs, which means that only fully deterministic CREST systems could be translated. Furthermore, problems arise when it comes to the verification of such systems. Even though some research effort has been put into the verification of hybrid DEVS, the solutions mostly rely on mapping DEVS onto other formalisms such as timed automata and performing verification thereon [SW12].

4.5 Summary

This chapter introduces CREST, a novel modelling DSL that focuses on representing the transfer of resources within a CPS. The language features a graphical syntax that merges structural concepts known from ADLs (e.g. ports and hierarchical components) and behavioural specifications based on HA and dataflow languages. CREST’s semantics is comparable to the *must-semantics* of HA. It also asserts that the six key modelling aspects of resource flow CPSs are enforced. Thus, the important principles of locality, synchronism and non-determinism are maintained, while allowing for parallelism and continuous variable evolution.

Both, the syntax and semantics are defined on top of a formal foundation that strictly defines the general behaviour but allows for flexibility by underspecifying e.g. the precise language of transition guards and update functions. Thus, modellers can flexibly choose a specification formalism they feel comfortable with and use it to define the behaviour. Hence, CREST behaviour can be defined in various forms, including for example mathematical specifications using ODEs for modelling of precise natural phenomena or, alternatively, common programming languages for practical engineering systems with a focus on executable models. The next chapter describes the use of Python as a host for the implementation CREST, and the use of standard Python source code for the definition of entities, transitions and updates.

Chapter 5

CREST Implementation

The installation of custom assembly CPSs such as smart home systems usually goes hand in hand with the setup of hardware, configuration of software APIs and development of short code scripts to customise behaviour. Even though the majority of devices promise plug&play solutions, most non-trivial system workflows still have to be manually programmed. CREST’s target audience comprises expert modellers, who are usually experienced programmers, and technophilic CPS creators, who are familiar with the use of scripting languages and interested in applying their skills to advanced use cases such as systems modelling.

CREST’s implementation leverages this situation by building on the existing programming knowledge of its users and allowing the creation of models using an internal DSL [Völ+13] in the Python programming language. This reuse of a popular GPL provides many advantages for end-users and tool developers, such as the existence of established development practices and the reuse of the language’s infrastructure.

In CREST’s case, the choice for Python is built upon several reasons. First, Python is a popular language that is widely known to be versatile and easy to learn. It experiences support from a large and active user community and is continuously growing in popularity. The language is also pre-installed and readily available on most modern operating systems. From a language development point-of-view, Python offers many appealing opportunities, such as user-defined code annotations and a customisable meta-class system (e.g. to influence the class creation mechanisms). Furthermore, it features off-the-shelf compatibility with other languages (e.g. C, C++), tool bindings for third-party software and numerous native library packages for e.g. graph plotting, data analysis, machine learning and artificial intelligence. Another strongpoint that should not be underestimated is the availability of an existing syntax that many users are already familiar with. CREST uses Python’s native features (e.g. classes, modules) for the representation of system architecture and its dynamic behaviour (e.g. methods, functions) for the modelling of transition guards, updates and influences. Modellers can therefore focus on the development of their systems, rather than learning a new programming language.

The rest of this chapter provides an overview of `crestdsl`, CREST’s Python implementation. First, Section 5.1 outlines `crestdsl`’s capabilities and the supported modelling activities. Section 5.2 then introduces `crestdsl`’s approach to systems

modelling using the growing lamp as example. Section 5.3 provides details of CREST’s simulation capabilities and the implementation of a function for the discovery of the next discrete behaviour change time. Finally, Section 5.4 outlines CREST’s interactive development and execution environment, which is based on Project Jupyter¹.

This chapter provides various code listings to introduce `crestdsl`. These listings are also available online, in a dedicated source code repository hosted at <https://github.com/crestdsl/thesis-code>. In the same place, the reader will find the possibility to launch a web-based execution environment for the `crestdsl` code. Many thanks to the Binder² team for hosting their service.

5.1 Overview

Before discussing `crestdsl`’s capabilities in detail, it is of interest to relate it to the CREST formalism and CREST diagrams that were introduced in Chapter 4. Further, we take a general look at `crestdsl`’s support of various modelling tasks such as simulation and syntax checking.

Chapter 4 describes the CREST formalism including its abstract syntax and formal semantics. CREST diagrams are introduced as a modelling language implementation of CREST that uses a graphical, concrete syntax. It is evident that this relationship reflects the framework of formalisms and languages, that was defined by Broman et al. [Bro+12]³. Without the use of CREST diagrams (or another concrete syntax), the modelling process is a purely theoretical task that remains on an abstract level. Although the created (abstract) models capture the modelling intent (the purpose of the model) and its structure according to an abstract syntax definition, the created models are not concretised or persisted⁴. To be useful for practical tasks, the modelling process requires the use of a language with a concrete syntax.

Modelling `crestdsl` is an alternative implementation of a concrete syntax for CREST. Its goal is the rapid prototyping of system models and to serve as a basis for tasks such as simulation and verification. `crestdsl` expresses domain concepts as Python objects. “Modelling” is therefore the process of creating the required domain objects and their relationships, as shown on the right side of Figure 5.1. To simplify the creation of models and speed up the development process, `crestdsl` implements a convenient scripting API that can be used to programmatically create large models. A complete discussion of `crestdsl`’s APIs exceeds the scope of this thesis, however. Hence, only a small subset is discussed in Section 5.2.

¹<https://jupyter.org/>

²<https://mybinder.org/>

³A short review of this framework was also presented in Section 2.1.

⁴This activity is different from *conceptualisation* [SB10], which captures the “generalized idea of [...] interacting components and its desired functionality”. These models are “typically very informal in terms of detail and accuracy”. Abstract models are accurate but not expressed in a language.

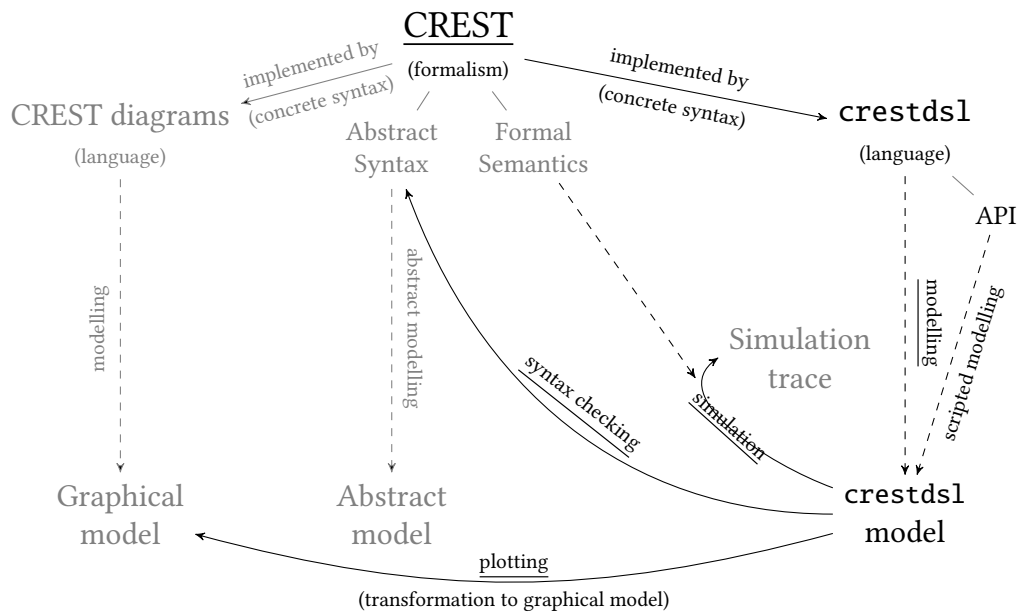


Figure 5.1 – Schema of CREST, CREST diagrams and crestdsl and their relations. The individual modelling tasks (e.g. simulation, and plotting) relate the concepts.

Lines represent relations, arrows modelling activities and dashed arrows instantiation or influences. Concepts from previous chapters are drawn in grey, modelling tasks that are discussed in this chapter are underlined.

For more details about the implementation of crestdsl, the reader is referred to the project’s online documentation at <https://crestdsl.github.io>.

Syntax Check crestdsl’s Python basis offers a lot of convenience for the creation of system models. One caveat is, however, that Python’s flexibility and dynamic type system open the door to wrong model configurations that are usually only discovered at runtime. To counteract this shortcoming, crestdsl implements a component that allows the *syntax checking* of a model. The process iterates over a model’s structure and asserts that it matches CREST’s abstract syntax. The checks test e.g. whether all entities define a current state, whether updates are linked to an automaton state of their own entity and whether transitions relate two states of the same entity.

Simulation Next to the modelling itself, crestdsl supports various other modelling tasks. For example, it provides a simulation component, that allows users to study the evolution of a model based on the advance of time or the modification of input values. To assert the correct simulation, crestdsl’s simulation process is implemented to closely reflect CREST’s formal semantics. To this extent, a sophisticated algorithm was developed at the core of crestdsl’s simulator. The algorithm allows the translation of crestdsl models to SMT constraints. The evaluation of these constraints permits the calculation of the points in time, at which where the model’s behaviour experiences discrete changes (“jumps”). Thus, an uninformed and costly step-based simulation process can be avoided.

Plotting Finally, `crestdsl` offers the possibility to transform the Python model into a diagram. This *plotting* process links the advantages of both CREST languages, by merging the development speed of `crestdsl` models with the simplicity and easy comprehension of CREST diagrams. Thus, users can develop a model using the scripting API and use CREST diagrams for an overview and simplified discussion.

5.2 `crestdsl` – CREST’s Python Implementation

CREST’s implementation is distributed as `crestdsl`, a software library for the Python programming language. `crestdsl` provides classes and functions for the definition of CREST models. To use the library features and start creating CREST models, it suffices to import the `crestdsl` library, using Python’s standard class-import mechanism, as shown in Line 2 of Listing 5.1.

`crestdsl`’s features (simulation, verification, drawing) are split across several subpackages. The classes for the creation of a CREST model are located in the `model` subpackage. Listing 5.1 shows how to import this dedicated subpackage and bind it to the `crest` namespace (Line 7).

Listing 5.1 – Importing `crestdsl`

```

1 # Import of the crestdsl library
2 import crestdsl
3
4 # We can also import a specific subpackage
5 # e.g. the model package which contains functions
6 # for creating CREST’s domain objects
7 import crestdsl.model as crest

```

Following the import statement, CREST entities can be created. In `crestdsl`, entities are instances of `crestdsl`’s `Entity` class. Users can add other CREST concepts (such as input ports or states) to the class by setting them as object attributes, as shown in Listing 5.2. Like in CREST, each entity `Port` requires the specification of its resource type and initial value. Resource types are instances of the `Resource` class, declaring a name and value domain. Similar to ports, states are defined by creating `State` objects and assigning them as attributes. Note that each entity needs to define exactly one of its states as current state using the `current` attribute.

Listing 5.2 – Definition of an entity

```

1 # use CREST’s domain types to specify the domain
2 watt = crest.Resource(unit="Watt", domain=crest.REAL)
3 lumen = crest.Resource(unit="Lumen", domain=crest.INTEGER)
4
5 my_lamp = crest.Entity()
6 my_lamp.in_port = crest.Input(resource=watt, value=100)
7 my_lamp.out_port = crest.Output(resource=lumen, value=0)
8 my_lamp.on = crest.State()
9 my_lamp.off = crest.State()
10 my_lamp.current = my_lamp.off

```

Entity Types This approach to entity definition can quickly become cumbersome. Especially when many entities with similar attributes have to be created, it is easier to define an entity type and use instantiation to obtain the individual objects. *crestdsl* allows the creation of entity types using Python’s class mechanism. A class must inherit from **Entity** to become an Entity type. Model concepts such as ports, states and transitions can then be specified as class attributes and methods.

Listing 5.3 shows the definition of an entity type `MyLamp`, which instantiates models that are equivalent to the object `my_lamp` above. The listing shows how the entity type is used to create two separate instances of `MyLamp` in Lines 7 and 8.

Listing 5.3 – Definition of entities via an entity type

```

1 class MyLamp(crest.Entity):
2     in_port = crest.Input(resource=watt, value=100)
3     out_port = crest.Output(lumen, 0)
4     on = crest.State()
5     off = current = crest.State()
6
7 my_new_lamp = MyLamp()
8 my_other_lamp = MyLamp()

```

Dynamic Behaviour Evidently, the lamp model shown in Listing 5.3 is not complete. It lacks functionality to switch from one state to another (i.e. transitions) and cannot modify the light output port. Listing 5.4 shows the definition of an entity type that defines guarded transitions and update functions. In *crestdsl*, all model concepts are implemented as classes, that should be instantiated and assigned as attributes. The definition of such a guarded **crest Transition** can be seen in Lines 8 and 9 of Listing 5.4. Note that the guard function itself is implemented as a Python **lambda** function that evaluates a condition and yields a Boolean value.

Additionally to the creation of **Transition** objects as attributes, *crestdsl* provides a convenient decorator⁵ API. This means that **@transition** can be used to annotate class methods and upgrade them to transitions. The decorators also specify the transitions’ source and target states, as shown for the `on_to_off` transition in Listing 5.4 (Lines 11 – 13). The use of decorators increases the language’s usability, it’s functionality, however, is equivalent to the use of **Transition** objects.

Similar to transitions, update functions can be declared both as **Update** objects or using the **@update** decorator. The listing specifies two update functions that modify the value of the `out_port`. Note how each update specifies its corresponding entity state in which it is active and the target port that will be updated. The functions additionally require a `dt` parameter to be specified, which at runtime will hold the time that has passed since the last time the update function was executed.

⁵Decorators are a well-known design pattern for aspect-oriented programming. Python’s decorators are implemented as annotations (e.g. `@my_decorator`) that intercept a function’s or method’s definition. For more information see <https://wiki.python.org/moin/PythonDecorators>.

Listing 5.4 – An entity type with transitions and update functions

```

1 class DynamicLamp(crest.Entity):
2     in_port = crest.Input(resource=watt, value=100)
3     out_port = crest.Output(watt, 0)
4     on = crest.State()
5     off = crest.State()
6     current = off
7
8     off_to_on = crest.Transition(source=off, target=on, \
9         guard=(lambda self: self.in_port.value >= 100))
10
11     @crest.transition(source=on, target=off)
12     def on_to_off(self):
13         return self.in_port.value < 100
14
15     # output = 90 watt output * 15 lumen per watt
16     output_when_on = crest.Update(state=on, target=out_port, \
17         function=(lambda self, dt: 90 * 15))
18
19     @crest.update(state=off, target=out_port)
20     def output_when_off(self, dt):
21         return 0

```

Entity Subclassing The aim of implementing a language as internal DSL is to reuse its existing infrastructure and simplify the interfacing with its ecosystem. Therefore, the implementation of `crestdsl` pays much attention to maintain compatibility with Python development practices and its programming paradigms. One of these best practices is the use of class inheritance. `crestdsl` uses this paradigm to encourage the extension, specialisation, adaptation and reuse of existing entities and avoid code repetition and low maintainability.

Listing 5.5 shows the example of a `SwitchLamp`. It is implemented as subclass of `DynamicLamp` (Listing 5.4). This new entity type adds an additional input to the lamp by defining a new `switch_input` port. The port is of type `switch` whose domain is comprised of set of values "on" and "off". `crestdsl` defines resources with discrete domains using lists, as shown in Line 1 of the listing. The example also extends and modifies `DynamicLamp`'s functionality, with `switch_off` and `off_to_on`, respectively.

Listing 5.5 – A class that inherits functionality from another class and extends it

```

1 switch = crest.Resource(unit="lampSwitch", domain=["on", "off"])
2 class SwitchLamp(DynamicLamp):
3     switch_input = crest.Input(resource=switch, value="off")
4
5     @crest.transition(source="on", target="off")
6     def switch_off(self): # extend DynamicLamp functionality
7         return self.switch_input.value == "off"
8
9     @crest.transition(source="off", target="on")
10    def off_to_on(self): # modify DynamicLamp functionality
11        return self.in_port.value >= 100 and \
12            self.switch_input.value == "on"

```

Constructors and Parameterised instantiation Another classic feature of object-oriented languages is the parameterisation of object creation using constructors. In Python, an object’s constructor method is named `__init__` and is called directly after the object has been created. By implementing a constructor method for an entity type, it is possible to dynamically adapt the entity’s functionality. Listing 5.6 shows an example of a `GenericLamp` type that permits the setting of the power threshold, i.e. how much power is required for the lamp to turn on, and the efficiency factor, i.e. what fraction of the input energy is converted to output energy. Python also allows for parameters to be declared as optional. This can be seen when initialising `powerful_lamp` in Line 19, which does not declare its efficiency and uses the default value, whereas for the `efficient_lamp` this value is set manually.

Listing 5.6 – A parametrisable class that specifies a constructor

```

1 factor = crest.Resource(unit="efficiency", domain=crest.REAL)
2 class GenericLamp(DynamicLamp):
3     threshold = crest.Local(watt, value=100) # default value
4     efficiency = crest.Local(factor, 0.75)
5
6     def __init__(self, threshold, efficiency=0.75):
7         self.threshold.value = threshold
8         self.efficiency.value = efficiency
9     @crest.transition(source="off", target="on")
10    def off_to_on(self):
11        return self.in_port.value >= self.threshold.value
12    @crest.transition(source="on", target="off")
13    def on_to_off(self):
14        return self.in_port.value < self.threshold.value
15    @crest.update(state="on", target="out_port")
16    def output_when_on(self, dt):
17        return self.in_port.value * self.efficiency.value * 15
18
19    powerful_lamp = GenericLamp(300) # default efficiency
20    efficient_lamp = GenericLamp(50, .97) # explicitly set eff.

```

The use of constructors is very powerful, since entities can be customised at run-time. For instance, ports and states can be dynamically added or updates modified.

Entity Compositions One of CREST’s most dominant features is its hierarchical entity structure. Obviously, this concept is also represented in the *crestdsl* implementation. Subentities are declared using the same class attribute definition approach as ports, states and transitions. Listing 5.7 provides an excerpt of a type with two subentities. The `LampComposition` consists of two individual lamps, that are controlled by the same light switch. If the composed lamp is turned on, the entity propagates the electricity to its subentities. Specifically, the “first” 100 watts are used to power the small, efficient lamp. In case there is enough energy available, the rest of the input power is used to drive the `big_lamp`, which will produce light if its threshold is surpassed. This functionality is controlled using update functions, that set the input port values of subentities. Lines 11 – 16 show an example for such a function.

Listing 5.7 – A composed entity with two subentities (full listing in Appendix C)

```

1 class LampComposition(crest.Entity):
2     # ... code for input and output ports
3
4     # subentities
5     big_lamp = GenericLamp(300)
6     small_lamp = GenericLamp(100, .9)
7
8     # ... code for states and transitions
9
10    # setting of subentity inputs
11    @crest.update(state=on, target=small_lamp.in_port)
12    def set_small_lamp_input_when_on(self, dt):
13        if self.in_port.value > 100:
14            return 100
15        else:
16            return 0
17
18    # ... more code for updates that set subentity inputs
19
20    @crest.influence(source=big_lamp.out_port, target=big_out)
21    def propagate_big_output(value):
22        return value
23
24    propagate_small_output = crest.Influence(
25        source=small_lamp.out_port, target=small_out)

```

Additionally, the subentities' light outputs have to be propagated to the outputs of the parent entity. The example uses **Influences** for the implementation of this functionality, since the link is static and independent of the `LampComposition`'s **current** state. Lines 20 – 22 and Lines 24 – 25 show two different ways of defining influences between two CREST ports. Note that the latter (`propagate_small_output`) does not specify an influence function that modifies the value which is written to the target port. By omitting this parameter, `crestdsl` uses the default an expression that returns the value that is currently held by the influence's source, without modification.

Syntax Checking As described in Section 4.1, CREST models have a thoroughly defined structure. For example, influences have to connect two ports and transitions need to establish relations between states of the same entity. Python on the other hand, is a highly flexible, dynamically typed language. In order to help users create syntactically valid models, `crestdsl` provides the `SystemCheck` class. This module can be called with either an entity type or an instance of a type. It then performs a series of checks that e.g. test the system's entity hierarchy and correct setup of entities, transitions and updates. Though the system checking routine is not exhaustive (there is no testing of dynamic runtime behaviour), the tool serves nonetheless as an important helper for preventing common modelling errors.

Listing 5.8 shows a short usage example of the `SystemCheck`. After its initialisation, `check_all()` runs all tests and returns `True` when the entity is correctly setup (Line 4). If an invalid system configuration is encountered, e.g. the erroneous set-

ting of the `current` automaton state to `None` (Line 6), the checking writes an error message to the error log and returns `False` (Line 9). The system checking can also be instructed to trigger an exception that provides additional information about the specific problem, as seen in Line 12. This can be useful for the creation of automated unit tests.

Listing 5.8 – Example of the `crestdsl SystemCheck` class

```

1 from crestdsl.model import SystemCheck
2 # create instance and SystemCheck object
3 gl = GenericLamp(300, .85)
4 SystemCheck(gl).check_all() # returns True
5
6 gl.current = None # point current state to None
7
8 # write to error log: [...] Entity has no current state
9 SystemCheck(gl).check_all() # returns False
10
11 # write to error log: [...] Entity has no current state
12 SystemCheck(gl).check_all(exit_on_error=True) # AssertionError

```

5.3 Simulation

`crestdsl`'s `Simulator` class implements its functionality as prescribed by the formal semantics. Its interface exposes the following main interaction methods

- `stabilise()` propagates all port value changes and triggers transitions until a fixpoint is reached;
- `advance(dt)` advances the time for `dt` time units; and
- `advance_to_behaviour_change()` advances time to the point where a new transition or another discrete behaviour change happens⁶.

HeatModule Example The rest of this chapter uses a heating module as running example. This module is a CREST component that can be used inside a growing lamp, such as the one introduced in the previous chapter. Compared to the `HeatElement` of the growing lamp in Figure 4.1, the `HeatModule` is an advanced version that features multiple states, including an error-state which is entered when the device overheats.

Figure 5.2 shows the CREST diagram of the heat module. The full entity description and its `crestdsl` source code, which also specifies the behaviour of the update functions within the system, is provided in Appendix C.2. The rest of this section uses excerpts of this listing to explain the simulation and calculation of the next transition time.

⁶For convenience reasons, `crestdsl` allows behaviour changes also in the form of `if-then-else`-conditions inside updates and influences.

Listing 5.9 – Use of the Simulator for stabilisation and time advance

```

1  from crestdsl.simulation import Simulator
2
3  mod = HeatModule()      # create a crest system
4  print(mod.current, mod.timer.value, mod.internal_temp.value)
5  # prints : HeatModule.off 0 0
6
7  sim = Simulator(mod)    # instantiate the simulator
8  sim.stabilise()        # perform the stabilisation routine
9  print(mod.current, mod.timer.value, mod.internal_temp.value)
10 # prints: HeatModule.off 0 22
11
12 mod.switch.value = "on" # modify the system's inputs
13 mod.electricity.value = 300
14 sim.stabilise()        # stabilise again
15 print(mod.current, mod.timer.value, mod.internal_temp.value)
16 # prints: HeatModule.on 0 22
17
18 sim.advance(10)         # advance time for 10 time units
19 print(mod.current, mod.timer.value, mod.internal_temp.value)
20 # prints: HeatModule.on 10 122.0

```

Each of the three main interaction methods has the capability to modify a model's state. Listing 5.9 shows an example of the use of a simulator. After the import of the class and instantiation with an entity object (in this case a `HeatModule` entity), a simulation can be run. The displayed code uses `print` functions to show the port's value changes (Lines 4, 9, 15 and 19). The according output of these statements is displayed as Python comments (Lines 5, 10, 16 and 20, respectively). As can be clearly seen, the `HeatModule` is instantiated with `off` as its `current` automaton state and both the `timer` and `internal_temperature` ports set to zero. Using the simulator's `stabilise` method, the system is brought into a state where all updates and enabled transitions have been executed (Line 8). Thus, the internal temperature is adjusted to the minimal value 22 (the assumed room temperature). Since no time has passed yet, the `timer`'s value remains 0 and the system's state is still `off` because the requirements for a transition to `on` are not yet satisfied.

In the next step (Lines 12 and 13), the lamp is switched `"on"` and electricity is applied to its input. The subsequent stabilisation discovers the newly enabled transition to state `on` and triggers it. The `print` statement proves this, but also shows that the `timer` value remains at 0, since still no time has passed yet.

The advance of time (Line 18) triggers the updates with a time delta of 10 time units. Thus, the `timer` value is updated and the internal temperature value reflects the increase of heat over time.

5.3.1 Different Simulators

`crestdsl` provides in fact several simulator implementations for the exploration of model behaviour. They mostly offer the same functionality and API, but differ in

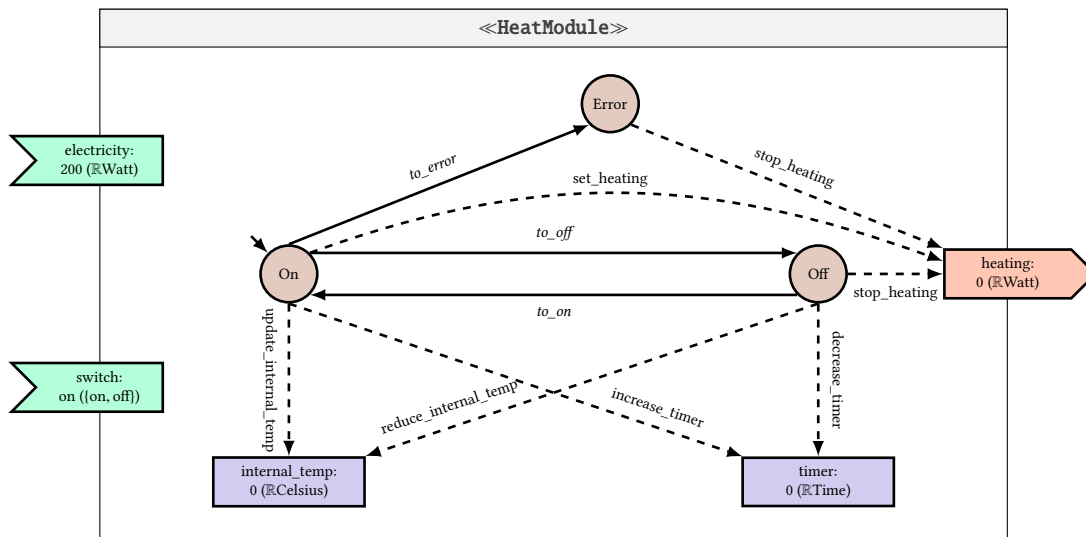


Figure 5.2 – A heat module with an on-timer and automatic cooldown period. The behaviour is mainly controlled by two input ports (electricity and switch). The module breaks when the internal temperature surpasses a threshold.

other aspects of the implementation. Below, three of them are presented, whose difference lies in the way they treat non-determinism.

As explained in the previous chapter, CREST's evolution is non-deterministic if two or more transitions are enabled from the same, currently active entity state. Despite the fact that non-determinism is a vital aspect of CREST to model the seemingly random behaviour, it is of high interest to also support its resolution and manual selection of the transition to follow, e.g. for the deliberate exploration of specific system behaviour. The treatment with such non-determinism is the discriminating difference between the individual simulators. The three simulators implement the following behaviour:

1. The first implementation, the most basic CREST simulator (*Simulator*), randomly chooses one enabled transition when it encounters non-determinism. It thus strictly follows the basic CREST semantics. This module is useful for initial explorations in non-deterministic models (e.g. to get a feeling for possible behaviour scenarios) and for general simulation of deterministic models.
2. Secondly, the *InteractiveSimulator* is used when it comes to manual exploration of non-deterministic models. Anytime a situation with multiple enabled transitions is encountered, the simulator stops and prompts the user to make a choice. Evidently, this feature is meant for the exploration of smaller models, where such situations occur infrequently. For highly non-deterministic models this form of exploration can become tedious. An example screenshot showing the use of the *InteractiveSimulator* is presented in Figure 5.3, which displays the input prompt and asks the user to select a transition.
3. Lastly, *PlanSimulator* can be used for the programmatic simulation of a system trace. The simulation is launched with a list of commands that help to

orchestrate the execution. The command list contains information about time advances and port setting actions. Additionally, the commands specify which transition to take, in case non-determinism is encountered. Thus, the `PlanSimulator` can be even used to precisely specify a certain system evolution and chosen behaviour. This capacity is highly useful for unit testing or the analysis of specific execution schedules. The definition of long and complex execution plans can quickly become overwhelming though, especially for large systems. Usually, execution plans are therefore machine generated, e.g. in combination with state space exploration or formal verification (Chapter 6).

```
Terminal
$ > python SimulateHeater_Interactive.py

Non-Determinism detected
There are multiple enabled transitions in entity: HeatModule
(Current time: 30 -- Current automaton state: on)

Choose one of the following transitions by entering the according number:
0 ... to_error (transition to 'error')
1 ... to_off (transition to 'off')

Other commands:
r ... choose a transition randomly
p ... plot the system
pe ... plot the entity in which non-determinism occurs
q! ... to exit the script (not recommended in Jupyter mode)

Any other input will be interpreted.
This means you can use it to e.g. inspect ports values.
The entity HeatModule is bound to the variable entity.
Example: entity.my_port.value will print the value of port my_port.

Your choice:
```

Figure 5.3 – Command line prompt signalling the encounter of a non-deterministic situation and asking the user to select an enabled transition.

5.3.2 Calculating the Next Behaviour Change Time

As briefly introduced in Chapter 4, the simulation of CREST relies on the discovery of *ntt*, i.e. the time until the next discrete behaviour change will occur. `crestdsl`'s approach to discovering the smallest *ntt* is built upon the creation of constraint sets that express the behaviour change conditions and the use of an SMT solver to find a minimal solution to these constraints.

The SMT approach that is used in `crestdsl` is only one of several ways to calculate the point in time when a transition can happen. Alternative ways include the discretisation of time and iterative search until the conditions for a transition are discovered. The problematic with this process is to find a “good” step size. Naive techniques use static step sizes and decrease the size when they come

close to a transition time. More elaborate tools employ numerical solvers based on Euler or Runge-Kutta methods to adapt their calculations. Research effort has also resulted in the development of so-called state event location, whose goal is to detect discrete changes in systems such as ours. Interested readers might refer to [PB96] or [EKP01] for descriptions of such methods.

The difficulty lies in the creation of these constraint sets, since CREST’s dataflow-inspired semantics imply that dependencies between ports have to be considered. `crestdsl` solves this problem by fully automatically analysing the source code of a system and creating constraint sets for all updates, transitions and influences that could cause a behaviour change (such as e.g. enable a transition).

When searching for the next behaviour change time of the `HeatModule`, for instance, it is necessary to evaluate all potential behaviour changes, including the transition from `On` to `Off`, which is modelled using the update shown in Listing 5.10.

Listing 5.10 – The `HeatModule`’s `to_off` transition

```

1 @crest.transition(source=on, target=off)
2 def to_off(self):
3     return self.switch.value != "on" or self.timer.value >= 30 \
4         or self.electricity.value < 200

```

The transition guard states that the heat module shuts off when either the switch is not `"on"`, the `timer`’s value reaches or exceeds 30 or the `electricity` drops below 200 watts. Thus, an SMT solver should check if constraint c_1 can be solved with the passing of time:

$$c_1 := (\text{switch} \neq \text{"on"}) \vee (\text{timer} \geq 30) \vee (\text{electricity} < 200)$$

The values 0, `"on"` and 200 in constraint c_1 are constants defined for the respective *theories*⁷. To increase the legibility, this section omits the variables’ annotation with theories, since they are implicit and can be looked up in the CREST diagram in Figure 5.2. `crestdsl`’s implementation obviously asserts that variables such as `switch` are defined using their correct data types.

Looking at the CREST diagram in Figure 5.2, we see that the three sub-expressions are based on the values of two input ports (`switch` and `electricity`) and one local port (`timer`). Since `HeatModule` is the system’s root entity, input port values are not modified by any domain concept (neither by updates, nor influences, nor subentities). `timer` on the other hand is continuously modified by the `increase_timer` update function. It is therefore necessary to add its functionality to the constraint set. From the update function’s source code (see Listing 5.11) we can see that when executing, the function reads the current timer value and adds `dt`, i.e. the amount of time since the update’s last execution, to it. Therefore, the next constraint is

$$c_2 := \text{timer} = \text{timer}_0 + \delta t$$

⁷A theory is an axiomatisation of a data type (e.g. integer, real, string) for SMT solvers.

where $timer_0$ and $timer$ are the timer’s values before and after `increase_timer`’s execution, respectively.

Listing 5.11 – The HeatModule’s `increase_timer` update function

```

1 @crest.update(state=on, target=timer)
2 def increase_timer(self, dt):
3     return self.timer.value + dt

```

As the system has no further dependencies, the final step is to assert that the “source” ports of this dependency analysis are linked to their current port values. This means that any variables that are not modified at runtime (e.g. by an update or influence) are mapped onto their current values. The constraints $c_3 - c_5$ express these constraints by linking `switch`, $timer_0$ and `electricity` to the values shown in Figure 5.2. A last constraint c_6 asserts that no negative δt values are considered, since CREST can only advance time, not step back. The resulting set of constraints contains all equations necessary to discover when the `to_off` transition will become enabled:

$$\begin{aligned}
 c_1 & := (\text{switch} \neq \text{“on”}) \vee (\text{timer} \geq 30) \vee (\text{electricity} < 200) \\
 c_2 & := \text{timer} = \text{timer}_0 + \delta t \\
 c_3 & := \text{timer}_0 = 0 \\
 c_4 & := \text{switch} = \text{“on”} \\
 c_5 & := \text{electricity} = 200 \\
 c_6 & := \delta t \geq 0
 \end{aligned}$$

By explicitly setting all “leaf” values in $c_3 - c_5$ except for δt , the constraint set asserts that δt is the only free (independent) variable that can be modified. Using a constraint solver, such as the Z3 Theorem Prover [DB08], it is possible to find a solution to this set of constraints. In fact, there exist infinitely many solutions to this problem, as any $\delta t \geq 30$ results in possible a solution of the equations. By using Z3’s *optimization* functionality it is possible to specify that we would like to obtain the minimum value for δt that satisfies the solution – in this example, the minimal next transition time is at $\delta t = 30$.

Choosing an SMT Solver An evaluation of the available range of SMT solvers resulted in the choice of the Z3 Prover. This is justified by numerous favourable features. First and foremost, Z3 is being actively developed and improved. To this extent, it regularly participates in competitions such as SMT-COMP 2018 [Hei+]. Z3 is also freely distributed under the MIT license and its source code is openly available⁸. From a technical point of view, it supports many built-in solvers and optimisers for various theories such as strings, integer, real and floating point numbers and it supports the definition of custom data type theories. The programmatic use is intuitive, due to its native API bindings for many modern programming languages, including C++, Python, and Java. Finally, Z3 convinces through its active community and good documentation.

⁸<https://github.com/Z3Prover/z3>

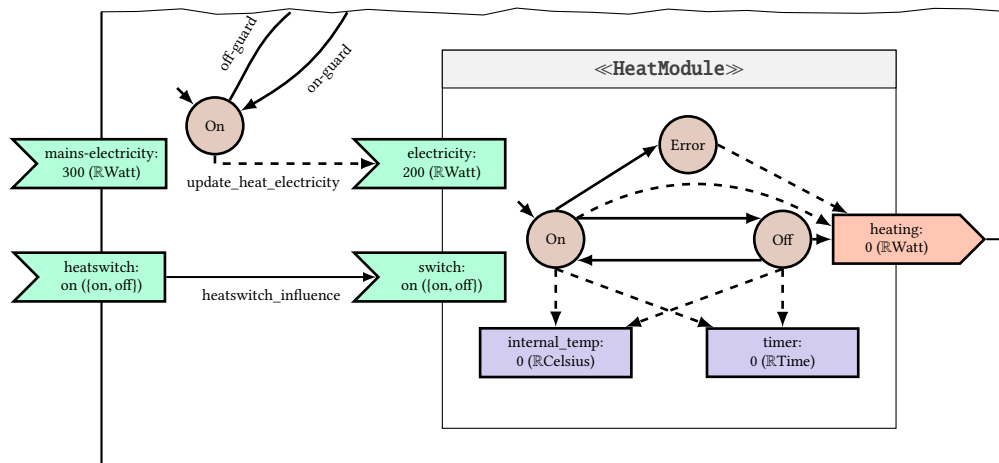


Figure 5.4 – CREST diagram of a GrowLamp system with a HeatModule. The rest of the GrowLamp system and some HeatModule details are not shown.

Complex Constraint Sets (Modifier Propagation) When simulating larger, composed systems, it is likely that a transition depends on a complex web of influences and update functions. In the heat module example, we see that the outcome of the *to_off* transition guard depends on the switch input value and the timer’s value, which is written by the *increase_timer* update. If this module is, however, part of a larger system, the constraint set has to change too, to reflect this change in the dependency structure. Figure 5.4 displays a part of a growing lamp’s CREST diagram, focusing its connection to the HeatModule. As can be seen, in this situation it is necessary to treat the HeatModule’s switch and electricity inputs differently, since these ports depend on the GrowingLamp’s influences and updates.

For example, the switch port is modified by the *heatswitch_influence*, which reads the *heatswitch* port and writes it to the *switch* input. Hence, to find the next transition time, it is necessary to add the dependencies of all “active modifiers”, such as updates linked to currently active automaton states and influences from all parent and subtentities into account. The electricity input is calculated by the *update_heat_electricity* update function, whose source code is shown in Listing 5.12. Note that the GrowLamp’s electricity input has been renamed to *mains_electricity* in Figure 5.4 for clarity.

Listing 5.12 – The GrowingLamp’s update which sets the HeatLamp’s electricity input

```

1 @update(state=on, target=heatelement.electricity)
2 def update_heat_electricity(self, dt):
3     # the heatelement gets the rest
4     return self.mains_electricity.value - 100

```

The new set of constraints for finding the minimum δt that will activate the *to_off* transition is shown below. Note that the constraints c_4 and c_5 have been replaced by the new constraints c'_4 and c''_4 , and c'_5 and c''_5 , respectively. When the constraints are handed to an optimising constraint solver such as Z3, it will find that the minimal solution is also at $\delta t = 30$.

$$\begin{aligned}
c_1 & := (\text{switch} \neq \text{"on"}) \vee (\text{timer} \geq 30) \vee (\text{electricity} < 200) \\
c_2 & := \text{timer} = \text{timer}_0 + \delta t \\
c_3 & := \text{timer}_0 = 0 & c'_5 & := \text{electricity} = \text{mains-electricity} - 100 \\
c'_4 & := \text{switch} = \text{heatswitch} & c''_5 & := \text{mains-electricity} = 300 \\
c''_4 & := \text{heatswitch} = \text{"on"} & c_6 & := \delta t \geq 0
\end{aligned}$$

Condition Behaviour Changes Due to `crestdsl`'s use of standard Python syntax, discrete behaviour changes can also be modelled using conditional (if-then-else) statements. An example for this behaviour is shown in Listing 5.13. It can be seen, that there are two possible behaviours provided by this function, depending on the value obtained from the calculation in Line 3. The function models that the value of `timer` is continuously decreased as long as the value after the update is greater than zero ("else"-branch). When `new_value` reaches or drops below zero, the function switches its output and continues executing the "then"-branch of the condition statement. Thus, there are two different continuous behaviours merged in this update.

Listing 5.13 – The HeatModule's `off_update_timer` update function

```

1 @crest.update(state=off, target=timer)
2 def off_update_timer(self, dt):
3     new_value = self.timer.value - 2 * dt
4     if new_value <= 0: # don't drop below 0
5         return 0
6     else:
7         return new_value

```

Since conditional statements are a vital aspect of Python's syntax, it is essential to support these discrete behaviour changes. However, it is also necessary to analyse if an update function can experience a discrete behaviour change due to the passing of time. Assuming for instance a current timer value of 15, the timer value of Listing 5.13 is decreased linearly for 7.5 time units until it reaches zero. Subsequently the function switches to executing the then-branch (i.e. Line 5). Hence, the simulation should take this effect into account to avoid calculations that use wrong port values (e.g. negative timer values) and misleading simulation results. The calculation algorithm for `nextbehaviourchange_time` therefore needs to perform two steps:

1. analyse which of the branches is currently active, and
2. evaluate if a switch to the other branch can happen by any positive `dt`-value

To solve the first task, the update's source code is analysed and converted, similarly to the approach before. The resulting constraint set is then capable of evaluating the result of the condition. For instance, under the assumption that the timer's cur-

rent value is 15, the constraint set for the evaluation of the if-condition is:

$$\begin{aligned} c_1 &:= \text{timer}_0 = 15 \\ c_2 &:= \text{new_value} = \text{timer}_0 - 2 * \delta t \\ c_3 &:= \text{new_value} \leq 0 \\ c_4 &:= \delta t = 0 \end{aligned}$$

Note that c_4 deliberately sets $\delta t = 0$, to evaluate which branch is currently active (i.e. before time passes). In case the constraint solver finds a solution, this means that the then-branch of the condition statement is active, otherwise the current system state triggers the else-branch. In the example above, the constraints cannot be solved, indicating that the else-branch is active.

Using this knowledge, it is possible to replace c_4 with $c'_4 := \delta t \geq 0$ and rerun the evaluation to see if there exists a δt such that the other branch is chosen. Z3 finds a solution for this new set of constraints with minimal $\delta t = 7.5$. This means that the update function's behaviour changes after 7.5 time units.

5.3.3 Limitations

The current implementation of `crestdsl` has certain limitations in terms of performance and behaviour that cannot be expressed. Below, three of these limitations are discussed in detail.

Infinitesimal Time Advances When looking at the condition in Line 4 of Listing 5.13, we see that the constraint is modelled as `new_value <= 0` to return zero whenever the value is at or below zero. From a logical point of view, the expression of the condition might equally be implemented as `new_value < 0` to “return to zero, if the value drops below”, as shown in Listing 5.14. The problem with this kind of implementation is, that in situations where `timer`'s value is exactly zero – and is hence the else-branch is still active – the required time advance to modify the constraint is infinitesimally small (i.e. ϵ). Thus, after an infinitely small time advance, the update will switch to the then-branch. Both Z3 and `crestdsl` implement a pragmatic solution to this problem in the form of a configurable ϵ value, that expresses a hard-coded value for the minimal expressible time delay⁹.

Listing 5.14 – A problematic, alternative implementation of `off_update_timer`

```

1 @crest.update(state=off, target=timer)
2 def off_update_timer(self, dt):
3     new_value = self.timer.value - 2 * dt
4     if new_value < 0: # return to 0
5         return 0
6     else:
7         return new_value

```

⁹ `crestdsl`'s configurable `epsilon` constant is has a default value of 10^{-10} .

While the assigning of a concrete value to ε provides a practical workaround to the theoretical problem of infinitely small numbers, it does in turn create a severe performance bottleneck. In fact, in the above example `crestdsl` will advance the minimal time possible (i.e. the numeric value assigned to ε) and return to the else-branch, since the timer's value is not smaller than zero any longer.

In this situation CREST's `next_behaviour_change_time` function will analyse the system and again find that the δt required to reach the then-branch is once again ε and advance by its concrete value. The advance within the then-branch resets the timer value to zero, which triggers the cycle again.

As can be seen, this workaround implementation prevents the code from performing impossible behaviour (i.e. advancing infinitely small time units), but leaves the code to advance in ε -sized timesteps.

The red line in Figure 5.5 shows a visual representation of this behaviour. Note how the ε -transition is taken five times and the value always returns to zero. However, since the value of ε in the simulator is chosen very small (10^{-10} , by default), there are too many iterations necessary to advance time efficiently. One possibility to overcome this issue would be to adjust the value of ε to a larger number. “Good” values depend on the target system and domain, and its system state, so that it cannot be automatically inferred.

Instead, `crestdsl` chooses a heuristic approach. If repeated ε -transitions are observed to be caused at the same location (e.g. the same if-then-else-condition), `crestdsl`'s simulator presumes to have encountered this execution pattern. Thus, after a certain limit (e.g. five) of ε -advances for the same behaviour change, this specific source of ε -advances is ignored until a non- ε -advance is triggered in the system. This behaviour is shown by the blue, dashed line in Figure 5.5.

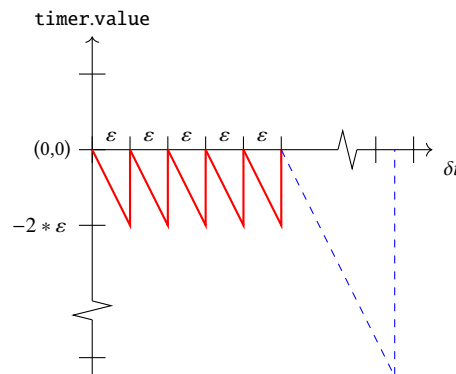


Figure 5.5 – Schematic representation of a heuristics to remove ε -sized step advances

Other modelling tools and platforms often offer advanced means such as so-called *zero-crossing detection* functions for the implementation of the above described behaviour. At the moment, `crestdsl` is implemented in plain Python. This means that such useful functions cannot be accessed directly. The provision of zero-crossing detection and similar functions is however considered an important future work to further increase the usability of the language.

Wrongly Identified Behaviour Changes `crestdsl` does not perform any semantic evaluation of the code and its exposed functionality. Hence, the creation of a constraint set for the code in Listing 5.15 will cause `crestdsl` to believe there is a change in behaviour when a clock port exceeds the value of 200, even though both branches perform the same calculation. It is up to the model developer to avoid such clumsy implementations which might hinder performance.

Listing 5.15 – Conditional code leading `crestdsl` to assume behaviour changes.

```

1 @crest.update(source=..., target=...)
2 def unnecessary_condition(self, dt):
3     if self.clock.value <= 200:
4         return self.clock.value + dt
5     else:
6         return self.clock.value + dt

```

Non-linear Optimisation The Z3 SMT Solver limits the types of its usage scenarios to linear optimisation problems. Hence, the optimisation of non-linear functions cannot be performed in `crestdsl` using the default setup.

An example for such a system is the well-known “Three Masses” [Car+06] case study, which models three point masses moving across and then falling off a surface, e.g. a table. The goal of this system is to model horizontal and vertical position, velocity and acceleration of these masses as they fall and bounce repeatedly off the ground. As an object’s velocity depends on its acceleration over time, and its position changes according to its velocity over time, the function for determining the mass’s position is non-linear.

Z3 is however capable of finding non-optimal solutions for such constraints. In fact, in the presence of non-linear functions, `crestdsl` produces a warning and uses Z3 to find a non-optimised solution for the constraint set. Though the produced results cannot be guaranteed to be correct, many times the `crestdsl` simulator manages to find useful results. The `crestdsl` model of the Three Masses system has been implemented as proof-of-concept. Its source code is provided in Appendix C.3.

5.4 Tool Implementation & Architecture

Evidently, the development and simulation of `crestdsl` models in standalone Python is not ideal. As convenient as Python classes and functions are for the model creation, CREST diagrams are naturally more comprehensible. Therefore, `crestdsl` provides APIs for the visualisation of models. An initial approach led to the creation of a plotting library based on the *Graphviz*¹⁰ software package for graph visualisation. Graphviz implements the *dot* language for the specification of graphs using textual syntax and several rendering engines to produce graphs according to various layout algorithms. Figure 5.6 shows the HeatModule rendered using Graphviz’s *dot*-engine.

This module allows for important visual representation and inspection of `crestdsl` models. Evidently though, switching between a Python IDE (for model editing), a

¹⁰<https://www.graphviz.org/>

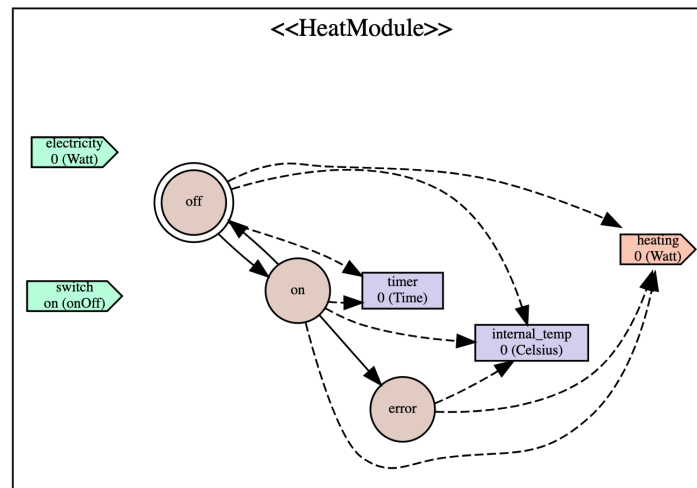


Figure 5.6 – Graphviz rendering of the HeatModule’s crestdsl model.

Python runtime (for model execution and graph creation) and an image viewer (for the graph display) leads to an inconvenient workflow.

The solution to this problem is to improve interactivity by using Project Jupyter¹¹, an interactive and browser-based Python runtime. Jupyter has recently become popular within the scientific community as a means for the exchange of data, computation and results. To this extent, Jupyter bundles code within so-called *notebooks*. Each notebook is an individual file that contains software source code and documentation. Inside notebooks, the code is split into individual *cells*, which can be executed and produce output. Depending on the cell’s type, different execution engines are available. For example, Python cells execute source code using an interactive Python interpreter, while Markdown cells are transformed to HTML and displayed, e.g. for documentation purposes.

Jupyter’s popularity also led to the creation of many extensions and plug-ins that allow the interpretation of other programming languages, the display of different file types, etc. Other significant development projects work towards facilitating the sharing of entire Jupyter environments, so that dependencies can be directly resolved. One of the most notable is Binder¹², which allows for example the entire crestdsl runtime environment to be shared online, so that users access crestdsl directly from within their browser, without the need of installing dependencies such as required Python libraries or the Z3 SMT solver. A demonstration of this feature is available online at crestdsl’s webpage (<https://crestdsl.github.io>).

5.4.1 Interactive Visualisation

The above mentioned Graphviz visualisations offer a limited way to render crestdsl models into CREST diagrams. The drawback of the static image output is that the produced CREST diagrams can become very complex and confusing when drawing

¹¹<https://jupyter.org/>

¹²<https://mybinder.org/>

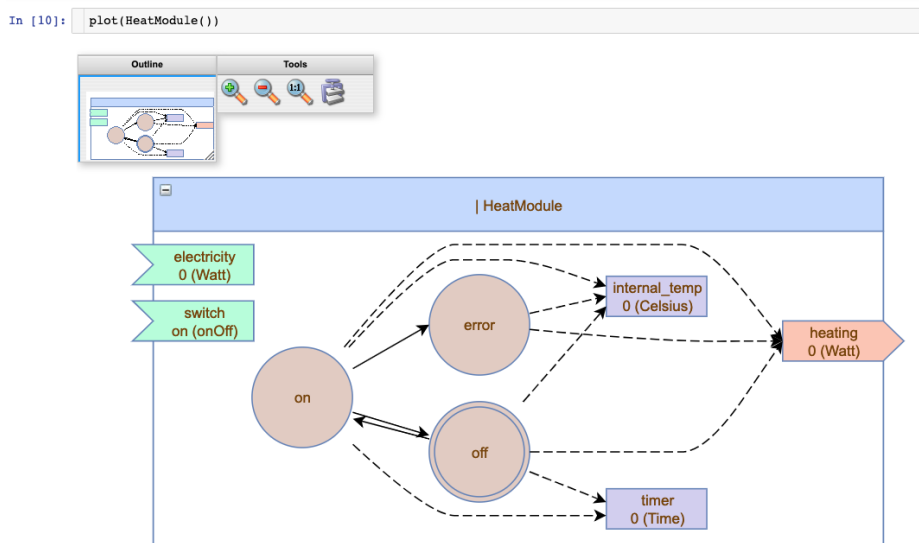


Figure 5.7 – Interactive CREST diagram within a Jupyter notebook

models with a large number of ports, states, updates and subentities. Another handicap is that CREST diagrams only contain the function names of updates, influences and transition guards. The detailed behaviour of these functions has to be looked up in the actual `crestdsl` source code. This “distance” can pose certain constraints on the exploration and debugging capabilities of `crestdsl` models.

To overcome these constraints, `crestdsl` provides another visualisation tool that allows the creation of interactive, HTML-based CREST diagrams. The functionality is based on the `mxGraph`¹³ and `elk.js`¹⁴ libraries. `mxGraph` is a JavaScript diagramming library, that offers reactive graph drawing capabilities. `elk.js` on the other hand is a JavaScript implementation of the Eclipse Layout Kernel¹⁵, which implements highly customisable diagram and graph layout algorithms.

This plotting feature is implemented in the `crestdsl.ui` module. By calling the `plot` function and defining an entity as parameter, a HTML-rendering of a CREST diagram is produced, as shown in Figure 5.7. The diagram is reactive and allows several forms of interaction, such as moving of objects on the canvas and zooming into certain parts of the diagram. Hovering the mouse cursor above any node (e.g. states, ports) or edge (e.g. transitions, updates) will display a tooltip, that shows the object’s name and additional information (e.g. a transition’s source and target states, or a port’s non-rounded value). Double clicking on any edge opens a modal pop-up window and displays its Python code, e.g. a transition’s guard function. One of the most important features is the collapsing and expanding of subentities. This allows to focus on the essential parts and unimportant details. Figure 5.8 shows some screenshots of these features.

¹³<https://jgraph.github.io/mxgraph/>

¹⁴<https://github.com/OpenKieler/elkjs>

¹⁵<https://www.eclipse.org/elk/>

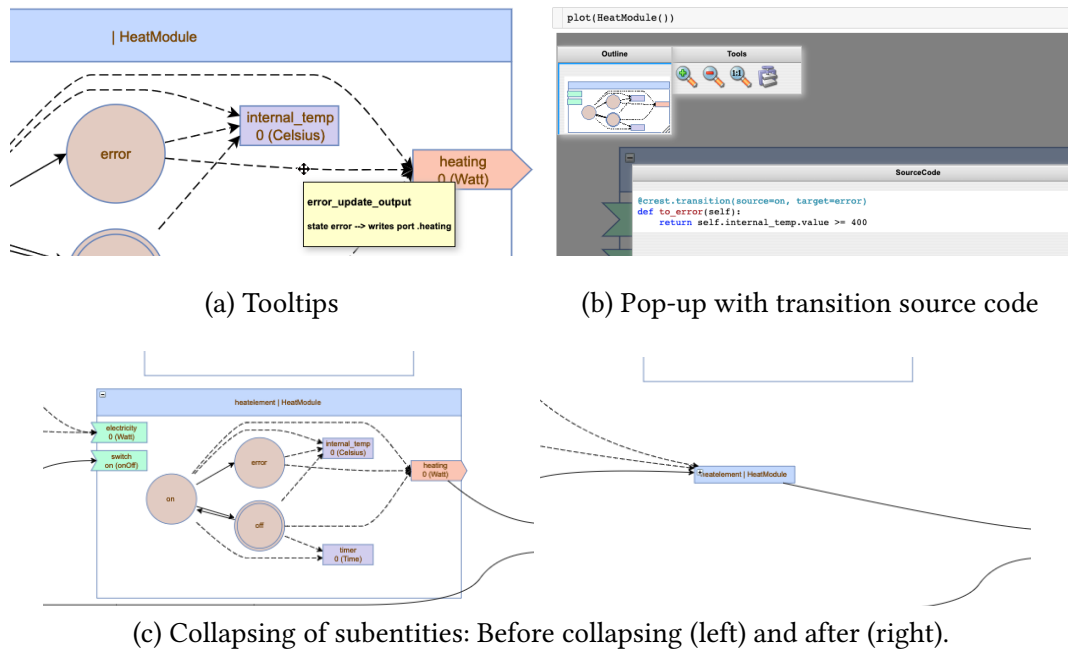


Figure 5.8 – Screenshots of interactive CREST diagram functionality

5.4.2 Trace Plotting

The use of Python as host language and Jupyter as interactive development and execution environment provides further advantages, such as the integration of various, modern Python libraries. For example, the Python Data Analysis Library (short: *pandas*)¹⁶ is an intuitive data analysis framework, whose powerful `DataFrame` class is used by `crestdsl` for the recording of system traces. Jupyter notebooks conveniently integrates the native display of dataframes, as shown in Figure 5.9.

Building upon `pandas`, several other frameworks provide convenient functionality for the further analysis, exploration and integration of data. *Plotly*¹⁷, for example,

¹⁶<https://pandas.pydata.org/>

¹⁷<https://plot.ly/python/>

```
In [10]: sim.trace.data
```

```
Out[10]:
```

	timestamp	HeatModule	HeatModule.timer	HeatModule.switch	HeatModule.heating	HeatModule.internal_temp	HeatModule.electricity
0	0	HeatModule.off	0.0	on	0.0	0.0	0.0
1	0	HeatModule.off	0.0	NaN	0.0	22.0	NaN
2	0	HeatModule.off	0.0	NaN	0.0	22.0	NaN
3	0	HeatModule.off	0.0	NaN	0.0	22.0	NaN
4	0	HeatModule.on	0.0	NaN	137.5	22.0	NaN
5	0	HeatModule.on	0.0	on	137.5	22.0	275.0
6	30	HeatModule.on	30.0	NaN	137.5	247.0	NaN
7	30	HeatModule.off	30.0	NaN	0.0	247.0	NaN
8	30	HeatModule.off	30.0	on	0.0	247.0	275.0
9	30	HeatModule.off	30.0	on	0.0	247.0	275.0

Figure 5.9 – Native display of a `crestdsl` system trace as `pandas` dataframe

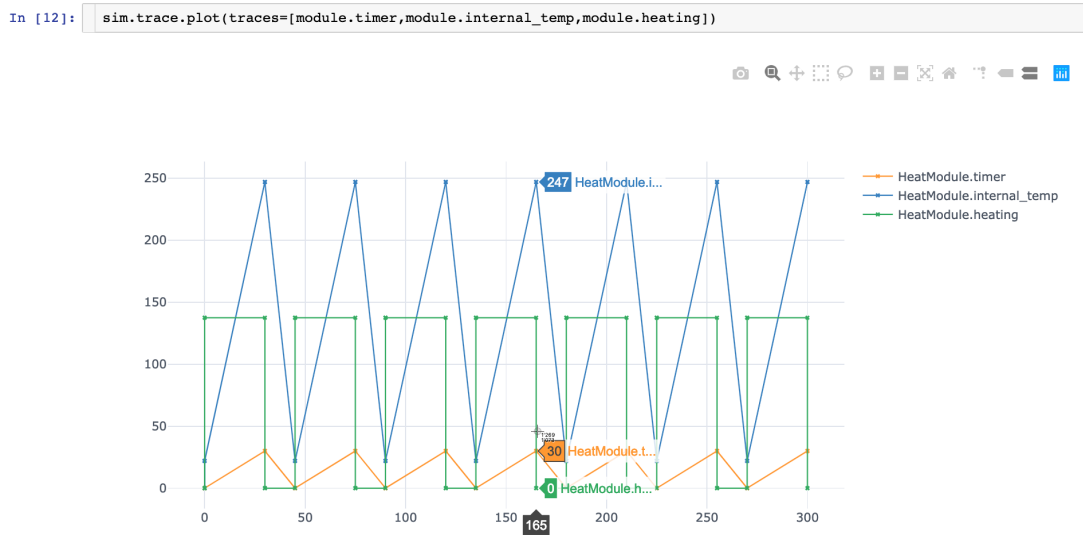


Figure 5.10 – Creation of an interactive plot of a `crestdsl` system trace

is a graph plotting library for Python that integrates its output directly into Jupyter notebooks and produces interactive charts that can be explored by scaling and zooming, or exported to image files. Figure 5.10 shows the graphical output produced by feeding the `pandas` dataframe of Figure 5.9 into the `Plotly` library.

5.5 Summary

This chapter presents `crestdsl`, an implementation of CREST as internal DSL within the Python host language. It highlights the convenient and user-friendly approach to defining CREST models using standard Python objects and classes, and masking the complexity of executable model creation and validation inside a software library. The library also provides APIs for the simulation and validation of `crestdsl` models. These features rely on `crestdsl`'s implementation of the `next_transition_time` function, as required by CREST's formal semantics. To implement this complex analysis, an algorithm was developed to extract a set of constraints that can be solved by an off-the-shelf SMT solver, such as the Z3 Prover.

The use of Python as a host language provides even further advantages by allowing the native integration of the many open source Python libraries and tools. Most notable is `crestdsl`'s integration into the Project Jupyter environment, which helps users create and intuitively interact with CREST models directly inside their browser. Additionally, this solution creates varied and handy extension possibilities, such as `crestdsl`'s use of an HTML- and JavaScript-based display to draw interactive CREST diagrams, which can be easily explored and analysed, and the integration with common Python frameworks, such as the `pandas` data analysis and `Plotly` graph plotting libraries.

Chapter 6

Verification

The simulation of CREST models provides significant value to users by allowing them to explore their system's evolution and answering “*What happens if ...*”-questions. Simulation is however not an efficient method when it comes to testing whether a system can reach a certain system configuration at all. For this task it is more appropriate to use formal verification methods such as model checking [CGP99]. These techniques can provide answers to questions such as “*Can the heating be on, while the system is in state off?*” or “*Is it possible that the temperature output exceeds 40 degrees?*”. More generally, model checking evaluates whether, given an initial system configuration, a system can evolve to reach a certain state (e.g. a deadlock) or follow a specified sequence of states (i.e. a system trace). CREST's formal structure and semantics create the possibility to apply such formal verification to its models.

Formally, model checking refers to the activity of asserting that a model satisfies certain *properties*. Properties are formal descriptions of a system's states or traces and expressed using temporal logic formulas [AH92]. Two commonly used temporal logic representatives are the linear temporal logic (LTL) [Pnu77] and the computation tree logic (CTL) [CE81]. These logic languages are quite similar – both express formulas over a system's *state space* (i.e. traces of its state evolution). Their difference lies in the details of their view of system evolutions. LTL operates on linear system traces (i.e. one particular evolution), while CTL analyses so-called *branching* system evolutions. This branching logic creates graphs where each system state can be succeeded by several others, as is commonly the case in non-deterministic systems. This means that in CTL, formulas can be used to check if some properties hold in all possible evolutions of the system, whether there exists at least one system trace where some properties hold or even combinations of those (e.g. if there is one evolution that leads to a state where all further evolutions satisfy a property). This capacity makes CTL it a good choice for the verification of CREST systems, where the goal is to assert that some properties are always (or never) satisfied, or whether the system can reach a certain configuration in at least one evolution.

Germination System Example This chapter uses a seed germination system to illustrate the formal verification of CREST models. The system contains two germination boxes, whose purpose is to create a warm and moist environment

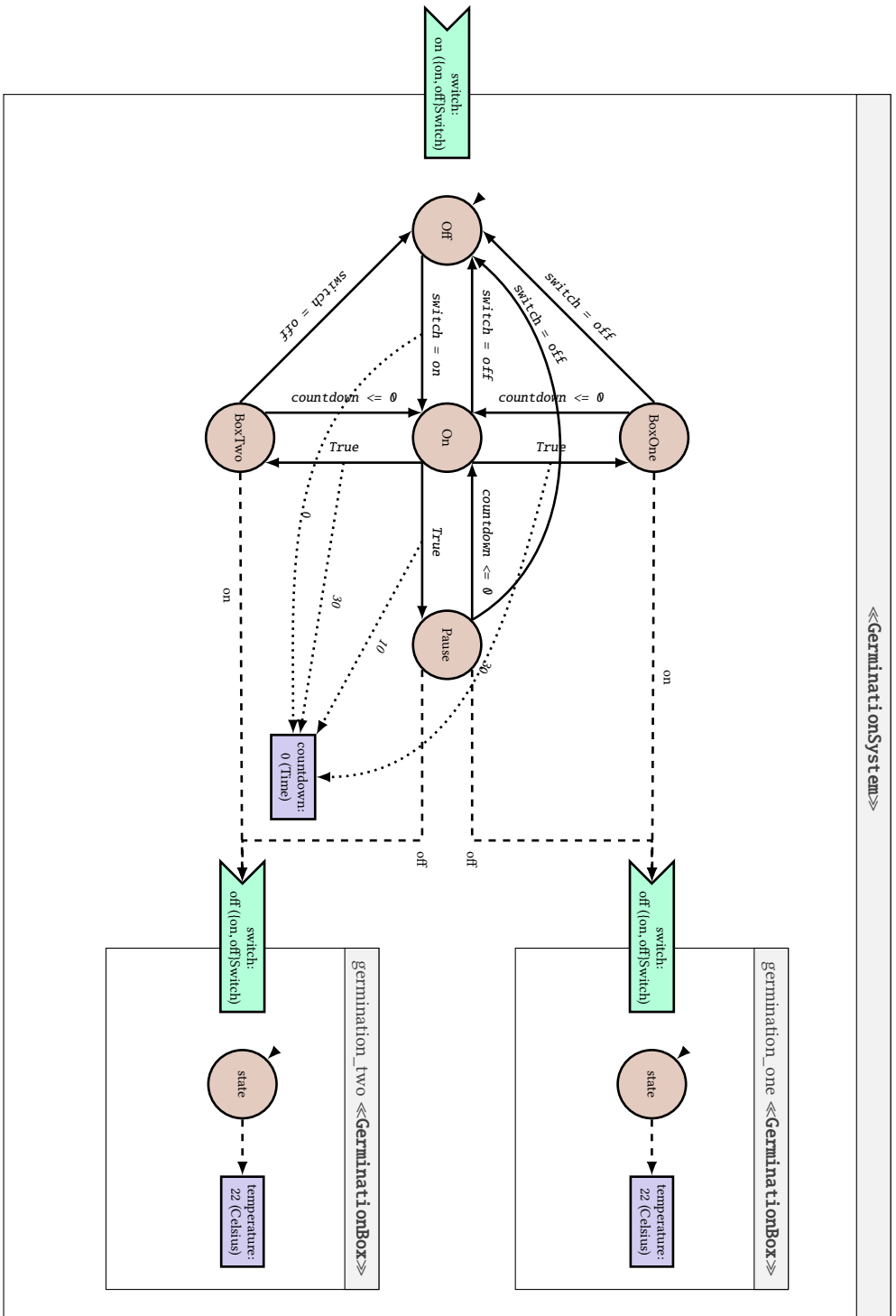


Figure 6.1 – The temperature aspect of a plant growing system with two germination boxes. For simplicity, transitions, actions and updates are annotated with the functionality directly, rather than the function names. Some updates are not drawn, to increase legibility and give focus on more important functionality. For example, the states **BoxOne**, **BoxTwo**, **On** and **Pause** all have an update function that progressively reduces the value of the countdown timer until it reaches zero.

for the germination of plant seeds. The CREST model shown in Figure 6.1 was created to help the analysis of the system's properties. This model focuses primarily on the temperature evolution within the boxes. Hence, the electricity aspect is ignored. Further, the temperature evolution over time is assumed to be linear. Each germination-box is modelled as an entity having one input port (an on/off switch), a single automaton state and a local port (temperature). The local port is set by an update function that is continuously executed. Its functionality is defined as follows: If the switch is in state `on`, the update increases the temperature value by 0.5 degrees centigrade per time unit (minute). We assume that temperature cannot exceed 40°C. Otherwise, i.e. if the switch is `off`, the update models the gradual temperature decrease by 0.25°C per minute, until reaching the room temperature of 22°C.

The entire system is modelled as an entity that contains two germination boxes. It has one on/off switch input that controls the behaviour automaton. Further, the system has a digital countdown timer that controls the evolution of the entity's state automaton. Once the countdown is set, its value continuously decreases until it reaches zero. When the switch is set to `on`, the system switches immediately to state `On` and resets the automaton (`countdown = 0`). When the switch is set to `off`, the system transitions to state `Off`, independent of the state it currently is in. From the `On` state, the system can transition to three potential states: `BoxOne`, where the first germination box is heated, `BoxTwo`, to turn on heating in the second box, and `Pause` to not heat either box. When transitioning to a `Box`-state, the countdown value is set to 30, when moving to `Pause` the value is set to 10. In this system the choice of which state to transition to is random.

In each state, update functions are used to set the two boxes' switch inputs. In state `BoxOne`, `germination_one`'s switch is set to `on` and the other is turned off. In state `BoxTwo`, `germination_two` is heated and the first one is off. In all other states (`Pause`, `On`, `Off`) both boxes are not heating (i.e. their switch is `off`).

CTL Model Checking CTL formulas consist of atomic propositions (APs) and operators that connect them. APs are (Boolean) predicates that express whether a given system configuration satisfies a property. For CREST systems, we can mainly distinguish two types of APs that are of interest: *state checks* and *port checks*. State checks express that a CREST entity is in a given state (e.g. “*The germination system is in state On*”) and port checks compare the value of one port to either a constant or another port's value (e.g. “*The countdown timer value is smaller than 5 minutes*”, “*The temperature in germination_two is always lower than the temperature in germination_one*”).

As known from other logic systems (e.g. propositional logic) *logic operators* such as conjunction (\wedge), disjunction (\vee) and implication (\Rightarrow) can be used to combine APs to more complex expressions. In addition, CTL allows the specification of system state evolution using *temporal logic operators*. For instance, the operator *AG* expresses that a formula is valid in all future states of all possible system evolutions. Thus, $AG\phi$ states that ϕ will always be valid, independent of the chosen transitions and time advances. All CTL operators are of the form QT , where Q is a quantifier and T is a

Table 6.1 – Common CTL operators and their meanings.

Operator	Meaning
$EX\phi, AX\phi$	ϕ will be valid in the next state
$EF\phi, AF\phi$	ϕ will be valid in some successor state
$EG\phi, AG\phi$	ϕ is valid in this state and all future states
$\phi EU\psi, \phi AU\psi$	ϕ will be valid (at least) until ψ becomes valid

“E” operators require the condition to be valid in one evolution path,
 “A” operators need to be satisfied in all possible evolutions.

temporal specification. The quantifier states whether a formula should be valid in (at least) one evolution path (“E” operators) or in all evolution branches (“A” operators). The temporal part specifies whether the formula should be valid in the next system state (“next”; X), in some future state (“finally”; F), in every future state (“globally”; G) or until another formula becomes satisfied (“until”; U). Table 6.1 shows a set of common CTL operators alongside their meaning.

Evidently, the definition and efficient use of CTL formulas is a complicated matter, even for CPS creators familiar with modelling and simulation techniques. Especially newcomers to the systems engineering world might struggle understanding and creating valid CTL formulas for the verification of their systems. This means, that despite the formalism’s theoretical usefulness for the verification of CREST systems, system creators might need help during this development phase. Thus, the implementation of `crestdsl` provides understandable APIs that can be conveniently used by unfamiliar users for the quick verification of their systems. Additionally, the help to these users can be further extended, as these APIs can be easily customised and extended to further increase usability. Details for this feature are described in Section 6.3.

Definition 13 (Formal CTL Syntax). Formally, CTL formulas can be constructed inductively from the following set of operators and a set of atomic propositions AP .

$$\phi = \text{True} \mid p \mid \neg \phi \mid \phi \wedge \phi \mid EX\phi \mid \phi EU\phi \mid \phi AU\phi \quad \text{where } p \in AP.$$

A range of other CTL operators has been created over the years (e.g. EF, AG). However, they all can be expressed as combinations of the operators above (e.g. $EF\phi = \text{True } EU \phi$ and $AX\phi = \neg EX\neg\phi$) [Bur+92].¹

Commonly, CTL formulas are evaluated on so-called *Kripke structures* [BCG88]. A Kripke structure is a form of directed graph, consisting of system states (graph

¹In other publications the EU/AU operators are sometimes written using a different notation. The formulas $\phi EU\phi$ and $\phi AU\phi$ are alternatively denoted as $E\phi U\phi$ and $A\phi U\phi$. This chapter uses the former notation, as the difference is only syntactic and the two notations are otherwise equivalent.

nodes), transitions (graph edges) and a labelling function that maps each state to the set of APs that hold in this state. The evaluation of CTL formulas is therefore reduced to analysing whether the Kripke structure contains states (nodes) or paths (sequences of transitions) that satisfy a corresponding trace or pattern.

Usually, during model checking, a provided formula is evaluated for a certain (initial) state of a Kripke structure. There generally exist two methods for the state space exploration: *local* and *global* model checking. The former starts the evaluation at the initial state and explores the graph in a “forward” manner until it encounters an example (i.e. a node that satisfies the formula) for *E*-formulas, or counter-example (for *A*-formulas). An algorithm implementation that follows this approach is described in [VL93]. Global model checking, on the other hand, first labels all nodes of the entire state space with the information which CTL subformula holds or does not hold. In a second step, a general graph search finds all nodes that satisfy the complete CTL formula. Finally, the algorithm checks whether the given initial state is among those states that were found to satisfy the formula in the second step. There have been many publications, discussions and comparisons of the two approaches. The interested reader is referred to [Mer01] and [MSS99]. The rest of this chapter describes a global model checking approach for CREST systems.

6.1 TCTL and Timed Kripke Structures

Given CREST’s hybrid nature, where discrete transitions and continuous time behaviour can be observed, the verification of a CREST system requires the consideration of its time aspect. To that extent, it is often required to express the occurrence (or the lack of it) of certain events before or after some point in time. A possible verification scenario might for instance test whether “*The system never heats the same germination box for more than 30 minutes continuously*”. Several temporal logic systems, including CTL, have been extended to include timing information [Bou09]. The timed extension of CTL resulted in the creation of the TCTL formalism [Hen+94], which can be used to express formulas for the verification of timed automata and hybrid systems [BL08; Alu+95]. TCTL’s timed operators are annotated with a time interval $I \in Interval(\mathbb{T})^2$ that provides a minimum and maximum point in time, between which the respective formula has to be satisfied.³ TCTL formulas can hence express whether certain properties hold within a given time interval.

Definition 14 (Formal TCTL Syntax). The formal syntax of a TCTL formula is provided inductively as follows:

$$\phi = \text{True} \mid p \mid \neg \phi \mid \phi \wedge \psi \mid \phi EU_I \psi \mid \phi AU_I \psi$$

²See Appendix B for a formal definition of time intervals in the time base.

³In fact, several different TCTL definitions have been provided in the literature. For example, [Hen+94] specifies temporal constraints using explicit (external) clock variables and operators for these clocks instead of time intervals. In [BCM05] it is shown that this TCTL variant is more expressive than its interval-based alternative. For this thesis we chose the latter, as it is more intuitive and closer to CREST’s semantics.

where $p \in AP$ is an atomic proposition, and $I \in \text{Intervals}(\mathbb{T})$ an interval in the time base \mathbb{T} that defines the time aspect of the EU_I and AU_I operators.

Notation (Interval notation). This thesis reuses the conventions established by former publications. Thus, we follow our predecessors' practices and use the more easily legible operator notations $\leq b$, $< b$, $\geq a$ and $> a$ instead of the intervals $[0, b]$, $[0, b)$, $[a, \infty)$ and (a, ∞) , respectively. Further, instead of defining the interval $[0, \infty)$, we simply omit the definition of the interval entirely. Hence, $\phi EU_{\leq a} \psi$ is equivalent to $\phi EU_{[0, a]} \psi$ for instance, and $\phi AU \psi$ is interchangeable with $\phi AU_{[0, \infty)} \psi$.

Based on the definitions above, further TCTL operators can be defined to improve TCTL's usability and legibility of the formulas. Common TCTL includes the timed versions of well-known CTL operators such as EF_I and AF_I , EG_I and AG_I . Note that it is not possible to define a TCTL equivalent of CTL's *next* (X) operator, as the continuous time nature of TCTL makes it generally impossible to define which point in time is "the next one".

Timed Kripke Structures Similar to the annotation of operators with temporal validity ranges, it is also necessary to add timing information to the Kripke structures on which the formulas are evaluated. The extension to *timed Kripke structures* is intuitive and performed by annotating each transition edge with a duration $t \in \mathbb{T}$. This duration expresses the amount of time that it takes to transition from one state to another.

Definition 15 (Timed Kripke Structure). Formally, a timed Kripke structure \mathcal{TK} is a quadruple $\mathcal{TK} = (S, \mathcal{T}, \rightarrow, L)$, where S is a set of system states and \mathcal{T} is a time base. $\rightarrow \subseteq S \times \mathcal{T} \times S$ is a left-total, timed transition relation. This means that for each state s there exists a transition $(s, t, s') \in \rightarrow$ that starts in s ⁴. $L : S \rightarrow \mathcal{P}(AP)$ is a labelling function that maps each state to the set of valid APs in that state. As an example, we might look at the timed Kripke structure in Figure 6.2.

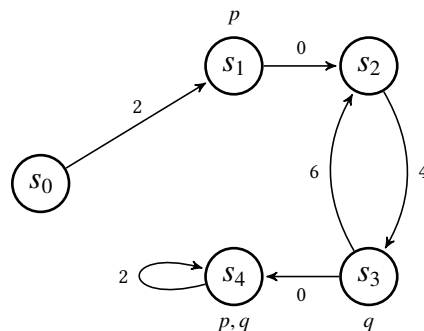


Figure 6.2 – A timed Kripke structure with five system states $S = \{s_0, \dots, s_5\}$. Nodes are annotated with the APs p and q .

⁴If \rightarrow is not left-total, it can be made so by e.g. adding a transition $(s, 0, s)$ to \rightarrow for each state s that does not have another transition

Pointwise vs. Continuous Interpretations The evaluation of TCTL formulas on timed Kripke structures permits two interpretation methods. The first one, *pointwise* interpretation, requires that a formula is valid for all states $s \in S$ of the structure. The alternative *continuous* interpretation considers additionally all implicit system states that are traversed when transitioning from a state s to another state s' ($s, s' \in S$), but are not explicit system states.

These are referred to as *configurations* and identified by their “distance” (i.e. the time interval) $\delta t \in \mathbb{T}$ from their predecessor state s of a transition. Thus, a configuration is denoted as $\langle s, \delta t \rangle \in S \times \mathcal{T}$. Note that each configuration is only identified by its direct predecessor. Hence the δt of any configuration $\langle s, \delta t \rangle$ has to be smaller than the transition time t of any $\langle s, t, s' \rangle \in \rightarrow$, i.e. $0 \leq \delta t < t$. Depending on the time base, there exist infinitely many reachable configurations along any transition whose length is not zero.

The new configuration concept further extends the need for definition of AP labels to configurations. Thus, we define that if an AP holds (resp. does not hold) in a state s , it also holds (resp. does not hold) in all of its configurations $\langle s, \delta t \rangle$. Figure 6.3 depicts three configurations of the continuous interpretation visually.

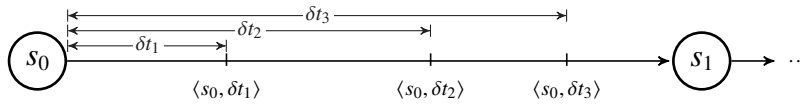


Figure 6.3 – Examples of configurations (i.e. implicit states) that are reachable through time advances.

The pointwise interpretation also permits the definition of configurations, although only configurations of the form $\langle s, 0 \rangle$ are considered.

Difference between Pointwise and Continuous Pointwise interpretation can be seen as a discrete form of model checking, where formulas are only evaluated for specific system states. Continuous interpretation, on the other hand, takes all system states into account, including the ones that are not explicitly represented. Evidently, this creates a semantic difference between these interpretation forms.

The following example highlights this divergence. The formula $\phi = AG_{<2}EF_{=2} p$ states that for all configurations in the interval $[0, 2)$ another configuration is reached after exactly two time units where the property p holds. When evaluating ϕ on state s_0 of the Kripke structure shown in Figure 6.4, the difference between the two interpretations becomes evident. In the pointwise view ϕ is satisfied, as in the interval $[0, 2)$ that follows s_0 only one configuration, namely $\langle s_0, 0 \rangle$, is considered. The formula $EF_{=2} p$ holds for this configuration, since its distance to s_1 is two time units.

When considering the continuous interpretation on the other hand, there exist infinitely many configurations $\langle s_0, \delta t \rangle$ between s_0 and s_1 . The evaluation of $EF_{=2} p$ on any configuration $\langle s_0, \delta t \rangle$ after s_0 (i.e. $\delta t > 0$) “reaches into” the transition from s_2 to s_3 , where p does not hold any longer. For example, evaluating $EF_{=2} p$ on $\langle s_0, 1 \rangle$ requires that p holds in a configuration that is 2 time units after this configuration,

i.e. in $\langle s_2, 1 \rangle$, which is clearly not satisfied. Thus, the formula does not hold in the continuous interpretation. It is easily visible, how a wrongly chosen viewpoint can lead to false verification results. If undiscovered, such behaviour can result in misled confidence in the best case and damage to the system and its operators in the worst.

Generally, the choice between these two interpretation forms is based on the specific a system and property to verify, since the more precise evaluation of formulas in the continuous interpretation requires significantly more computational resources. Thus, in practical applications pointwise interpretation is chosen, if the use case permits it. Continuous interpretation is used if the risk of system failure is high.

Lepri et al. have shown in [LÄÖ15] that the continuous model checking can be reduced to pointwise model checking through creation of a semantically equivalent Kripke structure with an extended state set and altered transitions. The article presents an approach that is based on the splitting of transitions into smaller ones of length $\gamma = \frac{gcd}{2}$, where gcd is the greatest common denominator of the evaluated formula's interval boundaries and all timed Kripke structure transition durations. Through this transformation, the resulting Kripke structure's transition lengths and the formula's boundaries are all multiples of γ , so that all configurations that are reachable by the formula have the form $\langle s, 0 \rangle$. The method further requires some minor alterations of the TCTL formula and its intervals, to ensure that the model checking semantics are preserved. The reader is referred to the original publication for a detailed explanation of the approach, the exact reasoning and the proof of soundness.

6.1.1 Model Checking

The verification of CREST systems is based on the concepts of TCTL model checking defined by other authors. Specifically, this thesis follows the approach of evaluating TCTL formulas on timed Kripke structures, as presented and formalised by Lepri et al. in [LÄÖ15]. This section briefly revisits some essential definitions. Section 6.2 establishes the relation to the verification of CREST systems. Interested and unfamiliar readers are encouraged to review the work of Lepri et al. for a complete explanation.

The pointwise TCTL model checking of timed Kripke structures is based on checking whether properties, specified as operator-combined atomic propositions, hold in a given configuration or sequence of configurations. The satisfaction relation \models is introduced to express that a TCTL formula holds in a given state or configuration.

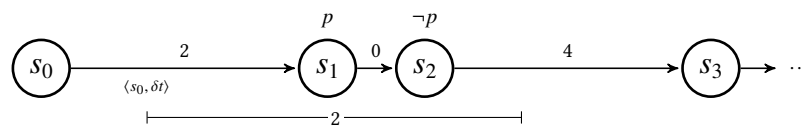


Figure 6.4 – One of the infinitely many configurations $\langle s_0, \delta t \rangle$ in the continuous interpretation, for which $EF_{=2} p$ does not hold.

Definition 16 (Satisfaction relation). The satisfaction relation \models associates a timed Kripke structure \mathcal{TK} and configuration $\langle s, \delta t \rangle \in S \times \mathcal{T}$ to a TCTL formula ϕ so that $\mathcal{TK}, \langle s, \delta t \rangle \models \phi$ expresses that ϕ is satisfied in configuration $\langle s, \delta t \rangle$.

Iff the formula ϕ is satisfied for the state s_0 of the \mathcal{TK} , the statement $\mathcal{TK}, \langle s_0, 0 \rangle \models \phi$ is valid. $\not\models$, the negated form of this relation, implies that a formula does not hold in a given state. The rest of the thesis uses the short notation $\mathcal{TK}, s \models \phi$ for $\mathcal{TK}, \langle s, 0 \rangle \models \phi$. Thus, $\mathcal{TK}, \langle s, 0 \rangle \models \phi \iff \mathcal{TK}, s \models \phi$.

The satisfaction of TCTL formulas depends on the subformulas they are composed of. A formula ϕ that consists only of an AP $p \in AP$, for example, holds in every state $s \in S$ iff the state is labelled with p . Thus:

$$\mathcal{TK}, s \models p \iff p \in L(s)$$

The formula $\phi = p$ is satisfied for the state s_1 of the \mathcal{TK} shown in Figure 6.2, since s_1 is annotated with p . Logic operators within formulas are evaluated in a compositional manner, such that e.g. $\mathcal{TK}, \langle s, \delta t \rangle \models \phi \wedge \psi \iff \mathcal{TK}, \langle s, \delta t \rangle \models \phi \wedge \mathcal{CK}, \langle s, \delta t \rangle \models \psi$.

Next to the verification of these so-called *state* formulas (i.e. formulas that can be verified by only looking at one state alone), TCTL also features *path* operators, potentially annotated with time intervals, such as EU_I , AF_I and EG_I . When model checking a formula ϕ that contains such operators, the satisfaction has to be verified on a sequence of states.

For example, to test the satisfaction relation $\mathcal{TK}, s_0 \models \psi_1 EU_{[5,10]} \psi_2$, we need to assert that in at least one system evolution, ψ_2 becomes valid at c time units, where $c \in [5, 10]$. Additionally, ψ_1 must hold in all states before reaching the time c . Such formulas have to be evaluated on *paths*, rather than individual states.

Definition 17 (Paths and Ticks). Formally, paths are infinite sequences of configurations, linked by *ticks* which are denoted $\xrightarrow{\tau}$, where $\tau \in \mathcal{T}$ is a tick's duration. For example, a path $\pi \in \Pi$ can be represented as $\pi = \langle s, \delta t \rangle \xrightarrow{\tau} \langle s', \delta t' \rangle \xrightarrow{\tau'} \langle s'', \delta t'' \rangle \xrightarrow{\tau''} \dots$

Since, the pointwise interpretation only considers configurations of the form $\langle s, 0 \rangle$, we can use the abbreviated notation $s \xrightarrow{\tau} s' \xrightarrow{\tau'} s'' \xrightarrow{\tau''} \dots$

Further, this means that all ticks are aligned with the respective transitions⁵.

A path for the \mathcal{TK} in Figure 6.2 could for example be the following

$$\pi = s_0 \xrightarrow{2} s_1 \xrightarrow{0} s_2 \xrightarrow{4} s_3 \xrightarrow{6} s_2 \xrightarrow{4} s_3 \xrightarrow{0} s_4 \xrightarrow{2} s_4 \xrightarrow{2} \dots$$

It is easy to observe that some configurations appear repeatedly along the path (s_2 , s_3 and s_4). This behaviour is natural for timed Kripke structures that contain loops. It is however disadvantageous for timed model checking. In fact, the model checking approach is based on the discovery of how much time has passed between two configurations of a path. Thus, it is necessary that any configuration can be

⁵In the continuous interpretation, ticks can connect any two configurations. Thus, each transition can be split into infinitely many different ticks and produce different paths.

uniquely identified and that it is possible to calculate how much time has passed since the path started. In the above example, we can distinguish the first occurrence of s_2 from the second one, since the former appears exactly two time units after s_0 and the next one twelve time units after s_0 .

Definition 18 (Timed Paths and Positions). To uniquely identify a specific occurrence of a configuration alongside a path, it must be annotated with a form of “timestamp”. This timestamp captures the path’s total duration until the configuration. Thus, each configuration within the path is extended to include a value $c \in \mathcal{T}$, such that c is the sum of all preceding tick durations. This means for the path $\pi = \langle s, \delta t \rangle \xrightarrow{\tau} \langle s', \delta t' \rangle \xrightarrow{\tau'} \langle s'', \delta t'' \rangle \xrightarrow{\tau''} \langle s''', \delta t''' \rangle \xrightarrow{\tau'''} \dots$ the timestamp value of $\langle s'', \delta t'' \rangle$ is $c'' = \tau + \tau'$. The *timed path* is the annotated version of the path and written as

$$\langle s, \delta t \rangle @ c \xrightarrow{\tau} \langle s', \delta t' \rangle @ c' \xrightarrow{\tau'} \langle s'', \delta t'' \rangle @ c'' \xrightarrow{\tau''} \langle s''', \delta t''' \rangle @ c''' \xrightarrow{\tau'''} \dots$$

where $c = 0$, $c' = \tau$, $c'' = \tau + \tau'$, $c''' = \tau + \tau' + \tau''$, etc.

Formally, a timed configuration is a triple of the form $\langle \langle s, \delta t \rangle, c \rangle \in S \times \mathcal{T} \times \mathcal{T}$. A path’s timed configurations are commonly referred to as *positions* and denoted in the form $\langle s, \delta t \rangle @ c$, e.g. $\langle s_2, \delta t_2 \rangle @ c$ for the example above. Accordingly, a timed path is a path, where each position was extended to include a respective timestamp c .

Since all positions are of the form $\langle s, 0 \rangle @ c$, the abbreviated notation for path positions $s @ c$ is used. Further, sometimes the subscript π is added to a position $s @ c_\pi$ to clarify that the timed position belongs to a specific path π .

Definition 19 (Paths). The function $Paths_{\mathcal{T}\mathcal{K}} : S \times \mathcal{T} \rightarrow \mathcal{P}(\Pi)$ is used to return all timed paths $\pi \in \Pi$ of the $\mathcal{T}\mathcal{K}$ that start in a given configuration.

Zeno Paths Note that using the definitions above, it is possible that two configurations still appear at the same time stamp. For example, the $\mathcal{T}\mathcal{K}$ shown in Figure 6.5 below has a loop that only consists of “instant” transitions, i.e. transitions of length zero. This means that for the following path is possible: $s_0 @ 0 \xrightarrow{2} s_1 @ 2 \xrightarrow{0} s_2 @ 2 \xrightarrow{0} s_1 @ 2 \xrightarrow{0} s_2 @ 2 \xrightarrow{0} s_3 @ 4 \xrightarrow{2} \dots$ Evidently, this causes problems for the requirement of identifying all path positions by their distance from the path’s beginning. Such behaviour is usually referred to as *Zeno* and is problematic for simulation and verification of any kind of model. Zeno behaviour is not only present when there are cycles of instant transitions. Generally, any path is called Zeno (or “time-convergent”), if it contains an infinite number of ticks whose sum of durations is finite, i.e. if there are infinitely many nodes that can be visited in finite time.

This thesis assumes that all timed Kripke structures are free from Zeno behaviour. Note that if the timed Kripke structure is created according to CREST’s *next_transition_time* function, the $\mathcal{T}\mathcal{K}$ will automatically not contain Zeno loops, since any valid CREST model already forbids the existence of Zeno paths. The de-

tails for this approach are described in the next section.

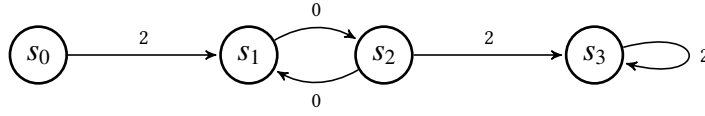


Figure 6.5 – A timed Kripke structure with Zeno behaviour.

Full TCTL Satisfaction relation Using the notions of timed paths and positions, it is possible to formally define the complete satisfaction relation $\mathcal{TK}, \langle s, 0 \rangle \models \phi$ for the timed interpretation of all TCTL operators.

This full satisfaction relation treats six different semantic cases. They are inductively defined and evaluated depending on the specific TCTL formula. These six cases can be split into three groups: two *state* formulas, two *logic* formulas, and the remaining two *path* formulas. Table 6.2 lists the individual cases.

State formulas operate on individual configurations. They include the case where $\phi = \text{True}$, which holds in any configuration, and the previously described case where the formula $\phi = p$, $p \in AP$ is an atomic proposition, which holds iff the configuration's state is labelled with the respective AP. Logic formulas include the negation (e.g. $\phi = \neg\psi$), which is satisfied if the subformula ψ does not hold, and the conjunction $\phi \wedge \psi$, which holds if the respective subformulas hold on the configuration.

Table 6.2 – Formal TCTL satisfaction relation. (Adapted version from [LÄÖ15].)

$\mathcal{TK}, s \models \text{True}$	always holds
$\mathcal{TK}, s \models p$	iff $p \in L(s)$
$\mathcal{TK}, s \models \neg\phi$	iff $\mathcal{TK}, s \not\models \phi$
$\mathcal{TK}, s \models \phi \wedge \psi$	iff $\mathcal{TK}, s \models \phi$ and $\mathcal{TK}, s \models \psi$
$\mathcal{TK}, s \models \phi EU_I \psi$	iff $\exists \pi \in \text{Paths}_{\mathcal{TK}}(s)$ there exists a $s'' @ c''$ s.t. $\mathcal{TK}, s'' \models \psi$, $c'' \in I$ and $\mathcal{TK}, s' \models \phi$, $\forall s' @ c' \in \text{pre}_\pi(s'' @ c'')$
$\mathcal{TK}, s \models \phi AU_I \psi$	iff $\forall \pi \in \text{Paths}_{\mathcal{TK}}(s)$ there exists a $s'' @ c''$ s.t. $\mathcal{TK}, s'' \models \psi$, $c'' \in I$ and $\mathcal{TK}, s' \models \phi$, $\forall s' @ c' \in \text{pre}_\pi(s'' @ c'')$

where ϕ and ψ are TCTL formulas and $p \in AP$ is an atomic proposition (AP).

Finally, path formulas are slightly more complex to verify, as they require the analysis of timed paths. For example, the satisfaction of a formula $\phi EU_I \psi$ (resp. $\phi AU_I \psi$) requires that one path (resp. all paths) from the initial configuration has (have) a position $s'' @ c''$ where ψ holds – i.e. $\mathcal{TK}, s'' \models \psi \wedge c'' \in I$. Further, ϕ has to hold in all configurations $s' @ c'$ that are attainable along the path before $s'' @ c''$. To get this set of timed configurations that are traversed before reaching a given position, we define the function pre_π .

Definition 20 (pre_π). The definition of the satisfaction relation \models uses a function pre_π that returns all positions of π that appear “before” a given position. For example, for a timed path $\pi = \langle s_0, \delta t_0 \rangle @ c_0 \xrightarrow{\tau_0} \langle s_1, \delta t_1 \rangle @ c_1 \xrightarrow{\tau_1} \langle s_2, \delta t_2 \rangle @ c_2 \xrightarrow{\tau_2} \dots$, the set returned by the function $pre_\pi(\langle s_2, \delta t_2 \rangle @ c_2)$ contains the positions $\langle s_0, \delta t_0 \rangle @ c_0$ and $\langle s_1, \delta t_1 \rangle @ c_1$. More formally, given a position, the function $pre_\pi : \langle S, \mathcal{T} \rangle @ \mathcal{T} \rightarrow \mathcal{P}(\langle S, \mathcal{T} \rangle @ \mathcal{T})$ is defined as follows:

$$pre_\pi(\langle s_i, \delta t_i \rangle @ c_i) = \{ \langle s_j, \delta t_j \rangle @ c_j \mid 0 \leq j < i \}$$

where $\pi \in Paths_{\mathcal{T}\mathcal{K}}$ is a $\mathcal{T}\mathcal{K}$ -path and $i, j \in \mathbb{N}$ are indices of the path positions of π .

6.1.2 Applied Model Checking

The actual model checking is based on the evaluation of TCTL formulas on timed Kripke structures using a graph algorithm-based approach. To avoid specifying dedicated procedures for all TCTL operators, Lepri et al. [LÄÖ15] make use of *normal form* for TCTL formulas, where each formula only contains the following operators:

$$\phi ::= \text{True} \mid p \mid \neg\phi \mid \phi \wedge \phi \mid EG\phi \mid \phi EU_I \phi \mid \phi AU_{I>0} \phi$$

where $p \in AP$, I and $I_{>0}$ are intervals and $\inf(I_{>0}) = 0 \notin I_{>0}$.

All other TCTL operators and formulas can be translated into normal form using equivalence rules. Subsequently, the authors provide algorithms and procedures for the evaluation of these operators on timed Kripke structures. The interested reader is referred to the original publication for details about the algorithm’s implementation.

Despite the work of Lepri et al., their approach to model checking cannot be directly applied to CREST systems, since their algorithms do not consider some particularities of CREST, such as transitions of length ε , for instance.

6.2 CREST Model Checking

For the verification of CREST systems, it is necessary to create a timed Kripke structure that faithfully represents the propositions of CREST system states and transitions between these states. Hence, the timed Kripke structure for CREST systems is based on the time advances of a CREST model. Its states S , thus, correspond to CREST system states (which are referred to as W in CREST’s semantics in Chapter 4). Kripke transitions represent CREST’s time advances, such that each transition $(s, t, s') \in \rightarrow$ relates to CREST’s $\langle s, t \rangle \xrightarrow{\text{advance}} s'$. Finally, the labelling function L maps states to a set of *state* and *port value* check propositions.

Definition 21 (CREST Kripke Structure). Through consideration of these requirements, a CREST Kripke structure \mathcal{CK} for the verification of CREST systems is defined as a quadruple $\mathcal{CK} = (S_{\mathcal{CK}}, \mathbb{T}, \rightarrow_{\mathcal{CK}}, L_{\mathcal{CK}})$, such that $S_{\mathcal{CK}} \subseteq W$ is a set of CREST system states, \mathbb{T} is the system’s time base, $\rightarrow_{\mathcal{CK}} \subseteq S_{\mathcal{CK}} \times \mathbb{T} \times S_{\mathcal{CK}}$ is a timed transition relation between these system states such that $\forall (s, t, s') \in \rightarrow_{\mathcal{CK}}$:

$\langle s, t \rangle \xrightarrow{\text{advance}} s'$ (according to CREST's semantics), and $L_{CK} : S \rightarrow \mathcal{P}(AP_{CK})$ is a function that assigns atomic propositions AP_{CK} to all states. Each $p \in AP_{CK}$ is related to either a state check or a port check, so that $p \in L_{CK}(s)$ expresses that the corresponding check evaluates to True in state $s \in S_{CK}$.

Notation. To increase legibility, the subscript $_{CK}$ is omitted in the rest of this chapter, unless required to avoid ambiguity. Thus, for example, S , \rightarrow and L refer to S_{CK} , \rightarrow_{CK} and L_{CK} , respectively.

The choice of timed Kripke structures for the representation of the evolution of CREST system states is very intuitive. Systems can advance according to the transition relation, which maps naturally to CREST's time advances. If a system spends δt time in a state s , it can trigger a transition $(s, t, s') \in \rightarrow$, if the transition time matches the amount of time spent in a state (i.e. $\delta t = t$).

Obviously, CREST's continuous semantics allow the system to advance in arbitrarily small timesteps. This means that for any transition $(s, t, s') \in \rightarrow$, $t > 0$, there are infinitely many system states $w \in W \setminus S$, that are reachable by advancing $\delta t < t$ time units⁶. Figure 6.6 shows a schematic representation of a timed transition $(s, t, s') \in \rightarrow$ with three example system states w_1, w_2 and w_3 that are "between" s and s' , i.e. reachable by advancing time in state s . Note the similarity to the concept of continuous configurations, that we introduced in the previous section (see Figure 6.3). According to CREST's semantics, w_1, w_2 and w_3 are reachable by spending time $\delta t_1, \delta t_2$ and δt_3 time units, respectively, in state s .

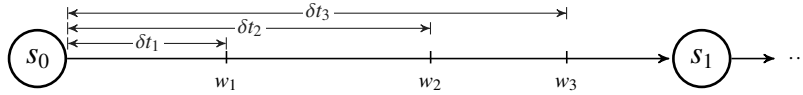


Figure 6.6 – Schematic representation of “intermediate” system states in CREST Kripke structures. The example shows three example system states w_1, w_2 and w_3 that are reachable from s by advancing $\delta t_1, \delta t_2$ and δt_3 time units, respectively.

Evidently, the purpose of a CK is to represent a CREST system's state space (i.e. all reachable system states). Therefore, each CREST system state $w \in W$ should be either directly represented as a CK -state $s \in S$ or indirectly through the advance of time from some s . The problematic lies in choosing a (finite) subset of system states $S \subseteq W$ so that the entire (infinite) state space of a CREST system can be represented and also enable efficient satisfiability testing of TCTL formulas. All CREST system states $w \notin S$ have to be reachable by advancing some time $\delta t \in \mathbb{T}$ from some state $s \in S$, so that δt is smaller than the duration time ($\delta t < t$) for some CK -transition $(s, t, s') \in \rightarrow$. Formally, this is expressed as

$$\forall w \in W \setminus S, \exists s \in S, \exists (s, t, s') \in \rightarrow, \exists \delta t \in \mathbb{T} \text{ such that } \langle s, \delta t \rangle \xrightarrow{\text{advance}} w \wedge \delta t < t$$

⁶Assuming that there is at least one variable whose value is modified by a CREST update, changes over time.

6.2.1 CREST Kripke Construction

CREST's semantics provide by default all mechanisms that are required for the creation of a CREST Kripke structure that represents a system's state space. The procedure is based on the usual (classic) state space exploration algorithm. Starting from an initial state, the creation of a state space proceeds iteratively. Assuming an unstable system state s^7 , the routine triggers the stabilisation routine to reach a stabilised successor state s' , and adds the state to S and transition $s, 0, s'$ to \rightarrow . Subsequently, the system calculates the next transition time ntt for s' , advances ntt and adds the transition (s', ntt, s'') , where s'' is the state reached by $\langle s', ntt \rangle \xrightarrow{\text{advance}} s''$.

One issue is that CREST systems expose non-determinism. This means that the stabilisation and advance procedures can have several successor states, that all have to be added. Procedure 1 describes the semi-algorithm for the CREST system's state space creation. Note, that it does not terminate on infinite state spaces.

Procedure 1: CREST state space exploration

Input : $CK = (S, \mathbb{T}, \rightarrow, L)$
 with $S = \{s_0\}$ and s_0 is a state of a CREST system
Output: CK , the explored CREST Kripke structure

```

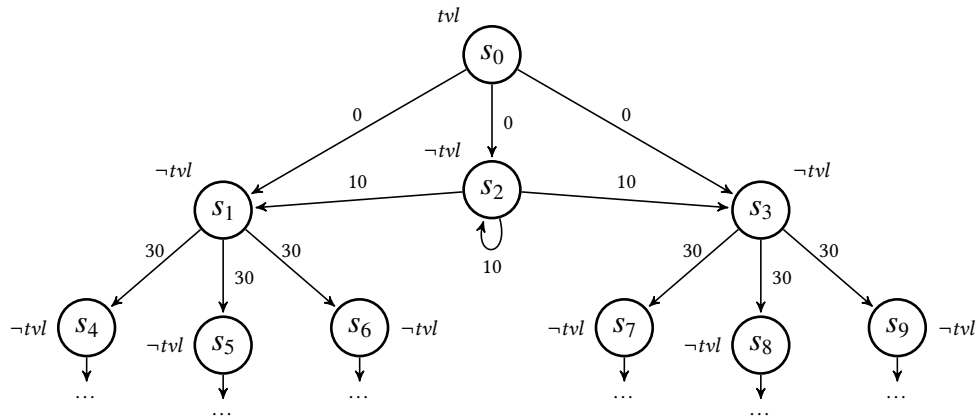
 $Q_u := S$  // The unexplored nodes
while  $Q_u \neq \emptyset$  do
  let  $s \in Q_u$  // Select and remove one node
   $Q_u := Q_u \setminus s$ 
   $ntt := \text{next\_transition\_time}(s)$  // Calculate next transition time
   $Q_s := \text{successors}(s, ntt)$  // Calculate stable successors
   $S := S \cup Q_s$  // Add the successors to the  $CK$ 
   $\rightarrow := \rightarrow \cup \{(s, ntt, s') \mid s' \in Q_s\}$  // Add transitions to successor states
   $Q_{new} := Q_s \setminus S$  // Select the unseen states
   $Q_u := Q_u \cup Q_{new}$  // Add them to the unexplored nodes
end
return  $(S, \mathbb{T}, \rightarrow, L)$  // Return the filled Kripke structure

```

AP Labelling Note that the algorithm does not modify the labelling L of the CK . It is therefore necessary to iterate over all nodes $s \in S$ and modify L , such that $L(s)$ returns the set of APs that hold in s . Practically, since the APs are state or port checks, this means evaluating whether the system is in a given state, or whether the specified port's value matches the check's constraint.

For example, Figure 6.7 shows the initial part of a CREST Kripke structure that has been created for the Germination box example. Additionally, the states are labelled with the AP ttl (for "timer very low"), which identifies system configurations where the countdown timer's value is equal to or below three – formally, $ttl \implies \text{countdown} \leq 3$. The table in Figure 6.7 shows that ttl holds in all states $s_i, i \in \mathbb{N}$ except s_0 . Formally: $ttl \in L(s_0) \wedge ttl \notin L(s_i), i \neq 0$.

⁷Remember that a state is unstable if it has enabled transitions or port values that have not been propagated yet. Such a state can be made stable through *stabilisation*. See Section 4.2 for details.



	GerminationSystem		germination_one		germination_two	
	curr. state	countdown	switch	temp.	switch	temp.
s_0	Off	0	off	22	off	22
s_1	BoxOne	30	on	22	off	22
s_2	Pause	10	off	22	off	22
s_3	BoxTwo	30	off	22	on	22
s_4	BoxOne	30	on	37	off	22
s_5	Pause	10	off	37	off	22
s_6	BoxTwo	30	off	37	on	22
s_7	BoxOne	30	on	22	off	37
s_8	Pause	10	off	22	off	37
s_9	BoxTwo	30	off	22	on	37

Figure 6.7 – A part of the CREST Kripke structure for the system in Figure 6.1. The table below provides details of the current state and several ports' values of the CK 's configurations.

CREST's semantics, however, allow the continuous, time-dependent evolution of variables – such as the `countdown` – using updates. Thus, due to the evolution a CREST system and its variables, it is possible that some of a state's configurations do not match the state's labels.

In the germination system for example, the `countdown` is continuously decreased as time passes. Therefore, for each state – except the initial state s_0 – there exists a point after which the `countdown`'s value drops below three and `tvI` becomes valid. In the CK shown in Figure 6.7, `countdown` ≤ 3 holds 27 time units after entering states s_1 and s_3 and 7 time units after s_2 . Therefore, these parts of the graph should be annotated with `tvI` and it is necessary to adapt the CREST Kripke structures.

In general, it is beneficial that any AP that is valid in a state is also valid in all its configurations. We refer to a state s of a CREST Kripke structure as *representative* for a set of APs if it fulfils that requirement. Formally, s is representative, iff every AP that holds in s (i.e. if $p \in L(s)$) holds in every configuration $\langle s, \delta t \rangle$, and all APs that don't hold in s don't become valid in any configuration of s . The concept of representative states is required to assert the validity of the model checking approach, since implementation of the satisfaction relation \models does not consider individual configurations, but only states. Thus, without representative states, a model checking algorithm could assume wrong labels and produce false results.

Representative State Formalisation Formally, representative states can be expressed using the Kripke configurations in a continuous interpretation:

$$\forall s, s' \in S, \forall (s, t, s') \in \rightarrow: p \in L(s) \implies \nexists \delta t, \delta t < t \text{ s.t. } \langle s, \delta t \rangle \not\models_c p$$

$$\text{and } \forall s, s' \in S, \forall (s, t, s') \in \rightarrow: p \notin L(s) \implies \exists \delta t, \delta t < t \text{ s.t. } \langle s, \delta t \rangle \models_c p.$$

Note that these definitions make use of the satisfaction relation in the continuous semantics \models_c . The specification of this relation exceeds the scope of this chapter. The reader is referred to Lepri et al. [LÄÖ15] for details of this formalisation.

The transformation from a CK with non-representative states to a CK with representative states relies on the repeated splitting of transitions and introduction of new states, until each state is representative. For example, for the transition $(s_1, 30, s_4) \in \rightarrow$ where `tvI` becomes valid after 27 time units, a new state s^{tvI} is created such that s^{tvI} is the “earliest” state where `tvI` is satisfied. The transition is replaced with two new transitions of duration 27 and 3 respectively: $(s_1, 27, s^{tvI})$ and $(s^{tvI}, 3, s_4)$. Hence, the total duration remains at 30. Obviously, L has to be adapted such that $L(s^{tvI}) = \{\text{tvI}\}$.

If necessary, i.e. if there were a configuration alongside the transition $(s^{tvI}, 3, s_4) \in \rightarrow$ where `tvI` is not satisfied anymore, the process is recursively repeated on the newly created transitions. This splitting process is visually depicted in Figure 6.8.

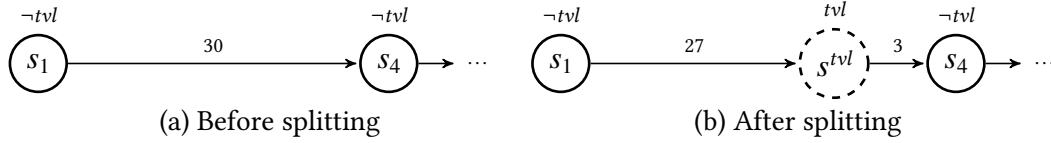


Figure 6.8 – Splitting of a transition to create representative states

The discovery of the points in time at which an AP stops (resp. starts) to hold is very similar to the discovery of the next transition time. In fact, for state checks (AP that verify automata states), the transitions are already split accordingly, since the *next_transition_time* method is used for the CREST Kripke structure’s creation. For port checks, a slightly adapted approach is used. In fact, first the check’s value is evaluated on a state s and creates two scenarios:

1. If the check does not hold, a new transition is temporarily added to the CREST system. This transition’s guard returns True if the check holds. Executing the *next_transition_time* routine on this modified CREST system returns the point in time when the guard (and thus the check) holds. If the value is ∞ or larger than the transition’s length we can disregard the result. If the returned value is before a transition to a successor state, this value is used to split the transition.
2. The check holds in s . In this situation the check is inverted (negated), so that the first scenario is executed.

The attentive reader will notice the similarities to *crestdsl*’s handling of discrete behaviour changes defined in *if-then-else* conditional statements.

ε -Transitions Certain APs express that a continuously decreasing (respectively growing) port’s value is strictly smaller (resp. bigger) than some constant. For example, the AP tl is used to express that the *countdown*’s value is *low*, i.e. below five in a certain state ($tl \implies \text{countdown} < 5$).

The naive approach would be to perform the splitting of the transitions as defined above and create a s , with $tl \in L(s)$. This results in a wrong state space though, as the check $\text{countdown} < 5$ does not hold for this state, since the timer value after 25 time units is 5, which is not considered low (i.e. $L(s) \neq tl$). However, all configurations after this point are considered low. Hence, all configurations $\langle s, \delta t \rangle$, $\delta t > 0$ should be annotated with tl , except $\langle s, 0 \rangle$. The solution is to split the second transition ($s, 5, s_4$) again and thereby create another state s^{tl} . The transitions from and to that state are (s, ε, s^{tl}) and $(s^{tl}, 5 - \varepsilon, s_4)$ ⁸, as shown in Figure 6.9. The thereby created state s^{tl} with $tl \in L(s^{tl})$ is representative of all configurations of that state.

Note that instead of introducing two new states s and s^{tl} connected by edges with weights $25, \varepsilon$ and $5 - \varepsilon$, a practical implementation would rather choose to only introduce one new node and split the transition into $25 + \varepsilon$ and $5 - \varepsilon$, to avoid “unnecessary” nodes and to speed up the analysis algorithms.

⁸ $\varepsilon \in \mathbb{T}$ is an infinitely small value such that there does not exist any value that is smaller (except zero). Formally, this constraint is expressed as $\varepsilon > 0 \wedge \nexists \tau \in \mathbb{T}, \tau < \varepsilon$.

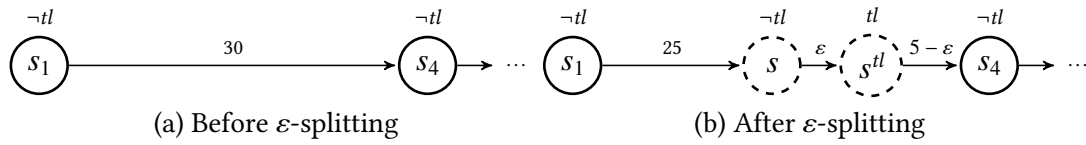


Figure 6.9 – Splitting a transition to create representative states using ϵ -transitions

6.2.2 Ensuring Left-Total Transitions

The CREST Kripke creation procedure iteratively visits unexplored states and adds transitions to successors to the structure. However, some CREST Kripke states do not have any successor states. Remember the `HeatModule` example described in Chapter 5. If the lamp becomes too hot, the system transitions to an error state that it cannot leave. Procedure 1 would therefore create a new node for this system configuration, but no outgoing transitions, since the state cannot be left anymore.

This creates a problematic situation, as both the formal definitions of timed Kripke structures in the previous section, and the algorithms described by Lepri et al. for the model checking rely on the fact that the transition relation is left-total, i.e. that there is an outgoing situation for every $s \in S$.

Therefore, to guarantee the soundness of the model checking approach, it is necessary to add new transitions (s, t, s) to \rightarrow . The length of this transition should be a non-infinitesimal value greater than zero, to avoid the introduction of Zeno behaviour.

From a practical point of view t should be chosen so that it facilitates the model checking process. Since the model checking of CREST Kripke structures relies on the reduction of continuous model checking to pointwise model checking by splitting transitions using the value gcd , t should be chosen as gcd or a small multiple of it, so that the size of S does not grow more than necessary during the transformation.

6.2.3 Replacing ϵ -values

One caveat of the approach of Lepri et al. is that it does not support the use of ϵ values in transition lengths. Indeed, they choose to abstract over ϵ -transitions in their time base. In the following, we review the application of this continuous model checking approach on CREST Kripke structures. To satisfy the requirements of their method, it is necessary to abstract over ϵ -transitions and replace ϵ with the value zero. Thereby, all ϵ -values are entirely removed.

This abstraction is a choice that is common in the domain of real-time systems, where state changes are assumed to be instantaneous. While it is a minor detail in most situations, in some corner-cases this approach may lead to false results. For example, when considering an excerpt of a CREST Kripke structure as shown in Figure 6.10 and a TCTL formula $\phi = \neg p AU_{>3} p$, we see that in the original CREST Kripke, p becomes valid after ϵ , which satisfies the interval. When setting the value of ϵ to zero, as shown in the adapted CREST Kripke (Figure 6.10, lower part), ϕ is not satisfiable, as $\neg p$ stops being valid after three time units already.

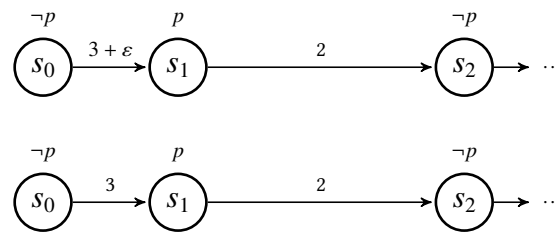


Figure 6.10 – Problematic removal of ε in CREST Kripke structures: The formula $\phi = \neg p AU_{>3} p$ holds for the original CREST Kripke structure (above), but not its seemingly equivalent, adapted version where the value of ε is set to 0.

It is possible to discover the error in this altered evaluation by searching for a path between s_0 and s_2 and calculating its total length based on the transition lengths in the original CREST Kripke structure. This possibility, however is not always provided. The situation becomes significantly more difficult to verify if there are transition loops along the path or complex subformulas. An automation of this approach would be theoretically feasible, but require the testing of all possible paths that satisfy the formulas. Depending on the underlying system – and thus CREST Kripke structure – the number of such paths might be too large to be computationally evaluated.

6.3 *crestdsl* Verification

The assertions that unwanted behaviour cannot occur or that a system trace to a beneficial system configuration exists are essential for effective CPS design. The *crestdsl* implementation provides the **verification** package, which exposes several APIs that provide these capabilities to the user. In fact, *crestdsl* implements two different interfaces: First, various convenience methods can be used to trigger simple reachability analyses and verifications. Second, for more elaborate queries, the library features a TCTL implementation that can be used for the analysis of CREST system state spaces. The user has the possibility to manually instruct the **ModelChecker** which implements the theory described in the previous sections.

6.3.1 Checks

Both of these approaches require the configuration of **checks**. Checks are tests for system properties that either assert or disprove a system’s configuration, similar to the atomic propositions introduced above. The two types of basic checks are state checks, which test whether an entity’s automaton is currently in a certain state, and port checks, which compare the values of entity ports. Checks are initialised using the **check** function, which takes either an **Entity** or a **Port** object as argument and returns a **StateCheck** or **PortCheck** object, respectively. Note that **check** automatically provides the correct object, based on the specified parameter.

To complete the check configuration, an operator and comparator value have to be set. State checks use equality operators ($=$, $!=$) and entity states as comparator,

Listing 6.1 – Examples of `crestdsl` checks.

```

1  from crestdsl.verification import check
2  sys = GerminationSystem() # initialise system
3
4  statecheck = check(sys) == sys.off # create a state check
5  print(statecheck.check()) # trigger the check (returns True)
6
7  portcheck = check(sys.countdown) >= 0 # two port checks
8  another = check(sys.germinationbox_one.temperature) == 33
9
10 # use of operators to alter / combine checks
11 negate = - another # negation using minus
12 orcheck = another | negate # disjunction using |
13 andcheck = another & negate # conjunction using &
14
15 # compare two ports' values
16 port_compare = check(sys.germinationbox_one.temperature) > sys.
    germinationbox_two.temperature)

```

port checks require the definition of operators that are suitable for the port's resource (e.g. comparison operators for numeric values, equality for enumeration types). Ports can be compared to either constant values or other ports.

The specification of checks is conveniently handled by an operator-based API, as shown in Listing 6.1. Operators can also be used for the definition of conjunctive, disjunctive and negated checks as shown in Lines 11 – 13. Every check provides a `check()` method that evaluates if it holds on the system in its current state. The method returns a Boolean value, indicating whether the check holds (see Line 5).

6.3.2 Simple API

Usually, the modeller is interested in whether the check is satisfiable at some point in the future, rather than the mere analysis of a current system configuration. `crestdsl`'s `verification` package also provides functionality to quickly evaluate such scenarios. For example, `is_possible(my_check)` will create an instance of the `Verifier` class, which can be used to test whether there is a possibility to reach a system state in which `my_check` is satisfied. Alternatively, the functions `always`, `always_reachable`, `never` and `forever` provide convenient implementations for other common verification scenarios. In the background, these functions create TCTL formulas that express the intended properties. Table 6.3 shows a mapping of the functions to the TCTL formulas they implement.

The verification functions serve as the façade of a more complex `Verifier` class, which implements the actual model checking interface. Each of the above functions returns a `Verifier` object, that can be further customised by “chaining” additional specifications. For example, the methods `after` (resp. `before`) can be used to adjust the interval of the timed TCTL operators, as shown in Listing 6.2. Finally, the verifier's `check()` method triggers the model checking routine.

Table 6.3 – Verification API and the corresponding TCTL formulas

crestdsl Function	TCTL Formula
<code>is_possible(chk)</code>	$EF\ chk$
<code>always(chk)</code>	$AG\ chk$
<code>always_possible(chk)</code>	$AG\ AF\ chk$
<code>always_possible(chk, within=time)</code>	$AG\ AF_{[0,time]}\ chk$
<code>never(chk)</code>	$AG\ \neg chk$
<code>forever(chk)</code>	$EG\ chk$

where *chk* is any basic or composed check, as introduced above.

This convenient interface, can be easily extended to include further functions, to adjust the verification API to the target domain and expertise of the users.

Behind the scene, the Verifier is responsible for creating a state space for the model checking algorithms. By default, the state space is automatically managed by the tool and grown large enough so that the formulas can be verified. However, in case of formulas that are not time constrained, i.e. whose interval is of the form $[a, \infty)$, *crestdsl* has to create the complete state space of the provided system. Note that this will produce a warning which informs the user that the procedure can easily lead to problems in systems with extremely large or unbounded state spaces, as the underlying loop structure only stops when no new nodes are encountered. This means, that the exploration will not terminate for infinite state spaces.

Listing 6.2 – Reachability evaluation of a *crestdsl* check.

```

1 from crestdsl.verification import check, is_possible, before
2
3 system = GerminationSystem()
4 chk = check(system.countdown) == 0
5
6 # chaining formula specification, after and before
7 is_possible(chk).after(20).check()
8 before(15).always(chk).check()

```

The model checking itself is executed in the background using *crestdsl*'s `ModelChecker` class, which implements the algorithms provided by Lepri et al. in [LÄÖ15] and the CREST adaptations described in the previous section. The algorithms and procedures were slightly adapted to make use of Python's convenient graph exploration libraries. Further, the modified algorithms automatically remove ε -transitions and also provide correct results in the presence of incomplete state spaces, i.e. when the state space exploration has been manually bounded.

6.3.3 TCTL Model Checking

Evidently, the formulas and checks that are automatically verified by the introduced functions provide essential analysis capabilities, but do not cover the wide range of functionality analyses required by experienced users. To support advanced model checking concepts, the `verification` package provides classes that allow the manual creation and exploration of state spaces, precise definition of formulas using TCTL operators and choice of model checker implementations.

To this extent, `crestdsl` implements a range of TCTL operators, including the standard logic operators (e.g. `Not`, `And`, `Or`, `Implies`) and timed operators (e.g. `EU/AU`, `EF/AF`, `EG/AG`). Intervals can be defined using the `Interval` class, which exposes a similar operator interface to checks. Users who need additional operators can obviously use standard Python mechanisms to implement them based on the provided templates. Listing 6.3 shows an example for the definition of complex TCTL formulas.

Listing 6.3 – Examples of `tctl` formula definitions.

```

1  from crestdsl.verification import tctl, check
2
3  # define a system and checks
4  sys = GerminationSystem()
5  tl = check(sys.countdown) < 5
6  tvl = check(sys.countdown) <= 3
7  is_pause = check(sys) == sys.pause
8
9  tl_until_tvl = tctl.AU(tl, tvl) # default interval = [0, inf)
10
11 # reachability formula
12 tvl_reachable_before_30 = tctl.EF(tvl, tctl.Interval(end=30))
13
14 # use of operators for interval specifications
15 after_10 = tctl.Interval() >= 10
16 within_30 = tctl.Interval() <= 30
17 always_pause_within_30 = tctl.AG(tctl.EF(is_pause, within_30),
    after_10)

```

The evaluation of these formulas has to be performed on a state space. In general, a state space is a graph structure whose nodes capture different states of a system and whose transitions are annotated with the duration. `crestdsl`'s `StateSpace` reuses the popular `networkx`⁹ Python library, which provides excellent graph and network analysis algorithms. The use of this library facilitates the implementation of model checking, as many graph algorithms (e.g. search for successor/descendent, shortest paths, and strongly connected components) can be used right away, without the need to reimplement this functionality. `StateSpace` objects are initialised with the system whose states should be explored, its global state being used as the initial state from which exploration starts. The class provides two methods of expansion: `explore` and `explore_until_time`. The former searches for unexplored nodes within the system and tries to find successor nodes, which modify the system's behaviour¹⁰

⁹<https://networkx.github.io/>

¹⁰This functionality relies on `next_behaviour_change_time`, as introduced in Chapter 5.

and adds them to the state space. This way, the state space iteratively grows every time `explore` is called. For convenience, a parameter can be passed that states how many iterations should be performed, or `None` to iterate until no new unexplored nodes can be added to the graph.

Alternatively, `explore_until_time(time)` can be used to grow the state space until all unexplored nodes have a shortest path distance that is greater than `time`. This exploration can be useful when only a subset of the state space is of interest and avoids the unnecessary creation of not needed nodes.

Note that despite the benefits of this approach in terms of performance, there is the possibility that the model checking algorithms will report wrong results for “edge cases”. An example is the evaluation of the formula $\phi = AG_{\leq 20} EF p$ on the (partially explored) state space shown in Figure 6.11. The formula states that for each node in the state space, there exists a path to a state where p holds. The state space has been grown to include all states that are reachable within 20 time units from s_0 . When considering the evaluation ϕ on state s_0 the result will be wrongly identified as unsatisfiable. It is easy to see, that there exists a path from state s_1 to s_3 , where p would hold. However, as the path length from s_0 to s_2 amounts to 22 time units, the `explore_until_time` procedure will cease exploration at s_2 , i.e. before s_3 is explored.

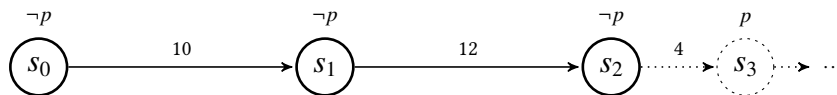


Figure 6.11 – Problematic TCTL evaluation on partial state spaces. Assuming s_3 has not been explored and is thus not part of the state space, the evaluation of $AG_{\leq 20} EF p$ on state s_0 will report a wrong result.

The actual model checking logic is implemented in the `ModelChecker` class. After initialisation of the model checker using the state space as parameter, the user merely has to call the `check` method, providing a TCTL formula as parameter. The unwrapping of the formula and the transformation of the state space to a CREST Kripke structure are performed automatically. Listing 6.4 shows an example that evaluates the formula that was defined in Listing 6.3.

Listing 6.4 – Running the *crestdsl* model checker.

```

1 from crestdsl.verification import StateSpace, ModelChecker
2 sys = GerminationSystem()
3 formula = tctl.AG(tctl.EF(is_pause, tctl.Interval() > 10),
4                 tctl.Interval() <= 30)
5 statespace = StateSpace(sys)
6 statespace.explore(None) # explore entire state space
7 mc = ModelChecker(statespace)
8 mc.check(always_pause_within_30) # trigger model checking

```

6.3.4 Limitations

`crestdsl`'s model checking APIs provide a convenient means to verify certain properties, but also an extended TCTL interface for the definition of complex scenarios.

Python's Performance Despite these user-friendly interaction capabilities, the model checker is not intended for the verification of highly complex systems. Its implementation is based on interpreted Python code and thus has a lower performance than other model checkers which operate closer to the hardware. Some issues might also arise in terms of memory, since a publicly available, non-optimised graphing library is used for the representation of the CREST Kripke structure. The system also does not use any other optimisations that are known from existing model checkers to avoid state space explosion (e.g. partial order reduction, symbolic states, etc.)

As a result, the model checker performs well on models with moderately sized state spaces. When it comes to large and complex state spaces, however, `crestdsl`'s model checker reaches CPU and memory limitations. In such situations dedicated high-performance model checkers (such as e.g. [Bér+01]) should be used, which implement optimization strategies and avoid these limitations.

Global Model Checking Another performance loss can arise from the implemented algorithm itself. This algorithm performs *global* model checking, which means that it always traverses the entire state space for evaluation. This can lead to poor efficiency when the formula's answering only requires testing parts of the state space.

Dedicated local algorithms may be better in such situations, although their performance heavily depends on the particular state space and the precise formula that should be evaluated. `crestdsl` provides a solution to this problem by allowing the parametrised exploration of the state space, that guarantees a minimum path length for all leaf nodes but does not explore further than that. Thus, by limiting the state space size, it is also possible to limit the number of "unnecessarily explored nodes".

6.4 Summary

This chapter describes the formal verification of CREST systems using a model checking approach. The concept is based on existing research results that use TCTL and timed Kripke structures for the exploration of state spaces. CREST proposes a global model checking method that reuses the algorithms defined by Lepri et al. [LÄÖ15] and adapts them to the CREST domain. The chapter highlights the theoretical aspect of verification and elaborates on the creation of timed Kripke structures that represent the state space of a CREST system. Furthermore, particularities such as the removal of infinitesimal values in transition durations are addressed.

Next to theoretical considerations, the `crestdsl` implementation of this approach is presented using code examples. `crestdsl` features two different APIs. Convenience methods offer a simple means to model checking to unfamiliar users. On the other hand, a powerful TCTL API can be used by experienced modellers. Both APIs evaluate formulas using an implementation of the global model checking algorithm.

Chapter 7

Conclusion

This final chapter reviews the work described in the thesis and summarises its results. Section 7.1 revisits the research questions that were introduced in Section 1.1 and places them alongside the developed solutions. Based on this work, Section 7.2 then outlines additional research areas of the Continuous REactive SysTems language (CREST) and introduces new challenges that build on it.

7.1 Summary

Based on the continued growth of modern cyber-physical systems (CPSs) in size, complexity and connectivity, system engineers created powerful modelling languages and intricate design software to support their development efforts. These modelling platforms allow the analysis, simulation and verification of systems already during their design phase, thereby avoiding the costly discovery of flaws at construction time. Nowadays, the use of these tools is indispensable in many domains, and often mandatory, e.g. for safety-critical systems, so that tool vendors strive to make their software universally applicable.

The problem of this versatility is, however, that the produced models are generic as well and do not match the underlying system or target domain closely. This semantic gap slows down model development time, causes misunderstanding and, if faults remain undiscovered, modelling errors that lead to false confidence in the system's design. Furthermore, the use of non-specialised modelling platforms requires users to spend time and effort on acquiring tool-specific knowledge and skills, which often involves steep learning curves and many months of training.

This research project explores the use of domain-specific languages (DSLs) to alleviate these issues. Specifically, this work investigates the creation of a language for the modelling of CPSs that are driven by the production, consumption and transfer of physical resources such as light, electricity and temperature. These *resource flow CPS* tend to be systems such as smart homes and office automation applications that consist of off-the-shelf components such as lamps, electrical heaters, and similar. The project aims to develop a flexible modelling language that is dedicated to these systems and offers a low entry barrier for novice users, but is powerful enough to be used efficiently by expert modellers. The research resulted in the development of

CREST, a DSL that reuses concepts from several well-established formalisms such as architecture description languages, hybrid automata and data flow languages, and merges them into a coherent language. CREST's formal syntax and semantics unify important modelling concerns such as a hierarchical system composition, discrete behaviour and continuous time into a formalism that can be used to model, simulate and verify the systems under study.

The methodology leading to these results was guided by five research questions that are progressively answered throughout this thesis. The rest of this section reviews these questions and summarises their respective findings.

Research Question 1. *What are the properties required from a language or tool to model the resource flows and behaviour of custom assembly CPSs?*

The first research question investigates which properties a modelling language or tool has to support, so it can be used for the representation of resource flows between the components of CPSs. To this extent, three case study systems were designed and analysed, as described in Chapter 3. The case studies include a smart home, an office automation system and an automatic gardening application. Despite the different domains, all three systems share in common that their functionality is based on the flow of resources within the system, and that they tend to be compositions of standard, off-the-shelf components. The examination resulted in the discovery of six key aspects which should be supported by a CPS language:

1. *Reactivity*; for components to observe and react to their environment.
2. *Synchronism*; to model instantaneous, discrete CPS behaviour changes.
3. *Parallelism*; to represent concurrent physical phenomena.
4. *Locality*; for coherent component separation and simple system composition.
5. *Continuous Time*; to express the continuity of resource flows.
6. *Non-determinism*; as it is inherent to many physical systems.

Research Question 2. *Are existing CPS languages suitable for the creation of useful models that remain close to their application? Do they constrain the expression of domain-specific features and thereby increase the entry barriers for novice users?*

The second research question examines whether existing modelling languages and tools support these requirements. Chapter 3 therefore evaluates twelve state-of-the-art languages that are used for CPS modelling and analyses their properties in respect to the aspects. Additionally, four other properties – *usability*, *expressivity*, *suitability* and the availability of a *formal* basis – are used to find good candidates for the modelling of our case study systems. The latter four aspects evaluate whether the

languages are good matches for CREST's specific target domain and user audience and can thus be seen as orthogonal to the former key aspects, which evaluate the languages on a general level.

The results of this evaluation show that only few languages meet all requirements. Furthermore, the candidates that support all of them are difficult to learn and do not provide the necessary domain-concepts. Thus, they induce a semantic gap between model and system or require significant customisation effort.

Research Question 3. *How can we create or adapt a language to fill the need of modelling domain-specific aspects such as resource flows in CPSs? How can existing language features and implementations be reused to lower development and maintenance efforts?*

The third research question examines the creation of a DSL for the modelling of the case study systems. Based on the evaluation results of the previous research question, the decision was made to create a dedicated language for the modelling of resource flows within CPSs.

The resulting CREST DSL meets all six language requirements, as described in Chapter 4. CREST aims for a low entry barrier to support modelling novices, and also to provide powerful modelling mechanisms that can be used by experts for intricate systems modelling. The DSL is inspired by existing languages and reuses well-established concepts. For instance, it borrows the hierarchical component structure view from architecture description languages, dataflow concepts from synchronous languages and the combination of continuous and discrete behaviour that is known from hybrid automata theory. CREST is implemented in the graphical CREST diagram language and follows the *Physics of Notation* methodology to increase usability.

CREST also serves as the basis for `crestdsl`, an internal DSL implementation in the Python programming language. `crestdsl` allows the modelling of hybrid systems using the syntax and execution environment of a widely popular programming language. It integrates seamlessly into the existing Python ecosystem and allows the reuse of integrated development environments, testing frameworks and continuous integration. Chapter 5 introduces to the modelling workflow with `crestdsl`.

Research Question 4. *How should the formal syntax and semantics of the DSL be characterised, so that the created CPS models are well-defined and can be simulated and analysed?*

The merging of aspects from several languages into a coherent DSL can easily lead to inconsistencies, resulting in unclear system designs and language semantics. The fourth research challenge therefore examines the possibility to formalise the language, so that CREST models are well-defined and an unambiguous simulation of its system models is possible. Chapter 4 presents the results of this formalisation: A mathematically defined system structure and formal semantics based on structured operational semantics rules. This formalisation allows the well-defined system

creation and continuous time simulation. It further provides a clear specification of CREST's expressiveness and makes it comparable to other formalisms.

Research Question 5. *How can we use formal verification approaches to verify system behaviour and which techniques can be used for verification of our DSL's CPS models?*

The last research question investigates how CREST can be formally verified. As the verification of CPS correctness is one of the main driving forces behind this research project, it is of high importance that the language offers this capability. This thesis provides the details of a formal verification approach for CREST systems and gives insight into some of its problematic areas. Based on these considerations, Chapter 6 describes the effort towards answering this final research question. The solution is based on the use of timed computation tree logic (TCTL) formulas for the description of system properties and their evaluation on CREST Kripke structures, an adaptation of timed Kripke structures that model the state space of a CREST system.

The creation of such CREST Kripke structures and the definition of TCTL formulas are described from a theoretical point of view, alongside the description of an evaluation algorithm. Finally, the chapter outlines use of `crestdsl`'s verification APIs, which implement the described approach and prove its viability.

7.2 Perspectives

As with most research projects, CREST too lacks significant application reports and requires a more extensive use in real-world scenarios. Even though this research was driven by actual case studies, it is essential that both CREST and `crestdsl` are applied to more and diverse modelling problems to prove their viability. These reports will also provide invaluable feedback of the usability of `crestdsl`, so that the language's application programming interfaces (APIs) can be adapted to the users' needs. Next to the improvement of the existing formalism and implementation, however, the work on CREST and `crestdsl` also opened the door for other intriguing research topics.

Adding New Modelling Concepts The CREST DSL was created to reduce the semantic gap between system and modelling language, and increase the application of formal methods for CPS modelling. The language's main goal is to provide a simple, clear means to expression of resource flows within CPSs and use these models for simulation and verification.

While CREST already reuses ideas from other modelling languages (e.g. entity composition, dataflow-based value propagation and hybrid system behaviour), it is of interest to analyse whether other language's features can be reused. For instance, as explained in Section 4.4, Petri nets (PNs) provide intuitive modelling of concurrency and resource movements within a system, but expose problems when creating hierarchical systems, and require language extensions for the modelling of complex data types. It would be of interest to evaluate the use of (Higher-order) PNs to model

the behaviour of CREST entities, such that the benefits of CREST (e.g. hierarchical composition, formally-defined resource types) can be combined with the strengths of PNs, while still providing a simple, dedicated formalism and language.

Verification Heuristics The applicability of modelling DSLs strongly relies on their verification capabilities. Chapter 6 of this thesis introduces a TCTL-based approach for the verification of CREST systems. Despite the use of a formally proven algorithm, the actual implementation of this approach can result in the need of significant amounts of computation power, due to CREST's continuous nature. Several heuristics have been developed to counteract this problematic in other formalisms. These techniques should be evaluated for their viability in CREST. Further, an analysis of new ones dedicated to the improvement of CREST's verification would certainly increase the applicability of the language.

Behaviour Approximation Oftentimes, CPSs builders not only need simple and affordable modelling tools, but also require help during the creation of the models themselves. The conceptualisation of physical influences between system components is non-trivial. In recent years, machine learning evolved into a well-established technique for approximation of such as functions (e.g. using regression and classification mechanisms). Initial results for CREST systems have been promising and were published at dedicated workshops [KCB18]. Further research is needed though to adapt the approach for general use.

Automated Controller Generation Another opportunity is the automated creation of system controllers. These components observe a system and modify its inputs to assert beneficial and avoid unfavourable behaviour. The creation of such controllers is a time-intensive process in general and highly complex in the domain of hybrid systems such as CREST. Recently, reinforcement learning (RL) techniques have shown promising results in diverse application areas, including systems controls. Usually, RL treats the system under study as black boxes and require large computational infrastructures to implicitly create a model of the system behaviour.

It is certainly of interest to combine the automated learning techniques with the advantages of formal modelling languages, such as the availability of well-defined system semantics and simulation. This connection can potentially harness the strengths of both domains by allowing automated planner and controller generation using model-based reinforcement learning.

Appendix A

GrowLamp Model – Function Implementations

This chapter presents the implementations for the transition guards, as well as update, action and influence functions in Figure 4.1. The functions are grouped by the entity in which they are defined.

GrowLamp The *on-guard* disables the transition to On as long as less than 100 watts electricity are available or when the switch is off. As soon as enough electricity is provided (≥ 100) and the switch is on, the transition is enabled.

$$on-guard(bind, pre) = \begin{cases} \text{False} & \text{if } bind(\text{electricity}) < 100\text{Watt} \vee bind(\text{switch}) = \text{off} \\ \text{True} & \text{if } bind(\text{electricity}) \geq 100\text{Watt} \wedge bind(\text{switch}) = \text{on} \end{cases}$$

The *off-guard* exposes the inverse behaviour.

$$off-guard(bind, pre) = \begin{cases} \text{False} & \text{if } bind(\text{electricity}) \geq 100\text{Watt} \wedge bind(\text{switch}) = \text{on} \\ \text{True} & \text{if } bind(\text{electricity}) < 100\text{Watt} \vee bind(\text{switch}) = \text{off} \end{cases}$$

When transitioning to On, the transition action `increment_count` is executed, which takes on-count's previous value and adds one.

$$increment_count(bind, pre, \delta t) = pre(\text{on-count}) + 1$$

There are several updates linked to the On state as well. `update_on_time` measures the total amount of time that the automaton spent in state On. Its value continuously increases as time passes.

$$update_on_time(bind, pre, \delta t) = pre(\text{on-time}) + \delta t$$

When the system is in state On, the light module receives 100 watts of electricity.

$$update_light_electricity(bind, pre, \delta t) = 100\text{Watt}$$

If there is more electricity available when the growing lamp is on, then the additional electricity is made available to the heating module:

$$\text{update_heat_electricity}(bind, pre, \delta t) = bind(\text{electricity}) - 100\text{Watt}$$

When the GrowLamp is Off, neither the lighting nor the heating module receive any electricity.

$$\text{light_electricity_zero}(bind, pre, \delta t) = 0\text{Watt}$$

$$\text{heat_electricity_zero}(bind, pre, \delta t) = 0\text{Watt}$$

The influence `fahrenheit_to_celsius` reads the room-temperature input, converts it to Celsius and writes it to the adder's temp-in input.

$$\text{fahrenheit_to_celsius}(bind, pre, \delta t) = (bind(\text{room-temperature}) - 32) \cdot 5/9$$

The following four influences simply read the source port's value and return it. Usually the definition of such influence functions are omitted entirely. They are provided for completeness.

$$\text{heatswitch_influence}(bind, pre, \delta t) = bind(\text{heatswitch})$$

$$\text{forw_heat}(bind, pre, \delta t) = bind(\text{heat}_H)$$

$$\text{forw_temp}(bind, pre, \delta t) = bind(\text{temperature}_A)$$

$$\text{forward_light}(bind, pre, \delta t) = bind(\text{light}_L)$$

LightElement The `on-guardL` is responsible for deciding when the automaton switches to state `OnL`. That is when there is more than 100 watts available at the `electricityL` input.

$$\text{on-guard}_L(bind, pre) = \begin{cases} \text{False} & \text{if } bind(\text{electricity}_L) < 100\text{Watt} \\ \text{True} & \text{if } bind(\text{electricity}_L) \geq 100\text{Watt} \end{cases}$$

When the LightElement is on, the `on_update` writes the output of 800Lumen to the output.

$$\text{on_update}(bind, pre, \delta t) = 800\text{Lumen}$$

The `off-guardL` enables the transition to `OffL` if less than 100 watts electricity are available.

$$\text{off-guard}_L(bind, pre) = \begin{cases} \text{False} & \text{if } bind(\text{electricity}_L) \geq 100\text{Watt} \\ \text{True} & \text{if } bind(\text{electricity}_L) < 100\text{Watt} \end{cases}$$

When the LightElement is off, no light is produced (`off_update` returns 0Lumen).

$$\text{off_update}(bind, pre, \delta t) = 0\text{Lumen}$$

HeatElement The heating module's behaviour is very simple. It converts all electricity to heat energy. This heat creates a temperature increase directly under the lamp which is measured in degrees Celsius. For example 100 watts electricity provide a temperature increase of 1 degree under the lamp. The HeatElement's behaviour is assumed to be linear.

$$\text{heat_output}(bind, pre, \delta t) = \begin{cases} bind(\text{electricity})/100 & \text{if } bind(\text{switch}) = \text{on} \\ 0\text{Celsius} & \text{if } bind(\text{switch}) = \text{off} \end{cases}$$

Adder The Adder entity calculates the sum of all input ports and writes the result to its output port. In this case, it sums the values of the heat-in and the temp-in ports.

$$\text{add}(bind, pre, \delta t) = bind(\text{temp-in}) + bind(\text{temp-in})$$

Appendix B

CREST Time Base

CREST's semantic requirements towards a time base for continuous time system definitions and verification are close to the properties of time domains that are described in other publications. In order to increase compatibility and coherence, CREST reuses the time domain defined by Lepri et al. in [LÁÖ15] and extends it. This section briefly recaptures those definitions which are used by CREST.

The time base \mathbb{T} of a CREST system is defined according to the theory $\text{TIME}_\varepsilon^\infty$. $\text{TIME}_\varepsilon^\infty$ is an extension of TIME and includes the infinitesimal and infinity elements ε and ∞ . TIME is defined as a linearly ordered commutative monoid $(\text{Time}, +, 0, <)$ with monus operator $(\dot{-} : \text{Time} \times \text{Time} \rightarrow \text{Time})$ and \min and \max operators:

$$\min : \text{Time} \times \text{Time} \rightarrow \text{Time} \quad \max : \text{Time} \times \text{Time} \rightarrow \text{Time}$$

These operators adhere to the axioms in Table B.1.

Table B.1 – Axioms in the theory TIME , for $t_1, t_2, t_3 \in \text{Time}$. (Reprint from [LÁÖ15])

A1. $(t_1 + t_2) + t_3 = t_1 + (t_2 + t_3)$	A9. $0 \leq t_1$
A2. $t_1 + 0 = t_1$	A10. $t_1 \leq t_2 \iff \exists t_3 : t_1 + t_3 = t_2 \wedge t_3 = t_2 - t_1$
A3. $t_1 + t_2 = t_2 + t_1$	A11. $\max(t_1, t_2) = \max(t_2, t_1)$
A4. $t_1 < t_2 \vee t_1 = t_2 \vee t_2 < t_1$	A12. $\max(\max(t_1, t_2), t_3) = \max(t_1, \max(t_2, t_3))$
A5. $\neg(t_1 < t_1)$	A13. $t_1 \leq t_2 \implies \max(t_1, t_2) = t_2$
A6. $t_1 < t_2 \implies t_1 + t_3 < t_2 + t_3$	A14. $\min(t_1, t_2) = \min(t_2, t_1)$
A7. $t_1 + t_2 = t_1 + t_3 \implies t_2 = t_3$	A15. $\min(\min(t_1, t_2), t_3) = \min(t_1, \min(t_2, t_3))$
A8. $t_1 \leq t_2 \iff t_1 < t_2 \vee t_1 = t_2$	

ε and ∞ $\text{TIME}_\varepsilon^\infty$ extends TIME by adding the infinitesimal element ε and infinity element ∞ , s.t. $\text{Time}_\varepsilon^\infty = \text{Time} \cup \{\varepsilon, \infty\}$ and $\varepsilon, \infty \notin \text{TIME}$. TIME 's operators $<$, $+$, $\dot{-}$, \max and \min are adapted, to include operations involving ε and ∞ , as shown by the axioms in Table B.2 and Table B.3 definitions for $\text{TIME}_\varepsilon^\infty$.

$$\begin{array}{ll} < : \text{Time}_\varepsilon^\infty \times \text{Time}_\varepsilon^\infty \rightarrow \mathbb{B} & + : \text{Time}_\varepsilon^\infty \times \text{Time}_\varepsilon^\infty \rightarrow \mathbb{B} \\ \dot{-} : \text{Time}_\varepsilon^\infty \times \text{Time}_\varepsilon^\infty \rightarrow \text{Time}_\varepsilon^\infty & \min : \text{Time}_\varepsilon^\infty \times \text{Time}_\varepsilon^\infty \rightarrow \text{Time}_\varepsilon^\infty \\ \max : \text{Time}_\varepsilon^\infty \times \text{Time}_\varepsilon^\infty \rightarrow \text{Time}_\varepsilon^\infty & \end{array}$$

Table B.2 – ∞ -axioms in the theory $\text{TIME}_\varepsilon^\infty$, for $t_1 \in \text{Time}$ and $t_2, t_3 \in \text{Time}_\infty$. (Reprint from [LÄÖ15])

A16. $t_1 < \infty$	A20. $\infty + t_2 = \infty$
A17. $\neg(\infty \leq t_1)$	A21. $\infty \dot{-} t_1 = \infty$
A18. $t_2 \leq \infty$	A22. $\max(t_2, \infty) = \infty$
A19. $\neg(\infty < t_2)$	A23. $\max(t_2, t_3) = t_2 \implies \min(t_2, t_3) = t_3$

Table B.3 – ε -axioms in the theory $\text{TIME}_\varepsilon^\infty$, for $t_1 \in \text{Time}$ and $t_2 \in \text{Time}_\varepsilon^\infty$, $0 < t_1, t_2$. (Extension of [LÄÖ15])

A24. $0 < \varepsilon$	A27. $\varepsilon \leq t_2$
A25. $\varepsilon < t_1$	A28. $\neg(\varepsilon < t_2)$
A26. $\neg(t_1 \leq \varepsilon)$	A29. $\min(t_2, \varepsilon) = \varepsilon$

Based on this theory, it is possible to define time variables as $t, t', t_1, \dots \in \mathbb{T}$ to denote time values. It is further possible to define other operations such as multiplication ($k * t = t_1 + t_2 + \dots + t_k$, where $t_1 = t_2 = \dots = t_k$), etc.

gcd Further, the gcd operation has been axiomatised to allow the calculation of the greatest common divisor of two numbers. It is based on the following operators, where $\text{Time}_{>\varepsilon} = \text{Time} \setminus \{0, \varepsilon\}$ (adaptation of [LÄÖ15]):

$$\begin{array}{l} | : \text{Time}_{>\varepsilon} \times \text{Time}_{>\varepsilon} \rightarrow \mathbb{B} \\ \text{gcd} : \text{Time}_{>\varepsilon} \times \text{Time}_{>\varepsilon} \rightarrow \text{Time}_{>\varepsilon} \\ \text{half} : \text{Time}_{>\varepsilon} \rightarrow \text{Time}_{>\varepsilon} \end{array}$$

Note that contrary to Lepri et al. gcd is not defined in its own theory (e.g. TIME^{gcd}), but is already part of $\text{TIME}_\varepsilon^\infty$. Its axioms are defined in Table B.4.

Table B.4 – *gcd*-axioms in the theory $\text{TIME}_\varepsilon^\infty$. $t_1, t_2, t_3 \in \text{Time}_{>\varepsilon}$. (Adapted from [LÄÖ15])

A30. $t_1 \mid t_2 \wedge t_2 \mid t_3 \implies t_1 \mid t_3$	A30. $\text{gcd}(t_1, t_2) = \text{gcd}(t_2, t_1)$
A31. $t_1 \mid t_2 \wedge t_2 \mid t_1 \implies t_1 = t_2$	A31. $\text{gcd}(\text{gcd}(t_1, t_2), t_3) = \text{gcd}(t_1, \text{gcd}(t_2, t_3))$
A32. $t_1 \mid t_1$	A32. $\text{gcd}(t_1, t_2) \mid t_1$
A33. $t_1 \mid t_2 \wedge t_1 \mid t_3 \implies t_1 \mid (t_2 + t_3)$	A33. $(t_3 \mid t_1 \wedge t_3 \mid t_2) \implies t_3 \mid \text{gcd}(t_1, t_2)$
A34. $t_2 < t_1 \implies \neg(t_1 \mid t_2)$	A34. $\text{half}(t_1) + \text{half}(t_1) = t_1$
A35. $t_1 \mid (t_1 + t_2) \implies t_1 \mid t_2$	

Time Intervals

Finally, we define the notion of time intervals on the time base \mathbb{T} . Time intervals are denoted as $[a, b]$, $(a, b]$, $[a, b_\infty)$ or (a, ∞) where $a, b \in \text{Time}$ and $b_\infty \in \text{Time}_\varepsilon^\infty$. Intervals follow the usual notation, where $[$ and $($ are the inclusive and exclusive lower bounds of the interval, and $]$ and $)$ define the inclusive and exclusive upper bounds, respectively. The set of all intervals of \mathbb{T} is written as $\text{Intervals}(\mathbb{T})$. We further call the smallest (resp. largest) member of the interval infimum (resp. supremum). Their definition is:

$$\begin{aligned} \text{inf}(I) &= \max\{t \in \mathbb{T} \mid \forall t' \in I : t \leq t'\} \text{ to be the infimum of } I \\ \text{sup}(I) &= \min\{t \in \mathbb{T} \mid \forall t' \in I : t' \leq t\} \text{ to be the supremum of } I \end{aligned}$$

Accordingly, the values of an interval $I \in \text{Intervals}(\mathbb{T})$ are all values that lie between the infimum and supremum:

$$(\forall t \in \mathbb{T} : \text{inf}(I) < t < \text{sup}(I) \implies t \in I)$$

Appendix C

Code listings

This chapter provides source code listings that could not fit into the main document, due to spatial reasons. These code listings are also available online at `crestdsl`'s online repository <https://github.com/crestdsl/thesis-code>. The repository further links to a live demo that can be executed straight from the browser.

C.1 `crestdsl` – Listings

Listing C.1 provides a listing of an entity with subentities. Notably, the two `GenericLamp` subentities are initialised instances assigned to class attributes (Line 11 – 12), similar to ports, states and other CREST concepts. The listing also shows that CREST updates can be used to set the subentity input ports. The subentity outputs are propagated to the `LampComposition`'s outputs using influences. These influences can be defined in two ways:

1. Using the decorator `@crest.influence` to annotate the influence function's definition (Lines 53 – 55)
2. by creating an `crest.Influence` object and optionally passing a function or lambda expression as function parameter. Omitting this parameter results in the use of the default function, which returns the source port's value.

Listing C.1 – A composed entity, with two subentities. Full source code of the example provided in Listing 5.7.

```

1  class LampComposition(crest.Entity):
2      # inputs
3      switch_input = crest.Input(resource=switch, value="off")
4      in_port = crest.Input(resource=watt, value=100)
5
6      # subentities
7      big_lamp = GenericLamp(300)
8      small_lamp = GenericLamp(100, .9)
9
10     # outputs
11     big_out = crest.Output(watt, 0)
12     small_out = crest.Output(watt, 0)
13
14     # states
15     on = crest.State()
16     off = crest.State()
17     current = off
18
19     # transitions
20     @crest.transition(source=off, target=on)
21     def off_to_on(self):
22         return self.switch_input.value == "on"
23
24     @crest.transition(source=on, target=off)
25     def on_to_off(self):
26         return self.switch_input.value != "on"
27
28     # setting of subentity inputs
29     @crest.update(state=on, target=small_lamp.in_port)
30     def set_small_lamp_input_when_on(self, dt):
31         if self.in_port.value > 100:
32             return 100
33         else:
34             return 0
35
36     @crest.update(state=off, target=small_lamp.in_port)
37     def set_small_lamp_input_when_off(self, dt):
38         return 0
39
40     @crest.update(state=on, target=big_lamp.in_port)
41     def set_big_lamp_input_when_on(self, dt):
42         if self.in_port.value < 100:
43             return 0
44         else:
45             return self.in_port.value - 100
46
47     @crest.update(state=off, target=big_lamp.in_port)
48     def set_big_lamp_input_when_off(self, dt):
49         return 0
50
51
52     # connect subentity output to entity output
53     @crest.influence(source=big_lamp.out_port, target=big_out)
54     def forward_big_output(value):
55         # influences only take one parameter: value
56         return value
57
58     forward_small_output = crest.Influence(
59         source=small_lamp.out_port, target=small_out)

```

C.2 Simulation – Listings

The heating module described in Section 5.3 – Simulation is a component that is usually built into a growing lamp, such as the one introduced in Chapter 4 – The CREST Language. Compared to the `HeatElement` of Figure 4.1, this `HeatModule` is an advanced version that features multiple states, including an error-state which is entered when the device overheats. From a technical point of view, it is an electrical heater with an efficiency factor of 0.1. This means, it converts one tenth of the energy provided to its `electricity` input into heating energy output, when the system is in state `On` and at least 200 watts electricity are applied. If the system is in state `Off`, the output is 0 watts. The transitions between on and off are controlled by the `switch` input. Additionally, the transition to `Off` is also automatically triggered after the module spent 30 minutes in state `On`, or when the electricity input value drops below 200 watts. The timing behaviour is implemented using a `timer` port. In the on-state, an update continuously grows the timer's value. When the system is turned off, the timer's value is reduced by double the rate. The system can only switch to `On` again, if the timer value reaches 0.

Unfortunately, this heat module is fragile and breaks if the internal temperature exceeds a threshold of 400 degrees Celsius. This behaviour is modelled by a transition to an `Error` state. Once the system reaches `Error`, it cannot leave this state anymore and produces an output of 0 watts.

The `internal_temperature` value is determined according to the amount of electricity fed into the system. The system is well-equipped to deal with electrical power up to 200 watts. If the electricity input exceeds this value, the heating's internal temperature will grow by one tenth of a degree per excess watt per time unit. If the power input drops below 200 watts or the heat module is turned off, the temperature will sink by the same rate (20°C per minute, if the system is `Off` or the `electricity` is 0 watts).

The CREST diagram of the heat module is shown in Figure 5.2. The full `crestdsl` source code, which also specifies the behaviour of the various update functions within the system is provided in Listing C.2.

Listing C.2 – Source code of the heat module example

```

1 # required resources
2 onOff = crest.Resource(unit="onOff", domain=["on", "off"])
3 watt = crest.Resource(unit="Watt", domain=crest.REAL)
4 celsius = crest.Resource(unit="Celsius", domain=crest.REAL)
5 time = crest.Resource(unit="Time", domain=crest.REAL)
6
7 class HeatModule(crest.Entity):
8     switch = crest.Input(resource=onOff, value="on")
9     electricity = crest.Input(resource=watt, value=0)
10    internal_temp = crest.Local(resource=celsius, value=0)
11    timer = crest.Local(resource=time, value=0)
12    heating = crest.Output(resource=watt, value=0)
13    # states
14    off = current = crest.State()
15    on = crest.State()
16    error = crest.State()
17    # transitions
18    @crest.transition(source=off, target=on)
19    def to_on(self):
20        return self.switch.value == "on" and self.timer.value <= 0 \
21            and self.electricity.value >= 200
22    @crest.transition(source=on, target=off)
23    def to_off(self):
24        return self.switch.value != "on" or self.timer.value >= 30 \
25            or self.electricity.value < 200
26    @crest.transition(source=on, target=error)
27    def to_error(self):
28        return self.internal_temp.value >= 400
29    # updates for heat energy output
30    @crest.update(state=on, target=heating)
31    def on_update_output(self, dt):
32        # 50 per cent efficiency
33        return self.electricity.value * 0.5
34    @crest.update(state=off, target=heating)
35    def off_update_output(self, dt):
36        return 0
37    @crest.update(state=error, target=heating)
38    def error_update_output(self, dt):
39        return 0
40    # update timer:
41    @crest.update(state=on, target=timer)
42    def on_update_timer(self, dt):
43        return self.timer.value + dt
44    @crest.update(state=off, target=timer)
45    def off_update_timer(self, dt):
46        new_value = self.timer.value - 2 * dt
47        if new_value <= 0: # don't go below 0
48            return 0
49        else:
50            return new_value
51    # updates for internal_temp
52    @crest.update(state=on, target=internal_temp)
53    def on_update_internal_temp(self, dt):
54        # if more than 200 watt, we grow
55        # one tenth degree per extra watt per time unit
56        # if lower, we sink at the same rate
57        factor = (self.electricity.value - 200) / 10
58
59        if self.electricity.value >= 200:
60            return self.internal_temp.value + factor * dt
61        else:
62            new_value = self.internal_temp.value + factor * dt
63            return max(new_value, 22) # don't go below 22
64    @crest.update(state=[off,error], target=internal_temp)
65    def off_error_update_internal_temp(self, dt):
66        # see formula above
67        new_value = self.internal_temp.value - 20 * dt
68        return max(new_value, 22) # don't go below 22

```

C.3 ThreeMasses – A Non-linear System

The following listings show the source code of the ThreeMasses system. It models three masses that are placed on a surface (e.g. a table). Initially one of them has initial horizontal velocity (i.e. it is rolling on the surface) until it collides with the second and transfers its momentum. The second one then collides with the third mass and pushes it off the table. The third mass then falls off the table at which point it repeatedly bounces off the floor, iteratively reducing its energy (and thus bouncing height). The system models the horizontal and vertical positions of each mass.

Resources (Physical Units) First we require the definition of resources such as position (in metres), velocity (metres per second) and acceleration (metres per second per second), the mass (in kilograms) and the restitution factor.

Listing C.3 – Resource definitions of the ThreeMasses system

```

1 m = crest.Resource("m", crest.REAL)           # meters (position)
2 mps = crest.Resource("m/s", crest.REAL)       # meters per second (velocity)
3 mps2 = crest.Resource("m/s/s", crest.REAL)    # meters per second per second
         (acceleration)
4 kg = crest.Resource("kg", crest.REAL)        # kilograms
5 restitution = crest.Resource("factor", crest.REAL) # the restitution factor

```

Model Parameters These are model parameters. They can be used to set initial conditions and similar, such as table height or table length. Note that these variables can only be used as initial values for ports or e.g. in `__init__` functions. They cannot be used in transition guards, updates, actions or influences, since the simulator cannot access their value and cannot calculate the correct next transition time.

Listing C.4 – Parameter definitions ThreeMasses system

```

1 # These variables are used as default values for ports
2 global e
3 e = 0.9 # restitution factor
4 L = 7 # table length
5 H = 3 # table height

```

Model of Individual Masses A simple mass entity. It is placed at a certain (x,y) position with a specific velocity (vx, vy) and acceleration (ax, ay). The x-velocity can be set using an input vx_in. If vx_in != 0 it will override the value currently held by the entity. It offers its x and vx values in the ports x_out and vx_out.

If the mass reaches the end of the table (x == L) the entity switches to the *falling* state, modifies its y-acceleration and hence also its y-position. From that moment it becomes similar to a bouncing-ball experiment, except that its x-value also changes. On every bounce, the restitution factor slows the velocity of both x and y by the restitution factor.

Listing C.5 – Source code of one Mass entity

```

1 class Mass(crest.Entity):
2     """----- Constructor -----"""
3     def __init__(self, x0, vx0=0):
4         """We have to provide an initial x-position on the table and an initial x-
5         velocity"""
6         self.x.value = x0
7         self.vx.value = vx0
8
9     """----- PORTS -----"""
10
11     e = crest.Local(restitution, e) # restitution factor
12     L = crest.Local(m, L) # table length
13     H = crest.Local(m, H) # table height
14
15     x = crest.Local(m, 0) # the position (init should be an input param)
16     y = crest.Local(m, 3) # the height of the table (should be an input param)
17
18     x_out = crest.Output(m, 0) # propagate output
19     vx_out = crest.Output(mps, 0) # propagate output
20     vx_in = crest.Input(mps, 0) # to set the speed from the outside
21
22     vx = crest.Local(mps, 3) # this should be an input param
23     vy = crest.Local(mps, 0) # this should be an input param
24
25     ay = crest.Local(mps2, 0) # acceleration
26
27     """----- INFLUENCES -----"""
28
29     propagate_x_out = crest.Influence(source=x, target=x_out)
30     propagate_vx_out = crest.Influence(source=vx, target=vx_out)
31
32     """----- STATES -----"""
33
34     on_table = current = crest.State()
35     falling = crest.State() # downward movement
36     bouncing = crest.State() # upward movement
37
38     """----- TRANSITIONS -----"""
39
40     fall_off_table = crest.Transition(source=on_table, target=falling, guard=(
41     lambda self: self.x.value == self.L.value and self.vx.value > 0))
42
43     # actually we should find when the vy == 0 for the guard.
44     # However, Python's floats aren't as precise as Z3's Real datatype, so a == 0
45     # can provide wrong results.
46     # it is safer to use inequalities when comparing floats (duh...)
47     fall = crest.Transition(source=bouncing, target=falling, guard=(lambda self:
48     self.vy.value <= 0))
49
50     # We should use an inequality here as well (y <= 0)
51     # However, since our system is non-linear (the y position changes with dt^2),
52     # this means that the optimization is non-linear. And Z3 is just not good at
53     # that!
54     # We therefore use the following trick: We try to find the place where the
55     # absolute value is very close to 0
56     # Why don't you try setting it to 10 ** -10, and executing the simulation
57     # again?
58     bounce = crest.Transition(source=falling, target=bouncing, guard=(lambda self:
59     abs(self.y.value) < 10**-3))
60
61     """----- Actions & Updates -----"""
62
63     # bounce actions
64     @crest.action(transition=bounce, target=vx)
65     def action_apply_vx_restitution(self):
66         """on bounce we apply restitution to vx"""
67         return self.vx.pre * self.e.value # apply restitution factor

```

```

59
60 @crest.action(transition=bounce, target=vy)
61 def action_vy_bouncing(self):
62     """on bounce we apply restitution to vy and also inverse the force to
63     change the direction """
64     return self.vy.pre * self.e.value * -1 # use restitution and inverse
65     force
66
67 # X value updates
68
69 @crest.update(state=[on_table, falling, bouncing], target=x)
70 def update_x(self, dt):
71     """ The X value is the previous x value + average velocity * dt"""
72     return self.x.pre + (self.vx.value + self.vx.pre) / 2 * dt
73
74 @crest.update(state=[on_table, falling, bouncing], target=vx)
75 def update_vx(self, dt):
76     """ If we have an external value set to vx_in, then we use that one. If
77     the external value is 0, then we continue using the internal value."""
78     if self.vx_in.value != 0: # external setting of speed
79         return self.vx_in.value
80     else:
81         return self.vx.pre
82
83 # Y value updates
84
85 @crest.update(state=[falling, bouncing], target=ay)
86 def update_ay_falling(self, dt):
87     """change the acceleration to the value of gravity (rounded)"""
88     return -9.81
89
90 @crest.update(state=[falling, bouncing], target=y)
91 def update_y(self, dt):
92     """The new position is the old position plus average velocity times passed
93     time"""
94     average_velocity = (self.vy.value + self.vy.pre) / 2.0
95     return self.y.pre + average_velocity * dt # traversed distance = (v0+
96     v_end)/2*t
97
98 @crest.update(state=[falling, bouncing], target=vy)
99 def update_vy(self, dt):
100     """The average velocity is the previous velocity plus acceleration times
101     time.
102     Note that velocity here can be positive and negative, depending on whether
103     we're falling or bouncing up again"""
104     return self.vy.pre + self.ay.value * dt

```

The Three Masses System This system initialises three equal masses (their measured mass is defined in port m). There are three individual states: the masses do not touch, mass1 touches mass2 and mass2 touches mass3 (In this model it is not possible for all masses to touch at the same time!)

The transitions are based on the x-positions. If their distance is lower than a certain threshold and the first one is faster than the second one, we call it a collision. Then some actions will kick in and change the involved masses' velocities. (i.e. set them externally). If they don't touch, we use updates to continuously set their `vx_in` to zero, so that the masses calculate `vx` themselves.

Listing C.6 – Source code of the non-linear ThreeMasses system

```

1
2 class ThreeMasses(crest.Entity):
3     """ - - - - - PORTS - - - - - """
4
5     e = crest.Local(restitution, e) # restitution factor
6     m = crest.Local(kg, 1) # the actual mass of each object is 1 kg, we assume
7     # all masses are equal. Otherwise we need to adapt our system
8
9     """ - - - - - SUBENTITIES - - - - - """
10    mass1 = Mass(x0=0, vx0=3) # mass1 is placed at 0 but has a velocity
11    mass2 = Mass(x0=6.5) # mass2 is placed at 6.5 (close to the end) but
12    # does not move
13    mass3 = Mass(x0=7) # mass3 is placed right at the edge (7), but
14    # does not fall because its velocity is 0
15
16    """ - - - - - STATES - - - - - """
17
18    no_touch = current = crest.State() # no collisions
19    m1_touch_m2 = crest.State() # when mass1 hits mass2
20    m2_touch_m3 = crest.State() # when mass2 hits mass3
21
22    """ - - - - - TRANSITIONS - - - - - """
23
24    # same as above. the x_out values are based on non-linear constraints
25    collide_m1_m2 = crest.Transition(source=no_touch, target=m1_touch_m2,
26    guard=(lambda self: abs(self.mass1.x_out.value - self.mass2.x_out.value) <
27    0.1 * 10**-3 and self.mass1.vx_out.value > self.mass2.vx_out.value ))
28    m1_m2_collision_to_no_touch = crest.Transition(source=m1_touch_m2, target=
29    no_touch, guard=(lambda self: abs(self.mass1.x_out.value - self.mass2.x_out.
30    value) > 0.1 * 10**-3 and self.mass1.vx_out.value <= self.mass2.vx_out.value
31    ))
32    collide_m2_m3 = crest.Transition(source=no_touch, target=m2_touch_m3,
33    guard=(lambda self: abs(self.mass2.x_out.value - self.mass3.x_out.value) <
34    0.1 * 10**-5 and self.mass2.vx_out.value > self.mass3.vx_out.value ))
35    m2_m3_collision_to_no_touch = crest.Transition(source=m2_touch_m3, target=
36    no_touch, guard=(lambda self: abs(self.mass2.x_out.value - self.mass3.x_out.
37    value) > 0.1 * 10**-3 and self.mass2.vx_out.value <= self.mass3.vx_out.value
38    ))
39
40    """ - - - - - UPDATES & ACTIONS - - - - - """
41
42    @crest.update(state=no_touch, target=mass1.vx_in)
43    def no_touch_mass1vx_in(self, dt):
44        # if the masses don't touch, then don't change their velocity
45        return 0
46
47    @crest.update(state=no_touch, target=mass2.vx_in)
48    def no_touch_mass2vx_in(self, dt):
49        # if the masses don't touch, then don't change their velocity
50        return 0
51
52

```

```
40 @crest.update(state=no_touch, target=mass3.vx_in)
41 def no_touch_mass3vx_in(self, dt):
42     # if the masses don't touch, then don't change their velocity
43     return 0
44
45 @crest.action(transition=collide_m1_m2, target=mass1.vx_in)
46 def m1_collide_m2_action_m1_vx(self):
47     # on collision with mass2 change mass1's velocity
48     m = self.m.value # read the current port values into local variables
49     e = self.e.value # this makes the formula below easier to read
50     return self.mass1.vx_out.value * (m - e * m) / (2 * m) + self.mass2.vx_out
51     .value * m * (1 + e) / (2*m)
52
53 @crest.action(transition=collide_m1_m2, target=mass2.vx_in)
54 def m1_collide_m2_action_m2_vx(self):
55     # on collision with mass1 change mass2's velocity
56     m = self.m.value
57     e = self.e.value
58     return self.mass1.vx_out.value * (1 + e) * m / (2*m) + self.mass2.vx_out.
59     value * (m - e * m) / (2*m)
60
61 @crest.action(transition=collide_m2_m3, target=mass2.vx_in)
62 def m2_collide_m3_action_m1_vx(self):
63     # on collision with mass3 change mass2's velocity
64     m = self.m.value
65     e = self.e.value
66     return self.mass2.vx_out.value * (m - e * m) / (2*m) + self.mass3.vx_out.
67     value * m * (1 + e) / (2*m)
68
69 @crest.action(transition=collide_m2_m3, target=mass3.vx_in)
70 def m2_collide_m3_action_m2_vx(self):
71     # on collision with mass2 change mass3's velocity
72     m = self.m.value
73     e = self.e.value
74     return self.mass2.vx_out.value * (1 + e) * m / (2*m) + self.mass3.vx_out.
75     value * (m - e * m) / (2*m)
```


Appendix D

Acronyms and Symbols

Acronyms











AADL	Architecture Analysis & Design Language
ADL	architecture description language
AP	atomic proposition
API	application programming interface
CPN	coloured Petri net
CPS	cyber-physical system
CREST	Continuous REactive SysTems language
CTL	computation tree logic
DD	decision diagram
DEVS	Discrete Event System Specification
DSL	domain-specific language
DSML	domain-specific modelling language
EADL	Embedded Architecture Description Language
ER diagram	entity-relationship diagram
FSA	finite state automaton
FSM	finite state machine
GPPL	general-purpose programming language
GUI	graphical user interface
HA	hybrid automaton
HDL	hardware description language

HLPN	high-level Petri net
HS	hybrid system
IDE	integrated development environment
IoT	Internet of Things
LTL	linear temporal logic
MARTE	UML Profile for Modeling and Analysis of Real Time and Embedded systems
MBE	model-based engineering
MDA	model-driven architecture
MDD	model-driven development
MDE	model-driven engineering
ODE	ordinary differential equation
OMG	Object Management Group
PN	Petri net
QSS	quantized state system
RTE	real-time and embedded
RTL	register-transfer level
SDL	Specification and Description Language
SMT	satisfiability modulo theories
SOS	structured operational semantics
SPT	UML Profile for Schedulability, Performance, and Time Specification
SysML	Systems Modeling Language
TA	timed automaton
TCTL	timed computation tree logic
TLM	transaction-level modeling
UML	Unified Modeling Language

List of Symbols

This thesis uses various symbols and icons. This is a summary of the most important ones. For some chapters it might be interesting to keep it close at hand.

Symbols used in CREST Diagrams

-  A CREST entity is visualised by its boundaries. Usually, its name/type is provided as a title inside a coloured bar.
-  An input port of a CREST entity.
-  An output port of a CREST entity.
-  A local port of a CREST entity.
-  A state of a CREST entity's behaviour automaton.
-  The current or initial state of a CREST entity's behaviour automaton.
-  A behaviour automaton transition connecting a state to a state. Transitions model state changes and are annotated with guard function names.
-  A (continuous) update relation connecting a state to a port. It modifies the target port's value when the automaton is in the corresponding state.
-  An influence function connecting a port to another port. Influences propagate a port's value to the target port independent from the state that the automaton is in.
-  An action connecting a transition and a port. An action changes its target port's value when the transition is triggered.

Symbols used for CREST's Formalisation

The list is sorted to match the order of definition in Chapter 4. Note that only recurring definitions are provided here.

- \sqcup Set partition operator; $S = \sqcup_i S_i$ or $S = S_1 \sqcup \dots \sqcup S_n$ iff $\forall i, j, i \neq j \implies S_i \cap S_j = \emptyset$ and $S = \bigcup_{1 \leq i \leq n} S_i$.

\mathbb{T}	A time base, e.g. $\mathbb{R}_{>0}$ or $\mathbb{Q}_{>0}$, extended by the infinitesimal value ε and infinity ∞ .
<i>Units</i>	A set of resource units; e.g. <i>Watt</i> , <i>Switch</i> , <i>Celsius</i> .
<i>Domains</i>	A set of value domains; e.g. \mathbb{R} , \mathbb{N} , {on, off}.
<i>Types</i>	A set of resource types used in a CREST system; $Types \subseteq Domains \times Units$; e.g. $\mathbb{R}Watt$, {on, off}Switch.
<i>Resources</i>	The set of resource (values), e.g. <i>0Watt</i> , <i>onSwitch</i> .
<i>Entities</i>	The set of entities that makes up a CREST system.
<i>root</i>	Returns the root of a CREST system's entity hierarchy.
<i>children(e)</i>	Returns the direct subentities of a CREST entity.
$Ports_e$	The set of ports of a single CREST entity e .
$Ports_e^I$	An entity e 's set of input ports.
$Ports_e^O$	An entity e 's set of output ports.
$Ports_e^L$	An entity e 's set of local ports.
<i>sources(e)</i>	Returns the ports that can be read by e 's updates and guards, i.e. e 's inputs, locals and e 's subentities' outputs.
<i>targets(e)</i>	Returns the ports that can be written by e 's updates, i.e. e 's outputs, locals and e 's subentities' inputs.
<i>binding(p)</i>	Maps ports onto their current values; e.g. <i>binding</i> (electricity) = <i>0Watt</i> .
<i>pre</i>	A binding that stores the ports' previous values (e.g. before an update execution).
$States_e$	An entity e 's set of automaton states.
<i>Transitions</i>	The set of all transitions of the form $State_e \times State_e \times \mathcal{T}$ defined for a CREST system.
\mathcal{T}	The set of transition guard names (i.e. not their implementations).
τ	A function that maps transition guard names to guard implementations.
<i>Updates</i>	The set of all updates of the form $State_e \times Port_e \times \mathcal{U}$ defined for a CREST system.
\mathcal{U}	The set of update function names (i.e. not their implementations).
ν	A function that maps update function names to their implementations.

- dependencies* Maps update function names to the *sources*-ports they depend on.
- io-dependencies* Defines dependencies between an entity's output and input ports. This is required for the resolution of cyclic dependencies that might be occurring due to CREST's black box view of subentities.
- current*(e) Returns the current automaton state of an entity e .
- W The set of all states a CREST system can be in.
 $W = Currents \times Bindings \times Bindings \times \mathbb{T}$
- w w usually refers to a CREST system's state. $w = \langle current, bind, pre, t \rangle$
- $w[e \mapsto s]$ Change the current state of entity e to s ($s \in States_e$).
- $w[p \mapsto r]$ Change the port value binding of port p to resource value r .
- $w[vs]$ Change of several port value bindings, where $vs = \{f : Ports \rightarrow Resources\}$.
- δt A variable used to refer to a time interval $\delta t \in \mathbb{T}$ e.g. the time to advance during simulation.
- ε $\varepsilon \in \mathbb{T}$ represents the infinitesimal value, i.e. the smallest possible value that is larger than zero. Formally: $\varepsilon > 0 \wedge \nexists \tau \in \mathbb{T}, \tau < \varepsilon$

Symbols used in the Verification chapter

- \neg, \wedge, \vee Logical operators used in computation tree logic (CTL) and timed computation tree logic (TCTL) formulas.
- $AP, p \in AP$ The set of atomic propositions, i.e. the predicates of a system state.
- EX, EU, AU Basic CTL operators. Other operators can be defined as based of these ones.
- $\phi, \psi, \phi_1, \phi_2$ Variables that represent computation tree logic (CTL) or TCTL formulas.
- EU_I, AU_I Timed versions of CTL operators that are used in TCTL formulas, annotated with an interval I .
- \mathcal{TK} The symbol used for a timed Kripke structure.
- \mathcal{CK} The symbol used for a CREST Kripke structure (an adapted \mathcal{TK}).
- \models Operator that represents the satisfaction relation between a TCTL formula and a \mathcal{CK} state.

Scientific Work and Publications

This section lists a summary of the publications for CREST. Next to the research on CREST and development of `crestdsl`, I was involved in several other projects and collaborations that fruited in scientific papers and the creation of hands-on artefacts. The work on these topics will also be outlined below.

CREST

Before composing this thesis, CREST and `crestdsl` have been the subject of several publications presented at international conferences and workshops. Here is a list of my publications that are relevant towards this topic:

- [KLB17] **Stefan Klikovits**, Alban Linard, and Didier Buchs. CREST - A Continuous, REactive SysTEms DSL. In: *5th International Workshop on the Globalization of Modeling Languages (GEMOC 2017)*, CEUR Workshop Proceedings, vol. 2019, pp. 286–291. 2017. URL: http://ceur-ws.org/Vol-2019/gemoc_2.pdf.

- [KLB18a] **Stefan Klikovits**, Alban Linard, and Didier Buchs. CREST - A DSL for Reactive Cyber-Physical Systems. In: *10th System Analysis and Modeling Conference (SAM 2018)*, Lecture Notes in Computer Science, vol. 11150, pp. 29–45. Springer, 2018. DOI: [10.1007/978-3-030-01042-3_3](https://doi.org/10.1007/978-3-030-01042-3_3).

- [KLB18b] **Stefan Klikovits**, Alban Linard, and Didier Buchs. *CREST Formalization*. Tech. rep. Software Modeling and Verification Group, University of Geneva, 2018. DOI: [10.5281/zenodo.1284561](https://doi.org/10.5281/zenodo.1284561).

- [KCB18] **Stefan Klikovits**, Aurélien Coet, and Didier Buchs. ML4CREST: machine learning for CPS models. In: *2nd International Workshop on Model-Driven Engineering for the Internet-of-Things (MDE4IoT) at MODELS'18*, CEUR Workshop Proceedings, vol. 2245, pp. 515–520. 2018. URL: http://ceur-ws.org/Vol-2245/mde4iot_paper_4.pdf.

CERN

In the first phase of my PhD, I was placed at CERN, the European Organization for Nuclear Research¹, where I performed research on static code analysis and automated generation of unit tests. The project domain was a large code base (roughly 1 million lines of code) written in a proprietary, ANSI-C-like language, and for which no automated unit testing framework existed yet. The software under test is used to control various applications at CERN, such as parts of the Large Hadron Collider and CERN's electrical power grid. The project resulted in the creation of a prototype application, that is capable of creating unit test input for individual code functions and small code units. This is achieved by separating a routine from its dependencies (e.g. global variables, subroutines). The resulting code was translated to C# for execution using Microsoft Research' Pex code exploration tool to provide test inputs and expected outputs. The theoretical aspects of this work were presented in three scientific papers.

- [Kli+15] **Stefan Klikovits**, David P. Y. Lawrence, Manuel Gonzalez-Berges, and Didier Buchs. Considering Execution Environment Resilience: A White-Box Approach. In: *Software Engineering for Resilient Systems - 7th International Workshop, SERENE 2015*, Lecture Notes in Computer Science, vol. 9274, pp. 46–61. Springer, 2015. DOI: [10.1007/978-3-319-23129-7_4](https://doi.org/10.1007/978-3-319-23129-7_4).
- [Kli+16] **Stefan Klikovits**, David P. Y. Lawrence, Manuel Gonzalez-Berges, and Didier Buchs. Automated Test Case Generation for the CTRL Programming Language Using Pex: Lessons Learned. In: *Software Engineering for Resilient Systems - 8th International Workshop, SERENE 2016*, Lecture Notes in Computer Science, vol. 9823, pp. 117–132. Springer, 2016. DOI: [10.1007/978-3-319-45892-2_9](https://doi.org/10.1007/978-3-319-45892-2_9).
- [KGB17] **Stefan Klikovits**, Manuel Gonzalez-Berges, and Didier Buchs. Towards Language Independent (Dynamic) Symbolic Execution. In: *Proceedings of the 24th PhD Mini-Symposium*, pp. 50–53. Budapest University of Technology and Economics, Department of Measurement and Information Systems, 2017. DOI: [10.5281/zenodo.291899](https://doi.org/10.5281/zenodo.291899).

¹<https://home.cern>

Other Work

Next to the work on my PhD projects, I contributed to other subjects that did not directly influence the findings of this thesis but allowed me to create a network of international research collaborations, exchange with experts in other domains and extend my knowledge. This section lists publications that resulted from my efforts.

MPM4CPS As a member of the European COST Action IC1404's "Multiparadigm Modelling of Cyber-Physical Systems" (MPM4CPS)², I joined the Working Group 1 on "Foundations". I was actively involved in the creation and documentation of two reports that were submitted as final deliverables. The first one is a catalogue of the state-of-the-art of formalisms, languages and tools used in the modelling and simulation domain. I wrote and reviewed over two-thirds of the definitions that ended up in the final report.

[Kli+19] **Stefan Klikovits**, Rima Al-Ali, Moussa Amrani, Ankica Barisic, Fernando Barros, Dominique Blouin, Etienne Borde, Didier Buchs, Holger Giese, Miguel Goulao, Mauro Iacono, Florin Leon, Eva Navarro, Patrizio Pelliccione, and Ken Vanherpen. COST IC1404 WG1 Deliverable WG1.1: State-of-the-art on Current Formalisms used in Cyber-Physical Systems Development. 2019. DOI: [10.5281/zenodo.2533455](https://doi.org/10.5281/zenodo.2533455).

I further contributed actively to the conceptualisation and creation of a framework that establishes relations between modelling languages and techniques. I am a co-author of the resulting report.

[ALA+19] Rima Al-Ali, Moussa Amrani, Soumyadip Bandyopadhyay, Ankica Barisic, Fernando Barros, Dominique Blouin, Ferhat Erata, Holger Giese, Mauro Iacono, **Stefan Klikovits**, Eva Navarro, Patrizio Pelliccione, Kuldar Taveter, Bedir Tekinerdogan, and Ken Vanherpen. COST IC1404 WG1 Deliverable WG1.2: Framework to Relate / Combine Modeling Languages and Techniques. 2019. DOI: [10.5281/zenodo.2527576](https://doi.org/10.5281/zenodo.2527576).

Additionally, the MPM4CPS Action pushed towards the publication of a text book related to its core topic, which is currently in the process of publication. The book targets the "Foundations of Multiparadigm Modelling for Cyber-Physical Systems". I co-authored a chapter that focuses on the use of Petri nets for the concurrent, non-deterministic systems.

[BKL19] Didier Buchs, **Stefan Klikovits**, and Alban Linard. Petri Nets: A Formal Language to Specify and Verify Concurrent Non-Deterministic Event Systems. In: *Foundations of Multi-Paradigm Modelling for Cyber-Physical Systems*. Ed. by P. Carreira, V. Amaral, and H. Vangheluwe. (in press). Springer, 2019.

²<http://mpm4cps.eu>

Modeling Frames I was invited to participate in the Computer Automated Multi-Paradigm Modeling (CAMPaM) workshops in 2016 and 2017. In both years, I joined groups that were researching the practical applicability of Modeling Frames [Zei84]. The work resulted in two publications.

[Den+17] Joachim Denil, **Stefan Klikovits**, Pieter J. Mosterman, Antonio Vallecillo, and Hans Vangheluwe. The Experiment Model and Validity Frame in M&S. In: *Proceedings of the Symposium on Theory of Modeling & Simulation, TMS/DEVS '17*, pp. 109–120. Society for Computer Simulation International, 2017. URL: http://scs.org/wp-content/uploads/2017/06/27_Final_Manuscript.pdf.

[Kli+17] **Stefan Klikovits**, Joachim Denil, Alexandre Muzy, and Rick Salay. Modeling frames. In: *14th Workshop on Model-Driven Engineering, Verification and Validation (MoDeVVA 2017), 19 September 2017*, CEUR Workshop Proceedings, vol. 2019, pp. 315–320. 2017. URL: http://ceur-ws.org/Vol-2019/modevva_3.pdf.

Petri Nets I also collaborated with my colleagues of the Software Modeling and Verification (SMV) group. We published a paper on the use of machine learning to guide the choice of model checking tools based on model properties.

[Buc+18] Didier Buchs, **Stefan Klikovits**, Alban Linard, Romain Mencattini, and Dimitri Racordon. A Model Checker Collection for the Model Checking Contest Using Docker and Machine Learning. In: *Application and Theory of Petri Nets and Concurrency - 39th International Conference, PETRINETs 2018*, Lecture Notes in Computer Science, vol. 10877, pp. 385–395. Springer, 2018. DOI: [10.1007/978-3-319-91268-4_21](https://doi.org/10.1007/978-3-319-91268-4_21).

We also presented our approach to teaching Petri nets to novices (e.g. undergraduate students) using a game called *Petri sport*.

[Kli+18] **Stefan Klikovits**, Alban Linard, Dimitri Racordon, and Didier Buchs. Petri Sport: A Sport for Petri Netters. In: *Petri Nets and Software Engineering. International Workshop, PNSE'18*, CEUR Workshop Proceedings, vol. 2138, pp. 35–56. 2018. URL: <http://ceur-ws.org/Vol-2138/paper2.pdf>.

Bibliography

- [AADL17] *Society of Automotive Engineers: Architecture Analysis and Design Language (SAE AADL) Version 2.2*. SAE Standard: AS5506C. Society of Automotive Engineers. 2017. DOI: [10.4271/AS5506C](https://doi.org/10.4271/AS5506C).
- [ABB97] Jean-Michel Autebert, Jean Berstel, and Luc Boasson. Context-free languages and pushdown automata. In: *Handbook of formal languages*, pp. 111–174. Springer, 1997.
- [AD94] Rajeev Alur and David L. Dill. A Theory of Timed Automata. *Theoretical Computer Science* 126, 1994, pp. 183–235.
- [AD98] Hassane Alla and René David. Continuous and hybrid Petri nets. *Journal of Circuits, Systems, and Computers* 8(01), 1998, pp. 159–188.
- [AH92] Rajeev Alur and Thomas A. Henzinger. Logics and models of real time: A survey. In: *Real-Time: Theory in Practice*, Lecture Notes in Computer Science, vol. 600, pp. 74–106. Springer, 1992.
- [AHH96] R. Alur, T. A. Henzinger, and Pei-Hsin Ho. Automatic symbolic verification of embedded systems. *IEEE Transactions on Software Engineering* 22(3), 1996, pp. 181–201. DOI: [10/ds232r](https://doi.org/10/ds232r).
- [Aks+08] S. Akshay, Benedikt Bollig, Paul Gastin, Madhavan Mukund, and K. Narayan Kumar. Distributed timed automata with independently evolving clocks. In: *Proceedings of the International Conference on Concurrency Theory (CONCUR 2008)*, Lecture Notes in Computer Science, vol. 5201, pp. 82–97. Springer, 2008.
- [Alu+95] Rajeev Alur, Costas Courcoubetis, Nicolas Halbwachs, Thomas A Henzinger, P-H Ho, Xavier Nicollin, Alfredo Olivero, Joseph Sifakis, and Sergio Yovine. The algorithmic analysis of hybrid systems. *Theoretical computer science* 138(1), 1995, pp. 3–34.
- [Alu11] Rajeev Alur. Formal Verification of Hybrid Systems. In: *2011 Proceedings of the Ninth ACM International Conference on Embedded Software (EMSOFT)*, ACM, 2011. DOI: [10.1145/2038642.2038685](https://doi.org/10.1145/2038642.2038685).
- [And+06] Bente Anda, Kai Hansen, Ingolf Gulleßen, and Hanne Kristin Thorsen. Experiences from introducing UML-based development in a large safety-critical project. *Empirical Software Engineering* 11(4), 2006, pp. 555–581. DOI: [10/dkf4vp](https://doi.org/10/dkf4vp).

- [Ash08] Peter J. Ashenden. *The Designer's Guide to VHDL*. 3rd ed. Morgan Kaufmann Publishers Inc., 2008.
- [Aul+13] Denis Aulagnier, Ali Koudri, Stéphane Lecomte, Philippe Soulard, Joël Champeau, Jorgiano Vidal, Gilles Perrouin, and Pierre Leray. SoC/-SoPC Development using MDD and MARTE Profile. In: *Model-Driven Engineering for Distributed Real-Time Systems*, pp. 201–232. Wiley, 2013. DOI: [10.1002/9781118558096.ch8](https://doi.org/10.1002/9781118558096.ch8).
- [BA07] Louis G. Birta and Gilbert Arbez. *Modelling and Simulation: Exploring Dynamic System Behaviour*. Springer, 2007.
- [BAG18] Ankica Barisic, Vasco Amaral, and Miguel Goulão. Usability driven DSL development with USE-ME. *Computer Languages, Systems & Structures* 51, 2018, pp. 118–157. DOI: [10.1016/j.cl.2017.06.005](https://doi.org/10.1016/j.cl.2017.06.005).
- [Bar+05a] J. Barnat, L. Brim, I. Černá, and P. Šimeček. DiVinE – The Distributed Verification Environment. In: *Proceedings of 4th International Workshop on Parallel and Distributed Methods in verification (PDMC)*, pp. 89–94. 2005. URL: http://anna.fi.muni.cz/PDMC/PDMC05/PDMC05_prelim.pdf.
- [Bar+05b] J. Barnat, V. Forejt, M. Leucker, and M. Weber. DivSPIN – A SPIN compatible distributed model checker. In: *Proceedings of 4th International Workshop on Parallel and Distributed Methods in verification (PDMC)*, pp. 95–100. 2005. URL: http://anna.fi.muni.cz/PDMC/PDMC05/PDMC05_prelim.pdf.
- [BBR07] J. Barnat, L. Brim, and P. Ročkait. Scalable Multi-core LTL Model-Checking. In: *Model Checking Software*, Lecture Notes in Computer Science, vol. 4595, pp. 187–203. Springer, 2007. DOI: [10.1007/978-3-540-73370-6_13](https://doi.org/10.1007/978-3-540-73370-6_13).
- [BC06] Tamara Beltrame and François E Cellier. Quantised state system simulation in Dymola/Modelica using the DEVS formalism. In: *Proceedings 5th International Modelica Conference*, pp. 73–82. Modelica, 2006.
- [BCG88] M. C. Browne, E. M. Clarke, and O. Grumberg. Characterizing finite Kripke structures in propositional temporal logic. *Theoretical Computer Science* 59(1), 1988, pp. 115–131. DOI: [10/br8284](https://doi.org/10/br8284).
- [BCM05] Patricia Bouyer, Fabrice Chevalier, and Nicolas Markey. On the Expressiveness of TPTL and MTL. In: *International Conference on Foundations of Software Technology and Theoretical Computer Science*, Lecture Notes in Computer Science, vol. 3821, Springer, 2005. DOI: [10.1007/11590156_35](https://doi.org/10.1007/11590156_35).
- [BCW12] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool Publishers, 2012.
- [Bel04] Alex E. Bell. Death by UML Fever. *Queue* 2(1), 2004, pp. 72–80. DOI: [10/bn8hs5](https://doi.org/10/bn8hs5).

- [Ben+11] Albert Benveniste, Timothy Bourke, Benoît Caillaud, and Marc Pouzet. A hybrid synchronous language with hierarchical automata: static typing and translation to synchronous code. In: *2011 Proceedings of the Ninth ACM International Conference on Embedded Software (EMSOFT)*, p. 137. ACM, 2011. DOI: [10.1145/2038642.2038664](https://doi.org/10.1145/2038642.2038664).
- [Bér+01] Béatrice Bérard, Michel Bidoit, Alain Finkel, François Laroussinie, Antoine Petit, Laure Petrucci, Philippe Schnoebelen, and Pierre Mckenzie. KRONOS – Model Checking of Real-time Systems. In: *Systems and Software Verification: Model-Checking Techniques and Tools*. Springer, 2001, pp. 161–168. DOI: [10.1007/978-3-662-04558-9_16](https://doi.org/10.1007/978-3-662-04558-9_16).
- [BF08] David Broman and Peter Fritzson. Higher-order acausal models. In: *EOOLT*, 2008.
- [BG00] Didier Buchs and Nicolas Guelfi. A formal specification framework for object-oriented distributed systems. *Transactions on Software Engineering* 26(7), 2000, pp. 635–652. DOI: [10.1109/32.859532](https://doi.org/10.1109/32.859532).
- [BG92] Gérard Berry and Georges Gonthier. The Esterel synchronous programming language: design, semantics, implementation. *Science of Computer Programming* 19(2), 1992, pp. 87–152. DOI: [10.1016/0167-6423\(92\)90005-V](https://doi.org/10.1016/0167-6423(92)90005-V).
- [BGJ91] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming* 16(2), 1991, pp. 103–149. DOI: [10/d5gqwb](https://doi.org/10/d5gqwb).
- [BH09] Beatrice Bérard and Serge Haddad. Interrupt timed automata. In: *International Conference on Foundations of Software Science and Computational Structures, Lecture Notes in Computer Science*, vol. 5504, pp. 197–211. Springer, 2009.
- [Bie+99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic Model Checking without BDDs. In: *TACAS '99: Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, Lecture Notes in Computer Science, vol. 1579, pp. 193–207. Springer, 1999.
- [BK11] Federico Bergero and Ernesto Kofman. PowerDEVS: a tool for hybrid system modeling and real-time simulation. *SIMULATION* 87(1-2), 2011, pp. 113–132. DOI: [10/bwzxtb](https://doi.org/10/bwzxtb).
- [BKS03] G. Berry, M. Kishinevsky, and S. Singh. System level design and verification using a synchronous language. In: *ICCAD-2003. International Conference on Computer Aided Design*, pp. 433–439. 2003. DOI: [10.1109/ICCAD.2003.159720](https://doi.org/10.1109/ICCAD.2003.159720).
- [BL08] Patricia Bouyer and François Laroussinie. Model Checking Timed Automata. In: *Modeling and Verification of Real-Time Systems*, pp. 111–140. Wiley, 2008.

- [Bla+10] David C. Black, Jack Donovan, Bill Bunton, and Anna Keist. *SystemC: From the Ground Up*. Springer, 2010.
- [Blo+12] Torsten Blochwitz, Martin Otter, Johan Akesson, Martin Arnold, Christoph Clauss, Hilding Elmqvist, Markus Friedrich, Andreas Junghanns, Jakob Mauss, Dietmar Neumerkel, et al. Functional mockup interface 2.0: the standard for tool independent exchange of simulation models. In: *Proceedings of the 9th International MODELICA Conference*, pp. 173–184. 2012.
- [Bou09] Patricia Bouyer. Model-checking Timed Temporal Logics. *Electronic Notes in Theoretical Computer Science* 231, 2009, pp. 323–341. DOI: [10/cz4brn](https://doi.org/10/cz4brn).
- [Boz+00] Marius Bozga, Susanne Graf, Laurent Mounier, Alain Kerbrat, Iulian Ober, and Daniel Vincent. SDL for Real-Time: What Is Missing? In: *2nd Workshop of the SDL Forum Society on SDL and MSC SAM2000*, pp. 108–122. IMAG, 2000. URL: <https://hal.archives-ouvertes.fr/hal-00374117>.
- [BP13] Timothy Bourke and Marc Pouzet. Zélus: A Synchronous Language with ODEs. In: *16th International Conference on Hybrid Systems: Computation and Control*, pp. 113–118. 2013.
- [Bro+05] Christopher Brooks, Adam Cataldo, Edward A Lee, Jie Liu, Xiaojun Liu, Steve Neuendorffer, and Haiyang Zheng. HyVisual: A hybrid system visual modeler. *University of California, Berkeley, Technical Memorandum UCB/ERL M 5*, 2005.
- [Bro+12] David Broman, Edward A. Lee, Stavros Tripakis, and Martin Törngren. Viewpoints, formalisms, languages, and tools for cyber-physical systems. In: *Proceedings of the 6th International Workshop on Multi-Paradigm Modeling*, MPM '12, pp. 49–54. ACM, 2012. DOI: [10.1145/2508443.2508452](https://doi.org/10.1145/2508443.2508452).
- [Bro+92] Manfred Broy, Frank Dederichs, Claus Dendorfer, Max Fuchs, Thomas F. Gritzner, and Rainer Weber. *The design of distributed systems: an introduction to focus*. Tech. rep. Technical University of Munich, Munich, Germany, 1992. URL: <http://www4.in.tum.de/publ/papers/TUM-I9202.pdf>.
- [Bro99] Jan F. Broenink. Introduction to Physical Systems Modelling with Bond Graphs. In: *SiE Whitebook on Simulation Methodologies*, 1999.
- [Brü+02] Dag Brück, Hilding Elmqvist, Sven Erik Mattsson, and Hans Olsson. Dymola for multi-engineering modeling and simulation. In: *Proceedings of Modelica*, 2002.

- [Bru+11] A. J. Bernheim Brush, Bongshin Lee, Ratul Mahajan, Sharad Agarwal, Stefan Saroiu, and Colin Dixon. Home Automation in the Wild: Challenges and Opportunities. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 2115–2124. ACM, 2011. DOI: [10.1145/1978942.1979249](https://doi.org/10.1145/1978942.1979249).
- [Bry86] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. In: vol. C-35, pp. 677–691. IEEE, 1986. DOI: [10.1109/TC.1986.1676819](https://doi.org/10.1109/TC.1986.1676819).
- [Bur+92] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and Lucius J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation* 98(2), 1992, pp. 142–170. DOI: [10.1016/0890-5401\(92\)90017-A](https://doi.org/10.1016/0890-5401(92)90017-A).
- [Bur+94] Jerry R. Burch, Edmund M. Clarke, David E. Long, Kenneth L. McMillan, and David L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 13, 1994, pp. 401–424.
- [But08] J. C. Butcher. *Numerical Methods for Ordinary Differential Equations*. 2nd ed. Wiley, 2008.
- [BV01] Jean-Sebastien Bolduc and Hans Vangheluwe. *The modelling and simulation package PythonDEVS for Classical hierarchical DEVS*. Tech. rep. McGill University, 2001. URL: <https://repository.uantwerpen.be/link/irua/108622>.
- [BY03] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In: *Advanced Course on Petri Nets*, Lecture Notes in Computer Science, vol. 3098, pp. 87–124. Springer, 2003.
- [Car+06] Luca P. Carloni, Roberto Passerone, Alessandro Pinto, and Alberto L. Sangiovanni-Vincentelli. Languages and Tools for Hybrid Systems Design. *Foundations and Trends in Electronic Design Automation* 1(1/2), 2006, pp. 1–193. DOI: [10/cxjxnq](https://doi.org/10/cxjxnq).
- [CBM90] Olivier Coudert, Christian Berthet, and Jean Christophe Madre. Formal Boolean Manipulations for the Verification of Sequential Machines. In: *EURO-DAC '90: Proceedings of the conference on European design automation*, pp. 57–61. IEEE Computer Society Press, 1990.
- [CE81] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In: *Workshop on Logic of Programs*, Lecture Notes in Computer Science, vol. 5000, pp. 52–71. 1981.
- [CGP99] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, 1999.
- [Che76] Peter Pin-Shan Chen. The Entity-Relationship Model: Toward a Unified View of Data. *ACM Transactions on Database Systems* 1, 1976, pp. 9–36.

- [CKW10] Rodrigo Castro, Ernesto Kofman, and Gabriel Wainer. A Formal Framework for Stochastic Discrete Event System Specification Modeling and Simulation. *SIMULATION* 86(10), 2010, pp. 587–611. DOI: [10/c6b3zc](https://doi.org/10/c6b3zc).
- [CL00] Franck Cassez and Kim Larsen. The impressive power of stopwatches. In: *International Conference on Concurrency Theory*, Lecture Notes in Computer Science, vol. 1877, pp. 138–152. Springer, 2000.
- [Cle96] P. C. Clements. A survey of architecture description languages. In: *Proceedings of the 8th International Workshop on Software Specification and Design*, pp. 16–25. 1996. DOI: [10.1109/IWSSD.1996.501143](https://doi.org/10.1109/IWSSD.1996.501143).
- [CMT96] Bernadette Charron-Bost, Friedemann Mattern, and Gerard Tel. Synchronous, asynchronous, and causally ordered communication. *Distributed Computing* 9(4), 1996. DOI: [10.1007/s004460050018](https://doi.org/10.1007/s004460050018).
- [Cri96] Flaviu Cristian. Synchronous and asynchronous. *Communications of the ACM* 39(4), 1996, pp. 88–97. DOI: [10.1145/227210.227231](https://doi.org/10.1145/227210.227231).
- [CW96] Edmund M. Clarke and Jeannette M. Wing. Formal methods: state of the art and future directions. *ACM Computing Surveys* 28(4), 1996, pp. 626–643. DOI: [10.1145/242223.242257](https://doi.org/10.1145/242223.242257).
- [CZ94] Alex Chung Hen Chow and Bernard P. Zeigler. Parallel DEVS: A parallel, hierarchical, modular modeling formalism. In: *Simulation Conference Proceedings, 1994. Winter*, pp. 716–722. IEEE, 1994.
- [DA01] René David and Hassane Alla. On hybrid Petri nets. *Discrete Event Dynamic Systems* 11(1-2), 2001, pp. 9–40. DOI: [10/dp9ks6](https://doi.org/10/dp9ks6).
- [DA10] René David and Hassane Alla. *Discrete, Continuous, and Hybrid Petri Nets*. 2nd. Springer, 2010.
- [DB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In: *14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer, 2008. URL: <http://dl.acm.org/citation.cfm?id=1792734.1792766>.
- [Dem+08] S. Demathieu, F. Thomas, C. André, S. Gérard, and F. Terrier. First Experiments Using the UML Profile for MARTE. In: *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, 2008. DOI: [10.1109/ISORC.2008.36](https://doi.org/10.1109/ISORC.2008.36).
- [DG05] Hernán P. Dacharry and Norbert Giambiasi. Formal verification with timed automata and devs models: a case study. In: *Argentine Symposium on Software Engineering*, pp. 251–265. 2005.
- [Did+07] DelaNote Didier, Stefan Van Baelen, Wouter Joosen, and Yolande Berbers. Using AADL in model driven development. In: *IEEE-SEE international workshop on UML and AADL 2007*, pp. 1–10. 2007.

- [DK08] Arie Van Deursen and Paul Klint. Little Languages: Little Maintenance? *Journal of Software Maintenance: Research and Practice* 10(2), 2008, pp. 75–92.
- [DKW08] V. D’Silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27(7), 2008, pp. 1165–1178. DOI: [10.1109/TCAD.2008.923410](https://doi.org/10.1109/TCAD.2008.923410).
- [DLS12] P. Derler, E. A. Lee, and A. L. Sangiovanni-Vincentelli. Modeling Cyber-Physical Systems. *Proceedings of the IEEE* 100(1), 2012, pp. 13–28. DOI: [10.1109/JPROC.2011.2160929](https://doi.org/10.1109/JPROC.2011.2160929).
- [Dol03] Laurent Doldi. *Validation of Communications Systems with SDL: The Art of SDL Simulation and Reachability Analysis*. Wiley, 2003.
- [DP12] Christina Deatcu and Thorsten Pawletta. A Qualitative Comparison of Two Hybrid DEVS Approaches. *SNE Simulation Notes Europe* 22(1), 2012, pp. 15–24. DOI: [10/gdm6rm](https://doi.org/10/gdm6rm).
- [EAST13] *EAST-ADL: Domain Model Specification Version 2.1.12*. EAST-ADL Association. 2013. URL: http://www.east-adl.info/Specification/V2.1.12/EAST-ADL-Specification_V2.1.12.pdf.
- [ECT03] Jad El-khoury, De-Jiu Chen, and Martin Törngren. *A Survey of Modeling Approaches for Embedded Computer Control Systems*. Tech. rep. KTH Royal Institute of Technology, Stockholm, Sweden, 2003. URL: <http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-13042>.
- [EKP01] Joel M. Esposito, Vijay Kumar, and George J. Pappas. Accurate Event Detection for Simulating Hybrid Systems. In: *Hybrid Systems: Computation and Control*, Lecture Notes in Computer Science, vol. 2034, pp. 204–217. Springer, 2001.
- [EM85] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 1: Equations und Initial Semantics*. Vol. 6. EATCS Monographs on Theoretical Computer Science. Springer, 1985. DOI: [10.1007/978-3-642-69962-7](https://doi.org/10.1007/978-3-642-69962-7).
- [ERG08] Huascar Espinoza, Kai Richter, and Sébastien Gérard. Evaluating MARTE in an Industry-Driven Environment: TIMMO’s Challenges for AUTOSAR Timing Modeling. In: *Proceedings of Design Automation and Test in Europe (DATE)*, p. 6. 2008.
- [Esc+01] Robert Eschbach, Uwe Glässer, Reinhard Gotzhein, Martin Löwis, and Andreas Prinz. Formal definition of SDL-2000 - Compiling and running SDL specifications as ASM models. *Journal of Universal Computer Science* 7(11), 2001, pp. 1024–1049.
- [Est08] Jeff A. Estefan. Survey of Model-Based Systems Engineering (MBSE) Methodologies. *International Council on Systems Engineering (INCOSE) MBSE Initiative*, 2008.

- [Fau+07] Madeleine Faugere, Thimothée Bourbeau, Robert de Simone, and Sébastien Gérard. MARTE: Also an UML Profile for Modeling AADL Applications. In: *12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007)*, pp. 359–364. IEEE, 2007. DOI: [10.1109/ICECCS.2007.29](https://doi.org/10.1109/ICECCS.2007.29).
- [FE98] Peter Fritzson and Vadim Engelson. Modelica — a unified object-oriented language for system modeling and simulation. In: *European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, vol. 1445, pp. 67–90. Springer, 1998.
- [FGH06] Peter H. Feiler, David P. Gluch, and John J. Hudak. *The Architecture Analysis & Design Language (AADL): An Introduction*. Tech. rep. Defense Technical Information Center, 2006. DOI: [10.21236/ADA455842](https://doi.org/10.21236/ADA455842).
- [Fis+00] Joachim Fischer, Eckhardt Holz, Martin Löwis, and Andreas Prinz. SDL-2000: A Language with a Formal Semantics. *Rigorous Object-Oriented Methods 2000*, 2000, p. 15.
- [Fra+07a] R. B. Franca, J. P. Bodeveix, M. Filali, J. F. Rolland, D. Chemouil, and D. Thomas. The AADL behaviour annex – experiments and roadmap. In: *12th IEEE International Conference on Engineering Complex Computer Systems*, pp. 377–382. 2007. DOI: [10.1109/ICECCS.2007.41](https://doi.org/10.1109/ICECCS.2007.41).
- [Fra+07b] Ricardo Bedin França, Jean-François Rolland, Mamoun Filali Amine, Jean-Paul Bodeveix, and David Chemouil. Assessment of the AADL Behavioral Annex. *Journées FAC*, 2007, p. 13.
- [Fri+06] Peter Fritzson, Peter Aronsson, Adrian Pop, Hakan Lundvall, Kaj Nyström, Levon Saldamli, David Broman, and Anders Sandholm. Open-Modelica – A free open-source environment for system modeling, simulation, and teaching. In: *Computer Aided Control System Design, 2006 IEEE International Conference on Control Applications, 2006 IEEE International Symposium on Intelligent Control, 2006 IEEE*, pp. 1588–1595. 2006.
- [Fri09] Sanford Friedenthal. SysML: Lessons from Early Applications and Future Directions. *INSIGHT* 12(4), 2009, pp. 10–12. DOI: [10/gdwpmmm](https://doi.org/10/gdwpmmm).
- [GAS05] Latefa Ghomri, Hassane Alla, and Zaki Sari. Structural and Hierarchical Translation of Hybrid Petri Nets in Hybrid Automata. *Proceedings of IMACS'05*, 2005.
- [GG15] David Goldsman and Paul Goldsman. Discrete-Event Simulation. In: *Modeling and Simulation in the Systems Engineering Life Cycle*, Simulation Foundations, Methods and Applications, pp. 103–109. Springer, 2015. DOI: [10.1007/978-1-4471-5634-5_10](https://doi.org/10.1007/978-1-4471-5634-5_10).
- [Gil62] A. Gill. *Introduction to the Theory of Finite-State Machines*. McGraw-Hill, 1962.

- [GMS01] Hubert Garavel, Radu Mateescu, and Irina Smarandache. Parallel State Space Construction for Model-checking. In: *8th International SPIN Workshop on Model Checking of Software*, Lecture Notes in Computer Science, pp. 217–234. Springer, 2001.
- [God91] Patrice Godefroid. Using partial orders to improve automatic verification methods. In: *Computer-Aided Verification*, Lecture Notes in Computer Science, vol. 531, pp. 176–185. Springer, 1991.
- [Gom+17] Cláudio Gomes, Casper Thule, David Broman, Peter Gorm Larsen, and Hans Vangheluwe. Co-simulation: state of the art. *CoRR* abs/1702.00686, 2017.
- [GU96] A. Giua and E. Usai. High-level hybrid Petri nets: a definition. In: *Proceedings of 35th IEEE Conference on Decision and Control*, vol. 1, 148–150 vol.1. IEEE, 1996. DOI: [10.1109/CDC.1996.574277](https://doi.org/10.1109/CDC.1996.574277).
- [Hal+91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. In: *Proceedings of the IEEE*, pp. 1305–1320. 1991.
- [Hal93] Nicolas Halbwachs. *Synchronous Programming of Reactive Systems*. The Springer International Series in Engineering and Computer Science. Springer, 1993.
- [Hal98] Nicolas Halbwachs. Synchronous Programming of Reactive Systems - A Tutorial and Commented Bibliography. In: *In Tenth International Conference on Computer-Aided Verification*, Lecture Notes in Computer Science, vol. 1427, pp. 1–16. Springer, 1998.
- [Har87] David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming* 8(3), 1987, pp. 231–274. DOI: [10/b97n8k](https://doi.org/10/b97n8k).
- [Hei+] Matthias Heizmann, Aina Niemetz, Giles Reger, and Tjark Weber. SMT-COMP 2019. <http://www.smtcomp.org/>. Accessed: 2019-03-02.
- [Hei98] Constance L. Heitmeyer. On the need for practical formal methods. In: *Proceedings of the 5th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, Lecture Notes in Computer Science, vol. 1486, pp. 18–26. Springer, 1998. URL: <http://dl.acm.org/citation.cfm?id=646845.706947>.
- [Hen+94] Thomas A Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Information and computation* 111(2), 1994, pp. 193–244.
- [Hen+98] Thomas A. Henzinger, Peter W. Kopke, Anuj Puri, and Pravin Varaiya. What’s decidable about hybrid automata? *Journal of Computer and System Sciences* 57(1), 1998, pp. 94–124.
- [Her+17] Mostafa Herajy, Fei Liu, Christian Rohr, and Monika Heiner. (*Coloured*) *Hybrid Petri nets in Snoopy - user manual*. Tech. rep. 2017,01. Institut für Informatik, 2017. URL: <https://opus4.kobv.de/opus4-btu/frontdoor/index/index/docId/4157>.

- [HG05] Alejandro Hernandez and Norbert Giambiasi. State Reachability for DEVS Models. In: *Argentine Symposium on Software Engineering*, 2005.
- [HG15] Paula Herber and Sabine Glesner. Verification of embedded real-time systems. In: *Formal Modeling and Verification of Cyber-Physical Systems, 1st International Summer School on Methods and Tools for the Design of Digital Systems*, pp. 1–25. 2015. DOI: [10.1007/978-3-658-09994-7_1](https://doi.org/10.1007/978-3-658-09994-7_1).
- [Hil99] Rich Hilliard. Aspects, Concerns, Subjects, Views, ... In: *OOPSLA'99 Workshop on MultiDimensional Separation of Concerns in Object-Oriented Systems*, 1999.
- [Hon+97] Joon Sung Hong, Hae-Sang Song, Tag Gon Kim, and Kyu Ho Park. A Real-Time Discrete Event System Specification Formalism for Seamless Real-Time Software Development. *Discrete Event Dynamic Systems* 7(4), 1997, pp. 355–375. DOI: [10/dtwr66](https://doi.org/10/dtwr66).
- [HP85] D. Harel and A. Pnueli. On the Development of Reactive Systems. In: *Logics and Models of Concurrent Systems*, NATO ASI Series, vol. 13, pp. 477–498. Springer, 1985.
- [HR04] Gregoire Hamon and John Rushby. An Operational Semantics for State-flow. *Fundamental Approaches to Software Engineering*, 2004.
- [HRR14] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. *CoRR* abs/1409.6578, 2014.
- [HW15] Amy E. Henninger and Elizabeth T. Whitaker. Modeling Behavior. In: *Modeling and Simulation in the Systems Engineering Life Cycle*, Simulation Foundations, Methods and Applications, pp. 75–87. Springer, 2015. DOI: [10.1007/978-1-4471-5634-5_8](https://doi.org/10.1007/978-1-4471-5634-5_8).
- [Hwa11] Moon Ho Hwang. Taxonomy of devs subclasses for standardization. In: *Proceedings of the 2011 Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*, pp. 152–159. 2011.
- [HWR14] John Hutchinson, Jon Whittle, and Mark Rouncefield. Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure. *Science of Computer Programming* 89, 2014, pp. 144–161. DOI: [10/f57vwn](https://doi.org/10/f57vwn).
- [HZ09] M. H. Hwang and B. P. Zeigler. Reachability Graph of Finite and Deterministic DEVS Networks. *IEEE Transactions on Automation Science and Engineering* 6(3), 2009, pp. 468–478. DOI: [10/bgfrvk](https://doi.org/10/bgfrvk).
- [IDEF18] *Integrated DEfinition Methods (IDEF)*. Accessed: 2018-07-29. Knowledge Based Systems, Inc. (KBSI). 2018.
- [IEEE99] *IEEE Standard VHDL Analog and Mixed-Signal Extensions*. IEEE Std 1076.1-1999. IEEE, 1999.

- [Inf06] InfoQ. Ivar Jacobson on UML, MDA, and the future of methodologies. https://www.infoq.com/interviews/Ivar_Jacobson. Accessed: 2018-09-28. 2006.
- [Iqb+12] Muhammad Zohaib Iqbal, Shaukat Ali, Tao Yue, and Lionel Briand. Experiences of Applying UML/MARTE on Three Industrial Projects. In: *Model Driven Engineering Languages and Systems*, Lecture Notes in Computer Science, vol. 7590, pp. 642–658. Springer, 2012. DOI: [10.1007/978-3-642-33666-9_41](https://doi.org/10.1007/978-3-642-33666-9_41).
- [Jen96] Kurt Jensen. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use - Volume 1, Second Edition*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 1996. DOI: [10.1007/978-3-662-03241-1](https://doi.org/10.1007/978-3-662-03241-1).
- [Jie+99] Jie Liu, Xiaojun Liu, Tak-Kuen J. Koo, B. Sinopoli, S. Sastry, and E.A. Lee. A hierarchical hybrid system model and its simulation. In: *Proceedings of the 38th IEEE Conference on Decision and Control*, vol. 4, pp. 3508–3513. IEEE, 1999. DOI: [10/cpnz7f](https://doi.org/10/cpnz7f).
- [JLL77] Neil D. Jones, Lawrence H. Landweber, and Y. Edmund Lien. Complexity of some problems in Petri nets. *Theoretical Computer Science* 4(3), 1977, pp. 277–299. DOI: [10/d7nhrf](https://doi.org/10/d7nhrf).
- [JR12] Kurt Jensen and Grzegorz Rozenberg. *High-level Petri nets: theory and application*. Springer, 2012.
- [Kah74] Gilles Kahn. The semantics of simple language for parallel programming. In: *IFIP Congress*, pp. 471–475. 1974.
- [Ken12] Ken Hanly. In 1936 Soviet scientist Lukyanov built an analog water computer. <http://www.digitaljournal.com/article/338106>. Accessed: 2018-09-09. 2012.
- [KJ01] Ernesto Kofman and Sergio Junco. Quantized-state Systems: A DEVS Approach for Continuous System Simulation. *Transactions of the Society for Computer Simulation International* 18(3), 2001, pp. 123–132.
- [KP89] Shmuel Katz and Doron Peled. An efficient verification method for parallel and distributed programs. In: *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, School/Workshop*, Lecture Notes in Computer Science, vol. 354, pp. 489–507. Springer, 1989.
- [LÁÖ15] Daniela Lepri, Erika Ábrahám, and Peter Csaba Ölveczky. Sound and complete timed CTL model checking of timed Kripke structures and real-time rewrite theories. *Science of Computer Programming*, 2015, pp. 128–192. DOI: [10/f6zwpd](https://doi.org/10/f6zwpd).
- [Lee08] Edward A. Lee. Cyber physical systems: Design challenges. In: *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pp. 363–369. IEEE, 2008.

- [Li+10] Juncao Li, Nicholas T. Pilkington, Fei Xie, and Qiang Liu. Embedded architecture description language. *Journal of Systems and Software* 83(2), 2010, pp. 235–252. DOI: [10/ds7fsc](https://doi.org/10.1016/j.jss.2010.05.001).
- [Lop15] Margaret L. Loper. Modeling Time. In: *Modeling and Simulation in the Systems Engineering Life Cycle*, Simulation Foundations, Methods and Applications, pp. 89–101. Springer, 2015. DOI: [10.1007/978-1-4471-5634-5_9](https://doi.org/10.1007/978-1-4471-5634-5_9).
- [LPY97] Kim G Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer* 1(1-2), 1997, pp. 134–152.
- [LZ07] Edward A. Lee and Haiyang Zheng. Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. In: *Proceedings of the 7th ACM & IEEE International Conference on Embedded Software*, EMSOFT '07, pp. 114–123. ACM, 2007. DOI: [10.1145/1289927.1289949](https://doi.org/10.1145/1289927.1289949).
- [MARTE11] *UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems (OMG MARTE) Version 1.1*. OMG Document Number: formal/11-06-02. Object Management Group. 2011. URL: <https://www.omg.org/spec/MARTE/1.1/PDF>.
- [Mat08] Norman Matloff. *Introduction to Discrete-Event Simulation and the Simpy Language*. Tech. rep. University of California, Davis, Department of Computer Science, 2008. URL: <http://heather.cs.ucdavis.edu/~matloff/156/PLN/DESImIntro.pdf>.
- [MDA14] *Object Management Group: Model Driven Architecture Guide (MDA Guide) Version 2.0*. OMG Document Number: formal/2014-06-01. Object Management Group. 2014. URL: <http://www.omg.org/cgi-bin/doc?ormsc/14-06-01>.
- [Mer01] Stephan Merz. Model checking: a tutorial overview. In: *4th Summer School on Modeling and Verification of Parallel Processes*. Ed. by Franck Cassez, Claude Jard, Brigitte Rozoy, and Mark Dermot Ryan. Vol. 2067. Lecture Notes in Computer Science. Springer, 2001, pp. 3–38. DOI: [10.1007/3-540-45510-8_1](https://doi.org/10.1007/3-540-45510-8_1).
- [MIJ14] Marius Minea, Cornel Izbasca, and Calin Jebelean. Experience with Formal Verification of SDL Protocols. *International Journal of Computing* 2(3), 2014, pp. 63–68.
- [Moo09] Daniel Moody. The “physics” of notations: toward a scientific basis for constructing visual notations in software engineering. *IEEE Transactions on Software Engineering* 35(6), 2009, pp. 756–779. DOI: [10.1109/TSE.2009.67](https://doi.org/10.1109/TSE.2009.67).
- [Mos07] Pieter Mosterman. Hybrid Dynamic Systems: Modeling and Execution. In: *Handbook of Dynamic System Modeling*, vol. 20073719, Chapman and Hall/CRC, 2007.

- [MSS99] Markus Müller-Olm, David Schmidt, and Bernhard Steffen. Model-Checking. In: *Static Analysis*, pp. 330–354. Springer, 1999.
- [Mue+01] W. Mueller, J. Ruf, D. Hoffmann, J. Gerlach, T. Kropf, and W. Rosenstiehl. The simulation semantics of SystemC. In: *Design, Automation and Test in Europe. Conference and Exhibition 2001*, pp. 64–70. 2001. DOI: [10.1109/DATE.2001.915002](https://doi.org/10.1109/DATE.2001.915002).
- [Nan94] Richard E. Nance. The conical methodology and the evolution of simulation model development. *Annals of Operations Research* 53(1), 1994, pp. 1–45. DOI: [10.1007/BF02136825](https://doi.org/10.1007/BF02136825).
- [Obj18] Object Management Group. The Official OMG MARTE Web Site. Accessed: 2018-07-29. 2018. URL: <https://www.omg.org/omgmarte/>.
- [Oqu04] Flavio Oquendo. π -ADL: an Architecture Description Language based on the higher-order typed π -calculus for specifying dynamic and mobile software architectures. *ACM SIGSOFT Software Engineering Notes* 29(3), 2004. DOI: [10/fb99bd](https://doi.org/10/fb99bd).
- [Osa+14] Yuki Osada, Tim French, Mark Reynolds, and Harry Smallbone. Hourglass Automata. *Electronic Proceedings in Theoretical Computer Science* 161, 2014, pp. 175–188. DOI: [10.4204/EPTCS.161.16](https://doi.org/10.4204/EPTCS.161.16).
- [Pag95] Ernest H. Page Jr. Simulation Modeling Methodology: Principles and Etiology of Decision Support. AAI9638633. PhD thesis. Virginia Polytechnic Institute and State University, 1995.
- [Pan10] R. K. Pandey. Architectural description languages (ADLs) vs UML: a review. *ACM SIGSOFT Software Engineering Notes* 35(3), 2010. DOI: [10/ccr867](https://doi.org/10/ccr867).
- [Pay61] Henry Martyn Paynter. *Analysis and Design of Engineering Systems*. MIT Press, Cambridge, MA, 1961, p. 347.
- [PB96] Taeshin Park and Paul I Barton. State event location in differential-algebraic models. *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 6(2), 1996, pp. 137–165.
- [Pel09] Radek Pelánek. Fighting State Space Explosion: Review and Evaluation. In: *Formal Methods for Industrial Critical Systems*, Lecture Notes in Computer Science, vol. 5596, pp. 37–52. Springer, 2009.
- [Per+12] Maxime Perrotin, Eric Conquet, Julien Delange, André Schiele, and Thanassis Tsiodras. TASTE: A Real-Time Software Engineering Tool-Chain Overview, Status, and Future. In: *SDL 2011: Integrating System and Software Modeling*, Lecture Notes in Computer Science, vol. 7083, pp. 26–37. Springer, 2012.
- [Pet62] C. A. Petri. Kommunikation mit Automaten. German. PhD thesis. Rheinisch-Westfälisches Institut für Instrumentelle Mathematik an der Universität Bonn, 1962.

- [Pnu77] A. Pnueli. The temporal logic of programs. In: *18th Annual Symposium on Foundations of Computer Science*, pp. 46–57. 1977. DOI: [10.1109/SFCS.1977.32](https://doi.org/10.1109/SFCS.1977.32).
- [Pop13] Louchka Popova-Zeugmann. *Time and Petri Nets*. Springer, 2013.
- [PV94] Anuj Puri and Pravin Varaiya. Decidability of hybrid systems with rectangular differential inclusions. In: *Computer Aided Verification*, Lecture Notes in Computer Science, vol. 818, pp. 95–104. Springer, 1994. DOI: [10.1007/3-540-58179-0_46](https://doi.org/10.1007/3-540-58179-0_46).
- [Raj+18] Akshay Rajhans, Srinath Avadhanula, Alongkrit Chutinan, Pieter J. Mosterman, and Fu Zhang. Graphical modeling of hybrid dynamics with simulink and stateflow. In: *Proceedings of the 21st International Conference on Hybrid Systems: Computation and Control (Part of CPS Week)*, HSCC '18, pp. 247–252. ACM, 2018. DOI: [10.1145/3178126.3178152](https://doi.org/10.1145/3178126.3178152).
- [Ras05] Jean-François Raskin. An introduction to hybrid automata. In: *Handbook of Networked and Embedded Control Systems*, Control Engineering, pp. 491–517. Springer, 2005.
- [Ray10] Pascal Raymond. Synchronous Program Verification with Lustre/Lesar. In: *Modeling and Verification of Real-Time Systems*, pp. 171–206. Wiley, 2010. DOI: [10.1002/9780470611012.ch6](https://doi.org/10.1002/9780470611012.ch6).
- [Reg15] Andy Register. Continuous Time Simulation. In: *Modeling and Simulation in the Systems Engineering Life Cycle*, Simulation Foundations, Methods and Applications, pp. 111–137. Springer, 2015. DOI: [10.1007/978-1-4471-5634-5_11](https://doi.org/10.1007/978-1-4471-5634-5_11).
- [Rev+00] L. Reveillere, F. Merillon, C. Consel, R. Marlet, and G. Muller. A DSL approach to improve productivity and safety in device drivers development. In: *Proceedings ASE 2000. Fifteenth IEEE International Conference on Automated Software Engineering*, pp. 101–109. IEEE, 2000. DOI: [10.1109/ASE.2000.873655](https://doi.org/10.1109/ASE.2000.873655).
- [RHS07] Laura Recalde, Serge Haddad, and Manuel Silva. Continuous Petri Nets: Expressive Power and Decidability Issues. In: *Automated Technology for Verification and Analysis*, Lecture Notes in Computer Science, vol. 4762, pp. 362–377. Springer, 2007.
- [Rod15] Alberto Rodrigues da Silva. Model-driven engineering: A survey supported by the unified conceptual model. *Computer Languages, Systems & Structures* 43, 2015, pp. 139–155. DOI: [10/gdtg4p](https://doi.org/10/gdtg4p).
- [RRW13] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. From Software Architecture Structure and Behavior Modeling to Implementations of Cyber-Physical Systems. In: *Software Engineering*, 2013.

- [RRW14] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. A Case Study on Model-Based Development of Robotic Systems using MontiArc with Embedded Automata. *CoRR* abs/1408.5692, 2014. arXiv: [1408.5692](https://arxiv.org/abs/1408.5692).
- [RRW15] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton. *CoRR* abs/1509.04505, 2015. arXiv: [1509.04505](https://arxiv.org/abs/1509.04505).
- [Sal03] A. Salem. Formal Semantics of Synchronous SystemC. In: *Design, Automation and Test in Europe. Conference and Exhibition 2003*, pp. 376–381. 2003. DOI: [10.1109/DATE.2003.1253637](https://doi.org/10.1109/DATE.2003.1253637).
- [Sar15] Robert G. Sargent. Types of Models. In: *Modeling and Simulation in the Systems Engineering Life Cycle*, Simulation Foundations, Methods and Applications, pp. 51–55. Springer, 2015. DOI: [10.1007/978-1-4471-5634-5_5](https://doi.org/10.1007/978-1-4471-5634-5_5).
- [SB10] John A Sokolowski and Catherine M Banks. *Modeling and Simulation Fundamentals: Theoretical Underpinnings and Practical Domains*. Wiley, 2010.
- [Sch06] Douglas C. Schmidt. Model-driven engineering. *IEEE Computer* 39(2), 2006. DOI: [10.1109/MC.2006.58](https://doi.org/10.1109/MC.2006.58).
- [SD97] Ulrich Stern and David L. Dill. Parallelizing the Mur ϕ verifier. In: *Computer Aided Verification. 9th International Conference*, Lecture Notes in Computer Science, vol. 1254, pp. 256–267. Springer, 1997.
- [SDL16] *Z.100: Specification and Description Language - Overview of SDL-2010*. ITU Recommendation Z.100 (04/16); Article Number: E 40655. International Telecommunication Union. 2016. URL: <http://handle.itu.int/11.1002/1000/12846>.
- [Sel03] B. Selic. The pragmatics of model-driven development. *IEEE Software* 20(5), 2003, pp. 19–25. DOI: [10/dvzhdq](https://doi.org/10/dvzhdq).
- [SPTP05] *UML Profile for Schedulability, Performance, and Time Specification (OMG SPTP) Version 1.1*. OMG Document Number: formal/05-01-02. Object Management Group. 2005. URL: <https://www.omg.org/spec/SPTP/1.1/PDF>.
- [SS01] Natalia Sidorova and Martin Steffen. Verifying Large SDL-Specifications Using Model Checking. In: *SDL 2001: Meeting UML*, Lecture Notes in Computer Science, vol. 2078, pp. 403–420. Springer, 2001.
- [SSB14] M. R. Sena Marques, E. Siegart, and L. Brisolaro. Integrating UML, MARTE and SysML to improve requirements specification and traceability in the embedded domain. In: *2014 12th IEEE International Conference on Industrial Informatics (INDIN)*, pp. 176–181. IEEE, 2014. DOI: [10.1109/INDIN.2014.6945504](https://doi.org/10.1109/INDIN.2014.6945504).

- [Sta06] Mirosław Staron. Adopting Model Driven Software Development in Industry – A Case Study at Two Companies. In: *Model Driven Engineering Languages and Systems*, Lecture Notes in Computer Science, vol. 4199, pp. 57–72. Springer, 2006. DOI: [10.1007/11880240_5](https://doi.org/10.1007/11880240_5).
- [SW09] Hesham Saadawi and Gabriel Wainer. Verification of real-time DEVS models. In: *Proceedings of the 2009 Spring Simulation Multiconference*, p. 8. 2009.
- [SW12] Hesham Saadawi and Gabriel Wainer. On the Verification of Hybrid DEVS Models. In: *Proceedings of the 2012 Symposium on Theory of Modeling and Simulation - DEVS Integrative M&S Symposium*, 26:1–26:8. 2012.
- [SysC12] *IEEE Standard for Standard SystemC Language Reference Manual (IEEE Std 1666-2011)*. IEEE Computer Society. 2012. DOI: [10/fxxf5p](https://doi.org/10/fxxf5p).
- [SysC16] *IEEE Standard for Standard SystemC Analog/Mixed-Signal Extensions Language (SystemC AMS) Reference Manual (IEEE Std 1666.1-2016)*. IEEE Computer Society. 2016.
- [SysML17] *OMG Systems Modeling Language (OMG SysML) Version 1.5*. OMG Document Number: formal-2017-05-01. Object Management Group. 2017. URL: <https://www.omg.org/spec/SysML/1.5/PDF>.
- [Tea18] Team SimPy. SimPy – Discrete event simulation for Python. Accessed: 2018-09-09. 2018. URL: <https://simpy.readthedocs.io>.
- [TM96] D. E. Thomas and P. R. Moorby. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, 1996.
- [UML17] *Object Management Group: Unified Modeling Language (UML) Version 2.5.1*. OMG Document Number: formal/17-12-05. Object Management Group. 2017. URL: <https://www.omg.org/spec/UML/2.5.1/PDF>.
- [Val15] Antonio Vallecillo. On the Industrial Adoption of Model Driven Engineering. Is your company ready for MDE? *International Journal of Information Systems and Software Engineering for Big Companies (IJISEBC)* 1(1), 2015, pp. 52–68.
- [Val92] Antti Valmari. A stubborn attack on state explosion. *Formal Methods in System Design* 1(4), 1992, pp. 297–322. DOI: <http://dx.doi.org/10.1007/BF00709154>.
- [Van00] H. Vangheluwe. DEVS as a Common Denominator for Multi-Formalism Hybrid Systems Modelling. In: *IEEE International Symposium on Computer-Aided Control System Design*, pp. 129–134. 2000. DOI: [10.1109/CACSD.2000.900199](https://doi.org/10.1109/CACSD.2000.900199).
- [Vau86] Jacques Vautherin. Parallel systems specifications with coloured Petri nets and algebraic specifications. In: *Advances in Petri Nets 1987*, Lecture Notes in Computer Science, vol. 266, pp. 293–308. Springer, 1986. DOI: [10.1007/3-540-18086-9_31](https://doi.org/10.1007/3-540-18086-9_31).

- [VB04] Markus Völter and Jorn Bettin. Patterns for model-driven software-development. In: *Proceedings of the 9th European Conference on Pattern Languages of Programms (EuroPLOP '2004), Irsee, Germany, July 7-11, 2004*. Pp. 525–560. 2004. URL: http://hillside.net/europlop/HillsideEurope/Papers/EuroPLOP2004/2004%5C_VoelterEtAl%5C_PatternsForModel-Driven.pdf.
- [vDKV00] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *ACM SIGPLAN Notices* 35(6), 2000, pp. 26–36. DOI: [10/d724t3](https://doi.org/10.1145/346133).
- [Ver06] Verilog-AMS Working Group. *IEEE Standard for Verilog Hardware Description Language (IEEE Std 1364-2005)*. IEEE Computer Society. 2006. DOI: [10.1109/IEEESTD.2006.99495](https://doi.org/10.1109/IEEESTD.2006.99495).
- [Ver14] Verilog-AMS Working Group. *Verilog-AMS Language Reference Manual Version 2.4*. IEEE Computer Society. 2014. URL: <http://www.accelera.org/images/downloads/standards/v-ams/VAMS-LRM-2-4.pdf>.
- [VFA15] Le Hung Vu, Damien Foures, and Vincent Albert. ProDEVS: An Event-driven Modeling and Simulation Tool for Hybrid Systems Using State Diagrams. In: *Proceedings of the 8th International Conference on Simulation Tools and Techniques*, pp. 29–37. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2015. DOI: [10.4108/eai.24-8-2015.2261136](https://doi.org/10.4108/eai.24-8-2015.2261136).
- [VHDL11] *IEC/IEEE International Standard - Behavioural languages - Part 1-1: VHDL Language Reference Manual*. IEEE Computer Society. 2011. DOI: [10.1109/IEEESTD.2011.5967868](https://doi.org/10.1109/IEEESTD.2011.5967868).
- [Vid+09] J. Vidal, F. de Lamotte, G. Gogniat, P. Soulard, and J. P. Diguët. A co-design approach for embedded system modeling and code generation with UML and MARTE. In: *Automation Test in Europe Conference Exhibition 2009 Design*, pp. 226–231. 2009. DOI: [10.1109/DATE.2009.5090662](https://doi.org/10.1109/DATE.2009.5090662).
- [VL93] Bart Vergauwen and Johan Lewi. A linear local model checking algorithm for CTL. In: *International Conference on Concurrency Theory*, Lecture Notes in Computer Science, vol. 715, pp. 447–461. 1993.
- [Vla+05] Boštjan Vlaovič, Aleksander Vreže, Zmago Brezočnik, and Tatjana Kapus. Verification of an SDL Specification — a Case Study. *Elektrotehniški vestnik (Electrotechnical Review)* 72(1), 2005, pp. 14–21.
- [Völ+13] Markus Völter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013. URL: <http://www.dslbook.org>.
- [Völ09] Markus Völter. MD* Best Practices. *Journal of Object Technology* 8(6), 2009, pp. 79–102. DOI: [10.5381/jot.2009.8.6.c6](https://doi.org/10.5381/jot.2009.8.6.c6).

- [VW86] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In: *Proceedings 1st Annual IEEE Symp. on Logic in Computer Science, LICS'86, Cambridge, MA, USA, 16–18 June 1986*, pp. 332–344. IEEE, 1986.
- [WH05] E. Woods and R. Hilliard. Architecture Description Languages in Practice Session Report. In: *5th Working IEEE/IFIP Conference on Software Architecture (WICSA '05)*, pp. 243–246. 2005. DOI: [10.1109/WICSA.2005.15](https://doi.org/10.1109/WICSA.2005.15).
- [Whi+13] Jon Whittle, John Hutchinson, Mark Rouncefield, Håkan Burden, and Rogardt Heldal. Industrial Adoption of Model-Driven Engineering: Are the Tools Really the Problem? In: *Model-Driven Engineering Languages and Systems*, Lecture Notes in Computer Science, vol. 8107, Springer, 2013. DOI: [10.1007/978-3-642-41533-3_1](https://doi.org/10.1007/978-3-642-41533-3_1).
- [WK98] Jos B. Warmer and Anneke G. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley Longman Publishing Co., Inc., 1998.
- [YHF16] Aznam Yacoub, Maâmar El-Amine Hamri, and Claudia S. Frydman. Using dev-promela for modelling and verification of software. In: *Proc. ACM Conference on SIGSIM Principles of Advanced Discrete Simulation, SIGSIM-PADS*, pp. 245–253. 2016. DOI: [10.1145/2901378.2901388](https://doi.org/10.1145/2901378.2901388).
- [Yov97] Sergio Yovine. Kronos: a verification tool for real-time systems. *International Journal on Software Tools for Technology Transfer* 1(1-2), 1997, pp. 123–133.
- [Zei76] Bernard P. Zeigler. *Theory of Modelling and Simulation*. Wiley, 1976.
- [Zei84] Bernard P. Zeigler. *Multifaceted Modelling and Discrete Event Simulation*. Academic Press Professional, Inc., 1984.
- [ZKP00] Bernard P. Zeigler, Tag Gon Kim, and Herbert Praehofer. *Theory of Modeling and Simulation*. 2nd. Academic Press, Inc., 2000.
- [ZS98] Bernard P. Zeigler and J S. Lee. Theory of quantized systems: Formal basis for DEVS/HLA distributed simulation environment. In: *Proceedings of SPIE - The International Society for Optical Engineering*, 1998.