Infrastructure for DEVS Modelling and Experimentation

Hongyan Song
Supervisor : Prof. Hans Vangheluwe

School of Computer Science
McGill University, Montréal, Canada

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfilment of the requirements of the degree of
Master of Science in Computer Science

Abstract

After over thirty years of research and development, Discrete EVent system Specification (DEVS) has been widely accepted and applied in the Modelling and Simulation community. Recently, standardizing DEVS formalism and expanding DEVS application have become major chellanges for DEVS researchers. In this thesis, we present our efforts to facilitate the process of DEVS modelling, and to promote DEVS standardization and application.

The Infrastructure for DEVS Modelling and Exmperimenting provides facilities to facilitate the DEVS modelling process at four different levels. At the modelling level, a visual DEVS modelling environment has been built, in which DEVS models can be created graphically, and simulator-neutral model representation in modelling language Modelica can be generated automatically. At the model verification level, a Modelica Model Compiler has been developed, by which the simulator-neutral model representations are checked automatically and translated into simulator-specific model representations. At the simulation level, simulation trace has been standardized using the XML DTD. Interfaces for generating standardized simulation trace represented in XML for Python DEVS has been provided. At the model validation level, a Visual DEVS Trace Plotter has been developed, by which the standardized DEVS simulation trace in XML can be plotted visually.

Après plus de trente ans de recherche et de développement, les spécifications de système discrètes d'événement (DEVS), ont été largement acceptées et appliquées dans la communauté de modelage et de simulation. Récemment, la normalisation du formalisme de DEVS et les extensions de l'application de DEVS sont devenus des défis principaux pour les chercheurs de DEVS. Dans cette thèse, nous présentons nos efforts de faciliter le processus de modelage en DEVS, et de favoriser la normalisation et l'application de DEVS.

L'infrastructure pour le modelage et expérimentation de DEVS fournit des équipements pour faciliter le processus de modelage de DEVS à quatre niveaux différents. Au niveau modelant, un environnement de modelage visuel de DEVS a été construit. Cet environnement permet la création des modèles graphiques de DEVS aussi bien que la génération automatique d'une représentation de modèle simulateur-neutre dans la langue modelante Modelica. Au niveau de la vérification de modèle, un compilateur de modèle de Modelica a été développé, par lequel les représentations des modèles simulateur-neutres sont vérifiées automatiquement et traduites en représentations de modèles simulateur-spécifiques. Au niveau de la simulation, la trace de simulation a été normalisée en utilisant le XML DTD. Des interfaces pour produire une trace normalisée de simulation représentée dans XML pour le Python DEVS ont été fournies. Au niveau de la validation de modèle, un traceur visuelle de trace de DEVS a été développée pour tracer visuellement la trace normalisée de simulation de DEVS.

Acknowledgements

Some parts of this thesis are based on previous works done in the MSDL (Modelling, Simulation, and Design Lab) at McGill University. I would like to express my sincere gratitudes to people who have made contributions to this thesis.

First, many thanks to my supervisor Prof. Hans Vangheluwe. His insight and passion in the modelling and simulation are the keys to the success of this thesis.

Thanks to Steven Xu's μModelica compiler and help on reusing components of that compiler.

Thanks to Denis Dube's original DEVS meta-model and help on using State Charts in AToM^3.

Thanks to Ernesto Posse's ideas on Python DEVS code generation in AToM^3 and help on system maintenance.

And finally, thanks to people who have read and commented on this thesis.

Contents

List of Figures

List of Tables

Introduction

Differential equations have been commonly accepted as a standard modelling formalism for describing continuous-time dynamic systems for many years. However, though there are many formalisms used for representing discrete event systems, there is still no consensus on a commonly accepted one for discrete event modeling.

Ho [Y.C89] lists the following challenges that must be addressed when one tries to develop a universally applicable modelling framework for Discrete Event Dynamic Systems (DEDS):

1. The discontinuous nature of Discrete Events
2. The continuous nature of most Performance Measures
3. The importance of Probabilistic Formulation
4. The need for hierarchical modelling
5. The presence of Dynamics as well as Structure
6. The feasibility of the Computational Burden
7. The need for both experimental and theoretical components

Zeigler [ZV93] claims that a system theoretic framework can provide a solid modeling foundation for addressing these issues in a unified manner. He further argues that discrete event simulation models can be captured as a subclass of systems using the Discrete EVent system Specification (DEVS) formalism, which is built on system theory.

DEVS was first proposed by Zeigler in the 1970's[ZPK00]. As a mathematical basis for discrete event modelling, DEVS provides not only a formal representation of discrete event dynamic systems that is independent of any computer realization, but also a guidline for how to build abstract DEVS simulation engines to simulate the models. Since its inception, DEVS has attracted a lot of attention, though no standard has emerged yet.

After over thirty years of research and development, the DEVS formalism has been extended from its original formalism currently known as Classic DEVS[Van04], which only supports sequential discrete event modelling and simulation, to Parallel DEVS, CELL-DEVS, Dynamic DEVS, Real-time DEVS and so forth[ZPK00]. Many DEVS modelling and simulation evironments, such as DEVSJava[Zei05], Python DEVS[BV02], and ADEVS[Nut05] have been designed and implemented. With the support of so many tools, DEVS has been applied in many different fields. [ZKB99] introduces an approach of distributed supply chain modelling using Parallel DEVS; [AEA$^+$02] presents a method of modelling fire spreading using Cell-DEVS; and [SK94] proposes ideas of modelling a water supply system using DEVS.

The major goal of modelling and simulation is to facilitate the understanding of the system under study. To achieve the goal of DEVS modelling and simulation, we first have to build convenient tools to make DEVS modelling and simulation easy to use and to be understood. There are many efforts in this direction. [PP93] presents visual modelling of DEVS using higraphs; [HK06] introduces a specification language DEVSpecl for DEVS model representation; [AHW05] and [ST00] propose ideas on trace visualization for studying complex systems.

In this thesis, we present our efforts to facilitate and promote DEVS modelling and simulation – an Infrastructure for DEVS Modelling and Experimentation. At the modelling level, we use meta-modelling and a visual modelling language to construct a visual DEVS modelling environment. In this approach, we not only build a meta-model for DEVS, but also our entire modelling environment is modeled. At the textual model representation level, we describe our models in Modelica[Fri04], which is a popular model description language for both continuous and discrete models, and use a Model Compiler to translate the simulator-neutral Modelica represented models into programming language-specific simulation models. At the simulation level, we standardize the trace representation in XML, and build a generic Visual Trace Plotter which can plot any trace that follows the XML trace description DTD.

The organization of this thesis is as follows. In chapter 1, we first review the Discrete EVent system Specification formalism and some of its popular variants. In chapter 2, we discuss the design architecture of the Infrastructure, in which a big picture of the whole system is given. In chapter 3, 4, and 5, we presents the sub-systems of the Infrastructure. In chapter 3, we introduce DEVS simulation trace standardization and the Visual Trace Plotter. In chapter 4, we discuss representing DEVS models in Modelica and the Model Compiler, which compiles models described in Modelica into models represented in Python DEVS. In chapter 5, we presents issues of meta-modelling the DEVS formalism, building a visual DEVS modelling environment, and generating Modelica model representations from visual DEVS models. In chapter 6, we give a case study, in which an example demonstrating the whole process of building DEVS models using the Infrastructure is discussed. Finally, in chapter 7, we draw conclusions on the Infrastructure and talk about further efforts to improve the Infrastructure.

1 Discrete EVent system Specification (DEVS) Formalism

1.1 Introduction

DEVS (Discrete EVent system Specification) was first proposed by Zeigler in the 1970's as a mathematical basis for deterministic discrete event modelling ([ZPK00]). DEVS provides a formal representation of discrete event dynamic systems that is capable of mathematical operation and independent of any computer realization. In DEVS, a complex system can be specified by two different kinds of models, atomic models and coupled models. Atomic models can be connected together to form coupled models. The composed coupled models and atomic models can be further connected to create more complex hierarchical coupled models.

Atomic models are the basic units in DEVS[Van04]. The structure of atomic models is described as components with inputs, states, and outputs. The behaviour of an atomic model is specified by functions that reflect the state change and output to the environment of the model corresponding to different inputs. The input and output concepts in atomic DEVS have two levels of meaning. First, each model may have input and output ports, by which the model can interact with its environment. Second, what can go through the ports are input events that the environment sends to the model and output events that the model sends back to the environment. The input and output in an atomic model mean not only events but also ports through which events come in or send out. The modularity of the atomic models comes from the port concept, by which the implementation of an atomic model only needs to consider the input events coming from its input ports and send its output to appropriate output ports. Issues such as how events come to input ports and how to send an event to other models do not need to be considered at the atomic model level.

The states specified in an atomic model are called sequential states, which are not the complete model states. They actually represent identifications of discrete model state space partitions[PP93], which means that the state space of an model can be partitioned exclusively into many different parts, and each of them is identified by a name (sequential state). The behaviour of a model can be seen as the model staying at different sequential states. Within different sequential states, the model can react to different events or react differently to the same events. After processing an event, the model may changes its state.

The behaviour of a model changing its sequential state from one to another is called state transition. How a model does its state transitions is determined by its transition functions. Input events lead to "external event transitions", which means that upon occurrence of an input event, whether the model transits to another state and how to transit to another state is determined by the *external event transition function.* The time interval a model stays in a particular state in the absence of external events is determined by the *time advance function.* When this time has elapsed, an event is triggered. The time scheduled events are called internal events. Upon the occurrence of an internal event, the model produces an output event by its *output function* and transits to a new state determined by its *internal state transition function.*

Atomic DEVS models can be coupled together by connecting their input and output ports to build a coupled model. Coupled models have their own input and output ports, which can be used to connect other coupled or atomic models to create even larger hierarchical models. When used to connect to other models, coupled models are not distinguishable from atomics models, so they are reusable to build hierarchical models in the same way as atomic components.

Since its inception, DEVS has been studied over 30 years. The specification has evolved and been extended by many researchers. The original version is now called Classic DEVS, and the latest edition is called Parallel DEVS ([ZPK00], [AB94], [Van04]). The major difference between Classic DEVS and Parallel DEVS is the way they deal with transition collisions. Classic DEVS does not provide any mechanism of processing transition collisions at atomic level. It is the task of the select function of a coupled model to break the tie between simultaneous transitions. In Parallel DEVS, transition collisions have been broken by the *confluent transition function* at atomic level. So there is no need at coupled level to process simultaneous transitions (see section 1.2.2).

In addition to Classic DEVS and Parallel DEVS, there are many other extended versions of DEVS. The most significant extensions are Cell-DEVS ([WG01a], [WG01b]), and Dynamic Structure DEVS (DSDEVS) ([Bar97], [Uhr01]). Cell-DEVS makes DEVS more efficient in modelling phenomena that are suitable for state space partitioning. DSDEVS makes it possible for DEVS to model situations that include dynamic structure changes.

One advantage of DEVS is that it provides not only a modular, hierarchical modelling framework, but also a concept of abstract simulator (i.e., operational semantics), by which the model behaviour can be generated. There are many different implementations of DEVS modelling and simulation tools available now ([Zei05], [BV02], [Nut05]). Each one has its own features and functionalities. We will investigate some of them to get a flavour of the difference.

The Modelling and Simulation industry in nature has a tight relationship with computer and software development technology. During the time of DEVS evolution significant progress has been made in the software engineering industry. From the software design point of view, structural software design has gradually finished its mission, object-oriented software development and model driven architechure become more and more populer. From a computing language aspect, much effort is spent on the research of visual language and software visulaization. Though these new trends have not reached the level that can replace all the tranditional technologies, they can potentially promote and facilitate the development and application of modelling and simulation technology. In this thesis, we present an infrastucture we built for facilitating DEVS visual modelling and experimentation, in which some new technologies like visual meta-modelling, and visual languages are used. Before we further discuss our infrastructure, let us review some related background about DEVS.

We will first discuss the basic DEVS formalisms, Classic DEVS and Parallel DEVS. Then we will look into algoritms for DEVS simulators. After that, we will present the features of some implementations of DEVS modelling and simulation tools. We will use DEVSJava, ADEVS, and Python DEVS as examples to explain the features and differences.

1.2 Discrete Event System Specification

The DEVS formalism fits the general structure of deterministic, causal systems in classical systems theory. DEVS allows for the description of system behaviour at two levels. At the lowest level, an atomic DEVS describes the autonomous behaviour of a discrete-event system as a sequence of deterministic transitions between sequential states as well as how it reacts to external input (events) and how it generates output (events). At the higher level, a coupled DEVS describes a system as a network of coupled components. The components can be atomic DEVS models or coupled DEVS in their own right. The connections denote how components influence each other. In particular, output events of one component can become, via a network connection, input events of another component. The DEVS formalism is closed under coupling: for each coupled DEVS, a resultant atomic DEVS can be constructed ([Zei84]). As such, any DEVS model, be it atomic or coupled, can be replaced by an equivalent atomic DEVS. The construction procedure of a resultant atomic DEVS is also the basis for the implementation of an abstract simulator or solver capable of simulating any DEVS model. As a coupled DEVS may have coupled DEVS components, hierarchical modelling is supported.

To be suitable for simulation under simulators implemented in different architectures, there are basically two types of DEVS specifications. Classic DEVS specifies models that are suitable for simulation under sequential DEVS simulators. Parallel DEVS defines models that are amenable to parallel simulation.

1.2.1 Classic DEVS

Classic Atomic DEVS Formalism

The atomic DEVS formalism is a structure describing the different aspects of the discrete-event behaviour of a system([ZPK00]):

$$AtomicDEVS \equiv (X,\ Y,\ S,\ \delta_{\text{ext}},\ \delta_{\text{int}},\ \lambda\ ,\ t_{\text{a}}\ )$$

The *time base* T is continuous and is not mentioned explicitly[Van04]

$$T = \mathbb{R}.$$

Each atomic DEVS model has a set of input events X, a set of output events Y, and a set of sequential states S. The model transits its state through state transition functions. Upon arrival of an input event x, the new state is determined by the return value of $\delta_{ext}((s,\ e),\ x)$, which means that the new state is determined by the current state s, the time it stays at s, and the input event x. When there is no external event, the time interval the model stays on its current state is determined by applying the t_a function to the current state. And the next state of the model is determined by $\delta_{int}(s)$, where s is the current state. Just before an internal state transition, the model can generate an output event by applying the λ function to the current state s, i.e. $\lambda(s)$. Details on each part of the formalism are explained as below.

The *state set* S is the set of *sequential states*: the DEVS dynamics consists of an ordered sequence of states from S. Typically, S will be a structured set (a product set)

$$S = \times_{i=1}^{n} S_i$$

This formalizes multiple (n) concurrent parts of a system. It is noted how a structured state set is often synthesized from the state sets of concurrent components in a coupled DEVS model.

The time the system remains in a sequential state before making a transition to the next sequential state is modelled by the time advance function

$$t_a : S \rightarrow \mathbb{R}^{+}_{0,\ +\infty}.$$

As time in the real world always advances, the image of t_a must be non-negative numbers. $t_a = 0$ allows for the representation of instantaneous transitions: no time elapses before transition to a new state. Obviously, this is an abstraction of reality which may lead to simulation artifacts such as infinite instantaneous loops which do not correspond to real physical behaviour. If the system is to stay in an end-state s forever, this is modelled by means of $t_a(s) = +\infty$.

The internal transition function

$$\delta_{int} : S \rightarrow S$$

models the transition from one state to the next sequential state. δ_{int} describes the behaviour of a Finite State Automaton; t_a adds the progression of time.

It is possible to observe the system output. The output set Y denotes the set of admissible outputs. Typically, Y will be a structured set (a product set)

$$Y = \times_{i=1}^{l} Y_i$$

This formalizes multiple (l) output ports. Each port is identied by its unique index i. In a user-oriented modelling language, the indices would be derived from unique port names.

The output function

$$\lambda : S \rightarrow Y \cup \{\emptyset\}$$

maps the internal state onto the output set. Output events are only generated by a DEVS model at the time of an internal transition. At that time, the state before the transition is used as input to λ. At all other times, the non-event $\emptyset$ is output.

To describe the total state of the system at each point in time, the sequential state $s \in S$ is not sufficient. The elapsed time e since the system made a transition to the current state s needs also to be taken into account to construct the total state set

$$Q = \{(s,\ e) \mid s \in S,\ 0 \leq e \leq t_a(s)\}$$

The elapsed time e takes on values ranging from 0 (transition just made) to $t_a(s)$ (about to make transition to the next sequential state). Often, the time left σ in a state is used:

$$\sigma = t_a(s) - e$$

Up to now, only an autonomous system was described: the system receives no external inputs. Hence, the input set X denoting all admissible input values is defined. Typically, X will be a structured set (a product set)

$$X = \times_{i=1}^{m} x_i$$

This formalizes multiple (m) input ports. Each port is identied by its unique index i. As with the output set, port indices may denote names.

The set Ω contains all admissible input segments ω

$$\omega : T \rightarrow X \cup \{\emptyset\}$$

In discrete-event system models, an input segment generates an input event different from the non-event $\emptyset$ only at a finite number of instants in a bounded time-interval. These external events, inputs x from X, cause the system to interrupt its autonomous behaviour and react in a way prescribed by the external transition function

$$\delta_{ext} : Q \times X \rightarrow S$$

The reaction of the system to an external event depends on the sequential state the system is in, the particular input and the elapsed time. Thus, δ_{ext} allows for the description of a large class of behaviours typically found in discrete-event models (including synchronization, preemption, suspension and re-activation).

When an input event x to an atomic model is not listed in the δ_{ext} specification, the event is ignored.

Classic Coupled DEVS

The coupled DEVS formalism describes a discrete-event system in terms of a network of coupled components[Van04].

$$CoupledDEVS \equiv < X_{self},\ Y_{self},\ D,\ \{M_d\},\ \{I_d\},\ \{Z_{i,\ d}\},\ Select >,$$

The component *self* denotes the coupled model itself. X_{self} is the (possibly structured) set of allowed external inputs to the coupled model. Y_{self} is the (possibly structured) set of allowed (external) outputs of the coupled model. D is a set of unique component references (names). The coupled model itself is referred to by means of $self$, a unique reference not in D.

The set of components is

$$\{M_i \mid i \in D\}.$$

Each of the components must be an atomic DEVS

$$M_i = (X_i,\ Y_i,\ S_i,\ \delta_{\text{ext},i},\ \delta_{\text{int},i},\ \lambda_i\ ,\ t_{\text{a},i}\ ),\ \forall i \in D$$

The set of inuencees of a component, the components influenced by $i \in D \cup \{self\}$ is I_i. The set of all influencees describes the coupling network structure

$$\{I_i \mid i \in D \cup \{self\}\}$$

For modularity reasons, a component (including $self$) may not influence components outside its scope the coupled model, rather only other components of the coupled model, or the coupled model $self$:

$$\forall i \in D \cup \{self\}{:}Ii \subseteq D \cup \{self\}$$

This is further restricted by the requirement that none of the components (including $self$) may influence themselves directly as this could cause an instantaneous dependency cycle (in case of a 0 time advance inside such a component) akin to an algebraic loop in continuous models:

$$\forall i \in D \cup \{self\}{:} i \notin I_i$$

Note how one can still encode a self-loop ($i \in I_i$) in the internal transition function.

To translate an output event of one component (such as a departure of a customer) to a corresponding input event (such as the arrival of a customer) in influencees of that component, output-to-input translation functions Z_i are defined:

$$\{Z_{i,j} \mid i \in D \cup \{self\},\ j \in I_i\},$$

$$Z_{self,j} : X_{self} \rightarrow X_j,\ \forall j \in D,$$

$$Z_{i,self} : Y_i \rightarrow Y_{self},\ \forall i \in D,$$

$$Z_{i,j} : Y_i \rightarrow X_j,\ \forall i, j \in D.$$

Together, I_i and $Z_{i,j}$ completely specify the coupling (structure and behaviour).

As a result of coupling of concurrent components, multiple state transitions may occur at the same simulation time. This is an artifact of the discrete-event abstraction and may lead to behaviour not related to real-life phenomena. A logic-based foundation to study the semantics of these artifacts was introduced by Radiya and Sargent [RS94]. In sequential simulation systems, such transition collisions are resolved by means of some form of selection of which of the components' transitions should be handled first. This corresponds to the introduction of priorities in some simulation languages. The coupled DEVS formalism explicitly represents a *select* function for tie-breaking between simultaneous events:

$$select :\ 2^D \rightarrow D$$

select chooses a unique component from any non-empty subset E of D:

$$select(E) \subset E.$$

The subset E corresponds to the set of all components having a state transition simultaneously.

Model coupling is the mechanism used in DEVS to group many models into a composite, coupled model. Coupled models are not distinguishable from atomic models when they are coupled with atomic models. Based on the feature of closure under coupling of DEVS, complex system can be hierarchically constructed by model coupling.

1.2.2 Clarification of Concepts

Sequential State and State Space Partition

As we mentioned earlier, Sequential State is one of the key concepts used in DEVS to describe model behaviour. The Sequential State of an atomic DEVS model is the exclusive partition identifications of the model state space ([PP93]). The trajectory of the model behaviour can be seen as a sequence of numbered pairs of Sequential State and holding time, where the holding time means how long the system remains in the Sequential State. The Sequential State has different meanings depending on the model state, and it is determined by one or some of the model state variables.

A traffic light can be a good example to explain the difference between Sequential State and model state. In DEVS, a traffic light can be modelled as an atomic DEVS. The atomic traffic light DEVS model state may have many attributes such as `producer`, `location`, `light colors` and so forth. However, no matter how many attributes the model state may have, the life cycle of a typical traffic light can be represented as repeating changing its color to `Red`, `Yellow`, and `Green` after certain time intervals. So the Sequential State of a typical traffic light DEVS model can be reprensented as the set of {`Red`, `Yellow`, `Green`}. In this example, the sequential state of a traffic light is determined by the values of one attribute (`light colors`) of the model state.

External, Internal, and Output Events

There are three kinds of events for a DEVS model, Input Events (External Events), Time Scheduled Events (Internal Events), and Output Events. Input Events are events from the model's environment; Time Scheduled Events are events scheduled by the model's time advance function; and Output Events are events the model generates to communicate with its environment. Input Events cause an external state transition, so Input Events are also called External Events. Time Scheduled Events lead to an internal state transition. That is why they are also called Internal Events. Output Events are the side-effect of processing the internal events. One thing needs to be remembered is that only an internal state transition can generate Output Events. If an external transition wants to produce an Output Event, it must schedule an Internal Event, and wait until the scheduled time interval has elapsed to generate an Output Event via the scheduled Internal Event.

State Transition and Transition Collisions

In DEVS, State Transition means, upon the occurrence of a certain event, the behaviour that a model changes its sequential state from one to another. External events can cause an external state transition and an internal event will lead to an internal state transition. What will happen if more than one event occurs at the same simulation time? Chow presents the concept of Transition Collisions to represent mutually interfering simultaneous events ([ABK94]). Transition Collisions can happen between multiple external events, external events and internal events, and multiple internal events. Different measures to deal with Transition Collisions among simultaneous events lead to the distinction between Classic DEVS and Parallel DEVS.

With these concepts in mind, now let us discuss Parallel DEVS.

1.2.3 Parallel DEVS

In Classic DEVS, the *select* function of a coupled model is used to break the tie of simultaneous internal events. At the simulation level, the simulator uses the *select* function to sequentialize these events. There is no mechanism both at atomic level and coupled model level to deal with the situations when an external event and an internal event occur simultaneously. In Classic DEVS, the external events are chosen to override any simultaneous internal event. Also, there is no mechanism for processing simultaneous external events. The simultaneous external events are simply serialized or one is selected and others are ignored.

In order to improve the capability for processing simultaneous events of Classic DEVS and to add the ability for parallel simulation, a revised version of DEVS, Parallel DEVS, was proposed. Parallel DEVS extends Classic DEVS at both atomic level and coupled level. At the atomic level, a Parallel DEVS has a input bag, which makes it possible for the Parallel DEVS to queue the distinct events happening simultaneously or the events that have not been processed yet. In addition to the input bag, each atomic Parallel DEVS has a `Confluent` transition function, which is used to handle simultaneous events. Because of the existence of the `Confluent` transition function constructed by the modeller at the atomic level, a coupled Parallel DEVS is relieved of the burden of selecting a right event to process when there are multiple events occuring at the same time.

Parallel DEVS Atomic Models

The atomic model for Parallel DEVS is ([ZPK00])

APDEVS = $(X,\ Y,\ S,\ \delta_{ext},\ \delta_{int},\ \delta_{con},\ \lambda,\ t_a\ )$,

where

X is the set of input events

Y is the set of output events

S is the set of sequential states

$\delta_{ext} :\ Q\ \times\ X^b\ \rightarrow\ S$ is the external state transition function, where X^b is a set of bags over elements in X

$\delta_{int} :\ S\ \rightarrow\ S$ is the internal state transition function

$\delta_{con} :\ S\ \times X^b \rightarrow S$ is the confluent transition function, subject to $\delta_{con}(s,\ \emptyset)\ =\ \delta_{int}(s)$

$\lambda :\ S\ \rightarrow\ Y^b$ is the output function

$t_a :\ S\ \rightarrow\ \mathbb{R}_0^+\ \cup\ \infty$ is the time advance function

$Q = \{(s,\ e)\mid s \in S,\ 0 < e\ < t_a(s)\}$ and e is the elapsed time since the last state transition.

Parallel DEVS improves DEVS formalism in the following areas.

1. The external state transition function δ_{ext} can accept bags of input events. The major difference between the data structure set and bag is that the set is exclusively ordered, which means that an element in a set can not appear more than once. While the bag is unordered, the same element can show up many times. This improvement makes it possible for Parallel DEVS models to accept multiple events at the same time.
2. Parallel DEVS uses the confluent transition function δ_{con} to deal with the transition collisions between external events and internal events. This gives the modeler an opportunity to control the model behaviour when the model receives an external event at the time of an internal state transition. The confluent transition function has two commonly used implementations, *internal events first* or *external events first*. *Internal events first* means, when there is a transition collision between an internal event and an external event, the internal event is processed first, then the external event. This is the default behaviour of the confluent transition function. It can be represented as $\delta_{con}(s, x) = \delta_{ext}(\delta_{int}(s), 0, x)$. This means that the internal event is processed by the internal transition function first, immediately after that, the external event is processed by the external transition function. That is why the elapsed time is 0 for the external transition function. *External events first* has the opposite meaning. The confluent transition function for an *exernal events first* model can be described as $\delta_{con}(s, x) = \delta_{int}(\delta_{ext}(s, t_a(s), x))$. This means that the internal event must wait until the external event has been processed.
3. In Classic DEVS, external events are always processed by the external state transition function δ_{ext}. In Parallel DEVS, an external event can be processed either by the external state transition function δ_{ext} or by the confluent transition function δ_{con} depending on

the time the external event arrives. When an external event arrives at a time e, if $0 < e < t_a(s)$, the event is processed by the δ_{ext} function, if $e = 0$ or $e = t_a(s)$, the event is processed by the δ_{con} function. Because the input of the confluent transition function is a bag, the implementation of the function should guarantee that $\delta_{con}(s, \emptyset) = \delta_{int}(s)$, which means that when the input bag is empty, the confluent transition function should have the same semantics as the internal transtion function.

Parallel DEVS Coupled Models

The structure of a Parallel DEVS coupled model is similar to the $ClassicDEVS$ coupled model except that the former does not have the *select* function ([ZPK00]).

CPDEVS $= < X, Y, D, \{M_d\}, \{I_d\}, \{Z_{i, d}\} >$

Here, X, Y, D, I_d and $Z_{i,d}$ have the same meaning as that for Classic DEVS coupled models, and for each $d \in D$, M_d is a Parallel DEVS model. For coupled models, Parallel DEVS removes the *select* function that exists in Classic DEVS coupled models. This clears the way for parallel simulation.

Now we can summarize the mechanisms used by Classic DEVS and Parallel DEVS to deal with transition collisions as follows. In Classic DEVS, transition collisions among internal events of a coupled model are tackled by the *select* function of a coupled Classic DEVS model; for transition collisions among external events and internal events, external events always override internal events; for transition collisions among external events, there is no mechanism specified in Classic DEVS to deal with this situation (which can actually never occur due to the sequential nature of Classic DEVS). In Parallel DEVS, the confluent transition function at atomic model level is used to tackle transition collisions among external events and internal events; simultaneous external events are collected in the *bag* data structure of Parallel DEVS models, and the external transition function and the confluent transition function of the Parallel DEVS accept *bags* rather than *sets* as their input parameters. Because models of Parallel DEVS are simulated in parallel, there is no transition collision among internal events.

1.3 Abstract DEVS Simulation Engine

The DEVS models can be simulated in many different ways. A generic and modular framework for DEVS simulation was proposed by Zeigler in [ZPK00]. The idea of the framework is that, usually a DEVS model is a hierarchical tree in which Atomic Models as leaf nodes and Coupled Models as root and branch nodes. So the simulator of the DEVS models may have a similar hierarchical structure. In order to distinguish the difference between simulators for coupled and atomic models, the simulators for coupled models are called coordinators. The mapping between a DEVS model and its hierarchical simulator is shown below.

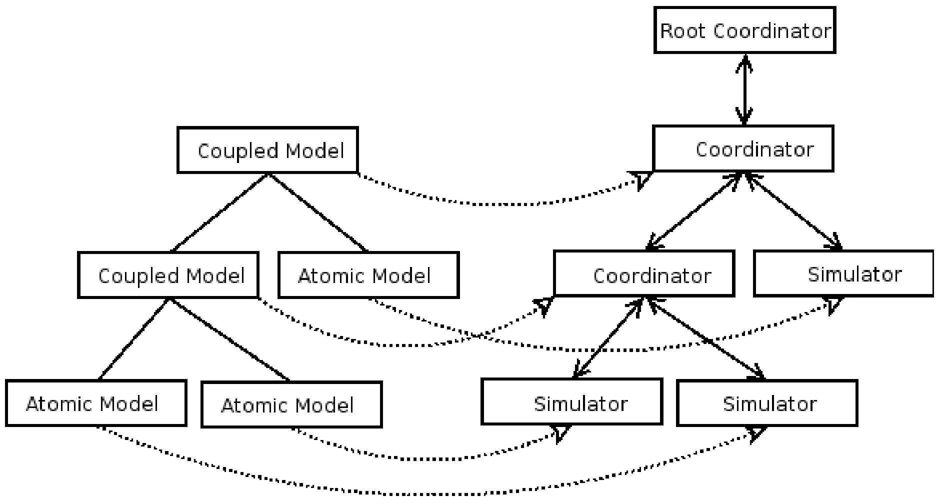


Figure 1.1: Mapping between DEVS Models and Simulators

Because it uses a sequential *select* function at coupled model level to serialize the transition collisions inside a coupled model, Classic DEVS is suitable for sequentialized simulation. On the other hand, Parallel DEVS is by nature suitable for parallel or distributed simulation. Parallel simulation can fully take advantage of the intrinsic parallelism of models. So for much complex systems, parallel simulation could be much more efficient than sequential ones. Certainly, the efficiency comes with more sophisticated algorithms, and much more effort for developing a parallel simulator.

1.3.1 Abstract Simulator for Classic DEVS

The main task of a simulator is to generate a behaviour trace for the model. We mentioned earlier that there are three kinds of events in each DEVS model, external events, internal time-scheduled events, and output events. So a qualified DEVS simulator should have the capability to properly process these kinds of events. At simulation level, each DEVS event is wrapped with a time stamp, which is the virtual time at which the event occurs. In addition to the three kinds of DEVS events, the Classic DEVS simulator introduces a new type of event to synchronize the model initialization process. So there are four kinds of messages sent between DEVS simulators and coordinators. They are:

1. Initialization message (i, t): sent at the initialization time from the parent simulator object to all its subordinates;

2. Internal state transition message (*, t): sent from the coordinator to its imminent child to schedule the next event;
3. Output message (y, t): sent from subordinates to their parents to notify output events of the subordinates;
4. Input message (x, t): sent from a coordinator to its subordinates to notify input events to the subordinates.

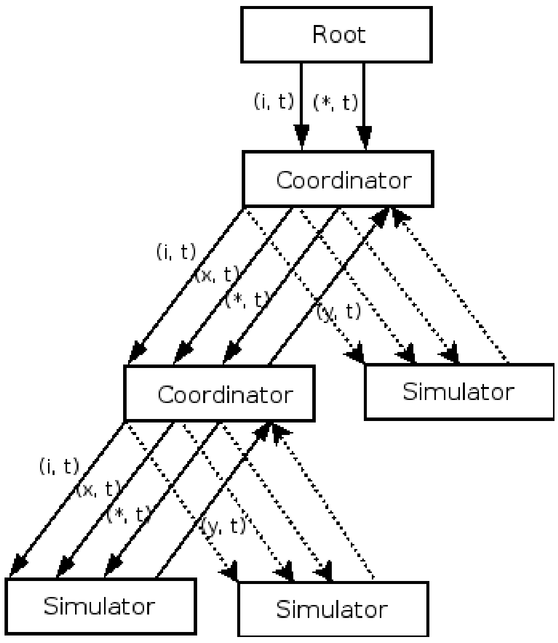


Figure 1.2: Messages Used in Classic DEVS Coordinators and Simulators

The messages sent among coordinators and simulators can be described as in Figure 1.2, in which line segments with text represent messages, and the arrow on a line segment indicates the direction of a message.

Simulator for Classic Atomic DEVS

From Figure 1.2, we can see that the simulator for a atomic model receives three kinds of events. At the beginning of each simulation run, the simulator receives an (i, t) event, at which the simulator updates its last event time t_l and computes its next event time t_n. Then t_l and t_n are sent to the parent coordinator.

1. A (*, t) message can cause an internal event transition, upon which the simulator calls the model's output function to generate an output event (y, t), which is sent to the parent of the simulator. Then, the internal state transition function is called, and the model state is updated. After that the t_l and t_n variables are recomputed and sent to the parent coordinator.

2. When a (x, t) message is received, an external state transition is triggered. The elapsed time is computed, and the external state transition function is called. After the state transition, the t_l and t_n are updated and sent to the parent coordinator.
3. Every time the t_l and t_n are updated, they must be sent to the parent coordinator. By collecting all the t_ls and t_ns of its subordinates, the coordinator can determine its last and next event times. How the coordinator determines its last and next event times will be discussed in the coupled model simulation algorithm.

```
1. when an (i, t) message is received
     $t_l = t - e$
     $t_n = t_l + t_a(s)$
     $send\ t_l,\ t_n\ to\ parent$
2. when a (*, t) message is received
     if ($t == t_n$)
         $y = \lambda(s)$
         $send\ (y,\ t)\ to\ parent$
         $s = \delta_{int}(s)$
         $t_l = t$
         $t_n = t_l + t_a(s)$
         $send\ t_l\ to\ parent$
     else
         $error$
3 when a (x, t) message is received
     if ($t_l <= t <= t_n$)
         $e = t - t_l$
         $s = \delta_{ext}(s,\ e,\ x)$
         $t_l = t$
         $t_n = t_l + t_a(s)$
         $send\ t_l,\ t_n\ to\ parent$
     else
         $error$
```

In this algorithm, variable t_l holds the virtual time when the last event occured, and the t_n represents the simulated time of the next event. During the simulation, the relation $t_n = t_l + t_a(s)$ holds. The variable e denotes the elapsed time since the last event, at a given global time t, we have $e = t - t_l$. Some issues of this algorithm should be remembered. First, the internal transition will be interrupted unconditionally by an external event. There is no mechanism in this algorithm to make an external event wait until the current internal transition finishes. Second, when an internal state transition is interrupted, the unfinished internal transition is discarded. There is no way to recover the internal transition from where it is interrupted. Third, there is no provision in the algorithm for the situation when an internal transition is interrupted but the model state does not change.

Coordinator for Classic Coupled DEVS

As for a simulator for atomic DEVS, a coordinator for a coupled DEVS also receives three kinds of events from its parent and sends one event to its parent. The only difference is that the coordinator can also get events from its subordinates and send events to its subordinates.

1. When an (i, t) event is received, the coordinator forwards the (i, t) to all its direct subordinates, and collects all the subordinates' next event times and last event times, and saves them in an event list. The last and next event time for the coupled model are computed using $t_l = max\{t_{ld} \mid d \in D\}$ and $t_n = min\{t_{nd} \mid d \in D\}$.
2. When a (*, t) event arrives, the coordinator uses the *select* function to determine which subordinate should be executed. After the execution of the subcomponent, the coordinator updates its last time and next time.
3. When a (x, t) event arrives, the coordinator first checks which components are connected to the port from which the (x, t) event comes, and sends the (x, t) to the components. It then updates its last and next event time.
4. When receiving an output event (y_{d*}, t) from the imminent component d*, the coordinator first checks whether the output port of d* is connected to the coupled model. If it is, then the event will be sent to the coordinator's parent coordinator. And then, the coordinator collects the subordinates that connect with the output port of d*, and sends (y_{d*}, t) to the subordinates.

```
1. when a (i, t) message is received

     for each d in D
          send (i, t) to d

     get all $t_{nd}$s and update event-list
     sort event-list according to the value of $t_{nd}$
     $t_l = max\{t_{ld} \mid d \in D\}$
     $t_n = min\{t_{nd} \mid d \in D\}$
     send $t_l$, $t_n$ to parent

2. when a (*, t) message is received

     if $(t == t_n)$
          d* = select(event-list)
          send (*, t) to d*
          get $t_{nd*}$ from d* and update the event-list
          sort the event-list
          $t_l = t$
          $t_n = min\{t_{nd} \mid d \in D\}$
          send $t_l$, $t_n$ to parent
     else
          error
```

```
3. when a ($y_{d*}$, t) message is received from port $p_{d*}$

       if $p_{d*}$ connects with one of the current coupled model's output ports
           send ($y_{d*}$, t) to parent
       for all $d \in D$ and d has connection with $p_{d*}$
           send ($y_{d*}$, t) to d

4. when a (x, t) message is received at port p

       if ($t_l$ <= $t$ <= $t_n$)
           for all $d \in D$ and d has connection with p
               send (x, t) to d through port pd that connects with p
               get all tnds and update event-list
           sort the event-list
           $t_l = t$
           $t_n = min\{t_{nd} \mid d \in D\}$
           $send\ t_l,\ t_n\ to\ parent$
       else
           error
```

Here, t_l and t_n have the same meanings as that in atomic simulators. The event-list is a list of triple $(d,\ t_{ld},\ t_{nd})$, and d* means the selected component whose event will be processed next.

Root Coordinator

The root coordinator is mainly a time scheduler. There is no real DEVS model corresponding to the root coordinator. Attached to it is the real simulator that has a corresponding DEVS model. At the beginning of each simulation run, the root sends a (i, t) message to its child (the attached simulator or coordinator), and computes the next event time. Then, it repeatedly sends (*, t) messages to its children and computes the next event time until the simulation ends.

```
$t = t_0$
send (i, t) to the child
$t = child.t_n$

do
    send (*, t) to the child
    $t = child.t_n$
until simulation end condition
```

1.3.2 Abstract Simulator for Parallel DEVS

The model specification for Classic DEVS in nature does not support parallel simulation. Parallel DEVS has revised the parts that impede parallel simulation of Classic DEVS models. Consequently the simulator for Parallel DEVS needs the mechanisms that support parallelism.

When it comes to parallel simulation, one important issue is how to synchronize the parallelized executions. Typically there are two different strategies used to synchronize parallel processors, conservative techniques and optimistic techniques ([AT02], [ZPK00]). The conservative method uses time stamped messages to synchronize the parallel processors. A processor is allowed to process an event at time t, only when it is certain that no other events will arrive with a time stamp that is smaller than t. This method implies that at any time only the event with the smallest time stamp can be executed. In an optimistic parallel simulation, the processor will proceed assuming that it will not receive an event with a time stamp less than the one that is being processed. When it receives an event with a time stamp t smaller than events that have been processed, the processor will roll back the processed events to the first event whose time stamp is smaller than t.

The overhead of the conservative method is the posibly huge number of synchronization messages that are used to determine which event is the one that has smallest time stamp. In the optimistic method, in order to roll back the events, all the state information of simulation must be saved. When the system is complicated, and the state can be very large and hence the memory requirement is not a small issue.

The Parallel DEVS formalism is in nature for parallel simulation, either for the conservative method or for the optimistic approach. In the following part of this chapter, we will discuss an algorithm for conservative simulation of Parallel DEVS ([ABK94]).

As in Classic DEVS, there is a mapping structure between Parallel DEVS models and Parallel DEVS simulators. The synchronization between simulators and coordinators is also done by sending messages. In order to support the parallelism, the messages used in the Parallel DEVS simulator are slightly different from those of the Classic DEVS simulator.

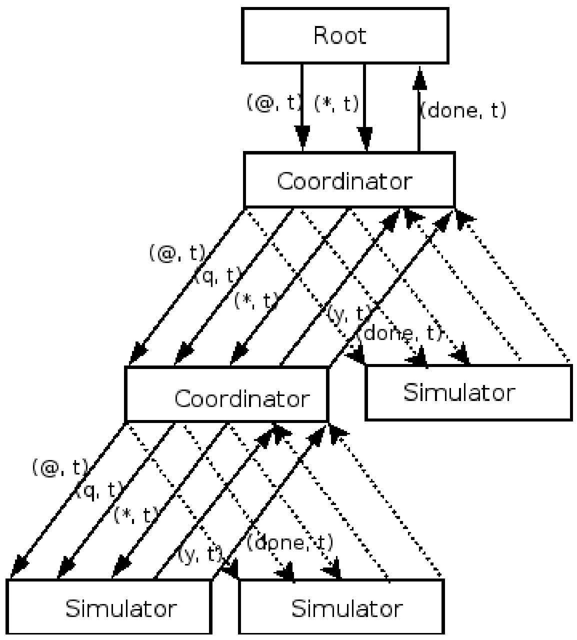


Figure 1.3: Messages for Parallel DEVS Coordinators and Simulators

Figure 1.3 shows the messages used for the implementation of a conservative Parallel DEVS simulator. There are five kinds of messages, (@, t), (q, t), (*, t), (y, t) and (done, t). (@, t), (q,t), and (*, t) are sent from coordinator to its subordinates, and (y, t), and (done, t) are sent by subcomponents to their parent. (@, t) is used by a coordinator to notify its subcomponents that it is the time to generate output; (q, t) is sent by a coordinator to inform its subordinates of the arrival of an input event; and (*, t) is to trigger the state transitions. The receiver will determine the type of the transition applied based on the context in which the message received. The (y, t) message is sent by the subordinates to their parent to notify an output; and the (done, t) is sent by a subcomponent to notify its parent that a message has been processed.

Abstract Simulator for Atomic Parallel DEVS

The main tasks of the atomic simulator are to process the received messages and report the results to the sender. The algorithm can be described as below.

```
when a (@, t) message is received

    if $t = t_n$
        $y = \lambda(s)$
        send (y, t) to the parent coordinator
        send (done, t) to the parent coordinator
    else
        error

when a (@, t) message is received

    if $t = t_n$
        $y = \lambda(s)$
        send (y, t) to the parent coordinator
        send (done, t) to the parent coordinator
    else
        error

when a (*, t) message is received

    if $t_l <= t <= t_n$ and bag is not empty
        if $t == t_n$
            if bag is empty
                $s = \delta_{int}(s)$
            else
                $s = \delta_{con}(s, bag)$
                empty bag
        else
            $e = e - t_l$
            $s = \delta_{ext}(s, e, bag)$
            empty bag
            $t_l = t$
            $t_n = t_l + t_a(s)$
            send (done, t) to the parent coordinator
    else
        error
```

```
when a (q, t) message is received
    lock the bag
    add q to the bag
    unlock the bag
    send (done, t) to the parent coordinator
```

Rather than finishing the processing of an event in one message, this algorithm uses different messages to notify the simulator when to generate an output, process an arrived event, and when to do a state transition. This gives a clear processing structure at the cost of adding some message sending overhead. Also, the confluent function for the atomic Parallel DEVS models solves the problem of transition collision. However, an internal transition is still unconditionally interrupted when an external event occurs before the scheduled time interval for the transition elapses.

Coordinator for Coupled Parallel DEVS

The (@, t) message received by a coordinator is simply sent to the coordinator's subordinates. The (y, t) message is sent to all the components in the sender's influencees set I_i. If the model of the coordinator is in the I_i, then the (y, t) is sent to the coordinator's parent. For a (*, t) message, the coordinator routes the (q, t) events down to all its atomic influencees.

```
when a (@, t) message is received from the parent coordinator

    if $t == t_n$
        $t_l = t$
        for each child in imminent child set
            send (@, t) to child
            cache child reference in the synchronize set
        wait until (done, t) received from all imminent children
        send (done, t) to parent coordinator
    else
        error

when a (y, t) message is received from child i

    for each influencee j in $I_i$
        $q = z_{i,\,j}(y)$
        send (q, t) to influencee j
        cache j in the synchronize set

    wait until (done, t) received from all influencees

    if $self \in I_i$
        $y = z_{i,\,self}(y)$
        send (y, t) to the parent coordinator

when a (q, t) message received from the parent coordinator
    lock the bag
    add event q to the bag
    unlock the bag
```

```
when a (*, t) message is received from the parent coordinator

    if $t_l <= t <= t_n$

        for each influencee j in $I_{self}$ and each q in bag
            $q = z_{self,\ j}(q)$
            send (q, t) to j
            cache j in the synchronize set
            empty bag
            wail until all (done, t) are received
            for each i in the synchronize set
                send (*, t) to i
            wait until all (done, t) are received
            $t_l = t$
            $t_n = min\{t_i\}$
            clear the synchronize set
            send (done, t) to parent coordinator
        else
            error
```

Root Coordinator for Parallel DEVS

The root coordinator has no corresponding DEVS model. It is attached to a coordinator that simulates a model. By repeatedly sending the (@, t) and (*, t) message to the attached simulator or coordinator, the root coordinator makes the simulation run until a certain termination condition is met.

```
$t = child.t_n$

do

    send (@, t) to the child
    wait until (done, t) received from the child
    send(*, t) to the child
    wait until (done, t) received
    $t = child.t_n$

until simulation end condition
```

There are many different ways to implement a Parallel DEVS simulator. In the next chapter, we will introduce DEVS Java and ADEVS. They both support Parallel DEVS. However, the interfaces of these two implementations are not the same.

1.4 DEVS modelling and Simulation Enviroments

After many years of research in DEVS, there are many modelling and simulation environments for DEVS available. Here, we will give a short introduction about Python DEVS ([BV02]), DEVS Java ([Zei05]), and ADEVS([Nut05]), from which we can get a general idea of DEVS modelling and simulation. More information about the latest development of DEVS modelling and simulation environments can be found at the DEVS standardization web site *http://www.sce.carleton.ca /faculty/wainer/standard/*.

1.4.1 PythonDEVS

Python DEVS is a DEVS modelling and simulation package developed at MSDL (Modelling, Simulation, and Design Lab) headed by prof. Hans Vangheluwe, at McGill university. The original purpose of Python DEVS is for teaching about DEVS, so it is not very complicated. The simulator is implemented in the programming language Python, so it is called Python DEVS. From a modelling and simulation point of view, Python DEVS only supports Classic DEVS modelling and sequentialized simulation. However, with the support of the meta-modelling tool AToM^3, Python DEVS has the capability of visual modelling and model transformation, which means that you can draw DEVS model visually, and transform an DEVS model to models represented in other model formalisms or vice versa. More information about Python DEVS visual modelling and model transformation can be found at *http://moncs.cs.mcgill.ca/MSDL/research/projects/DEVS/*.

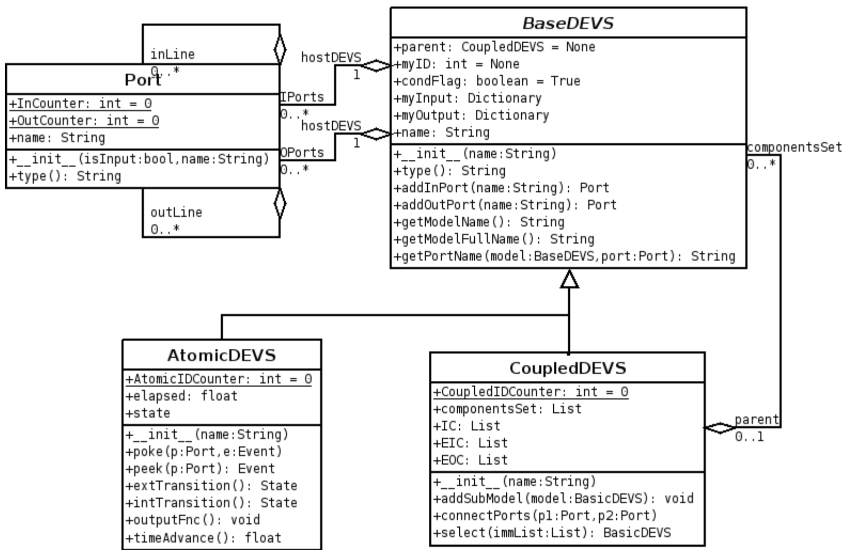


Figure 1.4: Prototypes of Python DEVS Models

Figure 1.4 is the class digram of the abstract model classes used in Python DEVS [3]. From this figure, we can see that `BaseDEVS` is the root of the inheritance class tree, which provides the functionalities that are common to both atomic DEVS and coupled DEVS. Besides the DEVS models, Python DEVS explicitly models a `Port` class. The reason for modelling the `Port` class is that ports cannot be neglected when you build a DEVS model especially when you want to couple models together, though ports are not explicitly described in Zeigler's DEVS specification.

The meanings of most of the attributes and the purpose of the functions are obvious. One thing that needs to be mentioned is the function of `getModelFullName`. In Python DEVS, each model instance has a name, and the name is given at the time when the model instance is created. This model name is purely a string and it can be invoked by using the function `getModelName`. After models are coupled together, the new created coupled model has a name, and the models that are coupled also have their own names. The full model name reflects the coupling relation of model names.

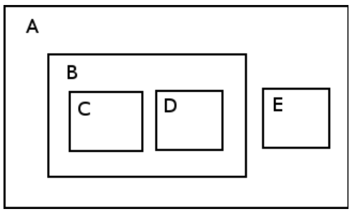


Figure 1.5: Fully Qualified Name in Python DEVS

Figure 1.5 is an example of a coupled DEVS model in which model `B` is created by coupling models `C` and `D`, and model `A` is built by coupling models `B` and `E`. In this figure, `C.getModelName()` returns ``C” and `C.getModelFullName()` returns ``A.B.C”. Here ``C” is the model's name and ``A.B.C” is the model's fully qualified name.

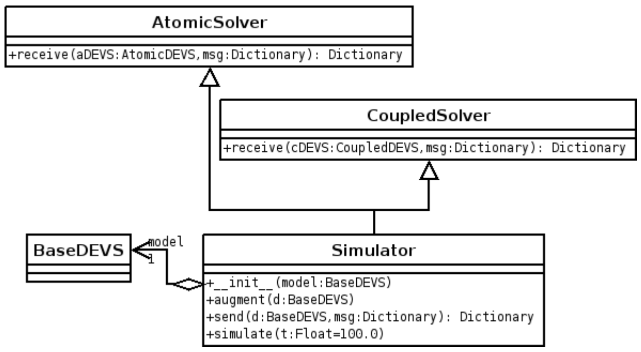

AtomicSolver
+receive(aDEVS:AtomicDEVS,msg:Dictionary): Dictionary
CoupledSolver
+receive(cDEVS:CoupledDEVS,msg:Dictionary): Dictionary
BaseDEVS
model
1
Simulator
+__init__(model:BaseDEVS)
+augment(d:BaseDEVS)
+send(d:BaseDEVS,msg:Dictionary): Dictionary
+simulate(t:Float=100.0)

Figure 1.6: Class Diagram of the Python DEVS Simulator

Figure 1.6 shows the structure of the Python DEVS simulator. The simulator architecture is simple. Here, the `AtomicSolver` has the same meaning with the concept of atomic simulator we mentioned above, and `CoupledSolver` is an alias of the coupled coordinator mentioned earlier, and the simulator is just the root coordinator. The `AtomicSolver` and `CoupledSolver` have only one function `receive`. In this function, all the messages have been processed and routed to their right destinations. The purpose of the `augment` function in the simulator is to add time related attributes to DEVS models. This is a feature of Python, in which class instances can be augmented at run time.

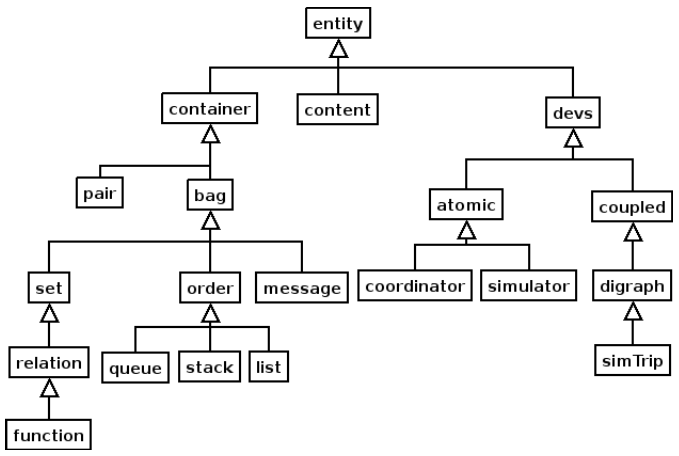


Figure 1.7: The Inheritance Diagram of DEVSJava

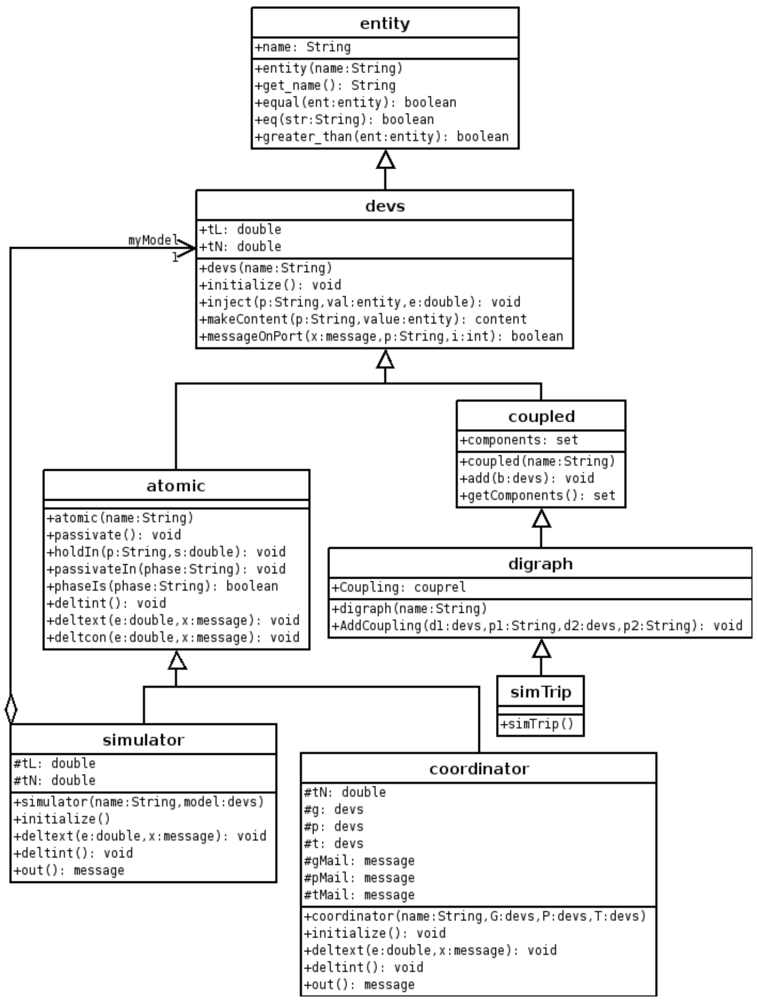


Figure 1.8: Class Diagram of DEVS Java

1.4.2 DEVSJava

DEVSJava is a DEVS modelling and simulation environment developed by Hessam Sarjoughian and Bernard Zeigler at the University of Arizona. DEVSJava supports parallel model executions on a uni-processor. Models in DEVSJava can also be readily mapped to DEVS/HLA and DEVS/CORBA for distributed execution in combined logical/realtime settings.

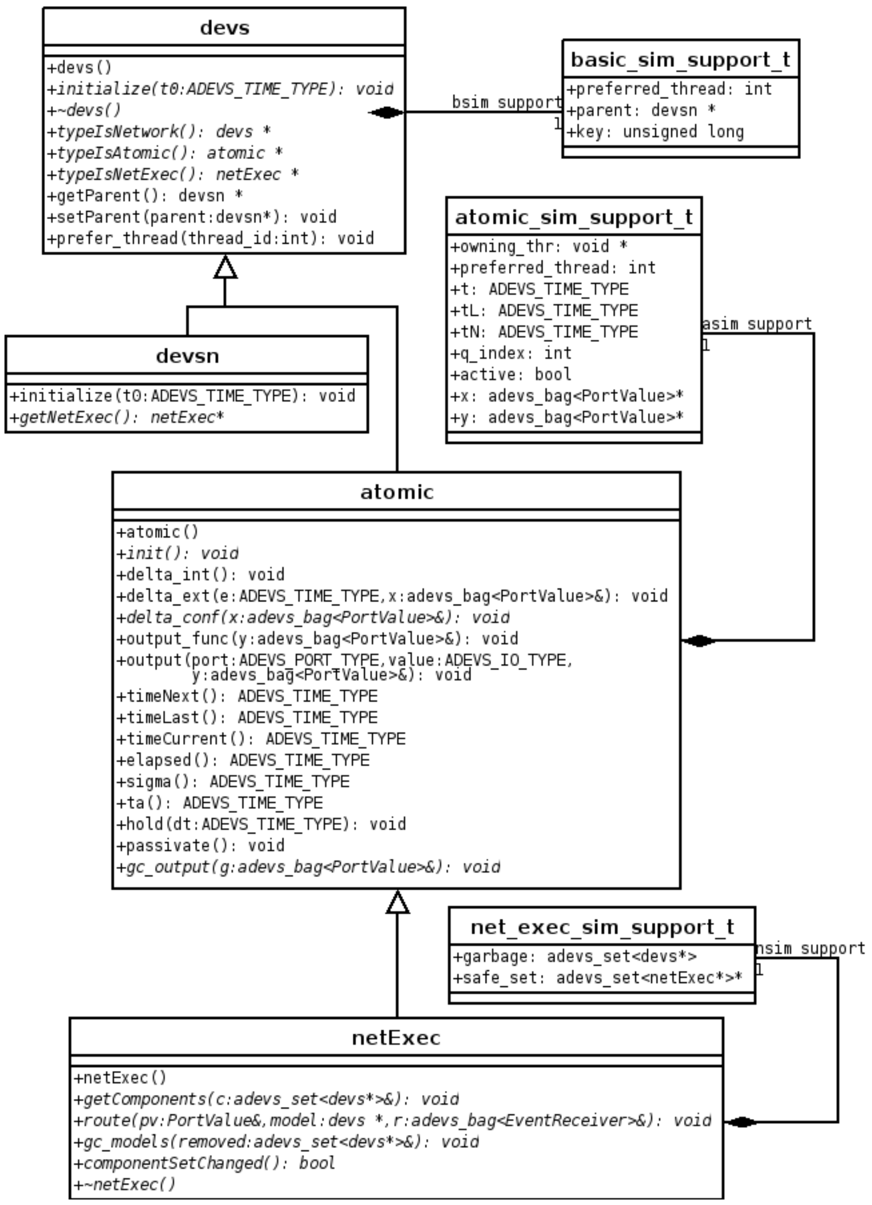

devs
+devs()
+initialize(t0:ADEVS_TIME_TYPE): void
+~devs()
+typeIsNetwork(): devs *
+typeIsAtomic(): atomic *
+typeIsNetExec(): netExec *
+getParent(): devsn *
+setParent(parent:devsn*): void
+prefer_thread(thread_id:int): void
bsim support
1
basic_sim_support_t
+preferred_thread: int
+parent: devsn *
+key: unsigned long
atomic_sim_support_t
+owning_thr: void *
+preferred_thread: int
+t: ADEVS_TIME_TYPE
+tL: ADEVS_TIME_TYPE
+tN: ADEVS_TIME_TYPE
+q_index: int
+active: bool
+x: adevs_bag<PortValue>*
+y: adevs_bag<PortValue>*
asim support
1
devsn
+initialize(t0:ADEVS_TIME_TYPE): void
+getNetExec(): netExec*
atomic
+atomic()
+init(): void
+delta_int(): void
+delta_ext(e:ADEVS_TIME_TYPE,x:adevs_bag<PortValue>&): void
+delta_conf(x:adevs_bag<PortValue>&): void
+output_func(y:adevs_bag<PortValue>&): void
+output(port:ADEVS_PORT_TYPE,value:ADEVS_IO_TYPE, y:adevs_bag<PortValue>&): void
+timeNext(): ADEVS_TIME_TYPE
+timeLast(): ADEVS_TIME_TYPE
+timeCurrent(): ADEVS_TIME_TYPE
+elapsed(): ADEVS_TIME_TYPE
+sigma(): ADEVS_TIME_TYPE
+ta(): ADEVS_TIME_TYPE
+hold(dt:ADEVS_TIME_TYPE): void
+passivate(): void
+gc_output(g:adevs_bag<PortValue>&): void
net_exec_sim_support_t
+garbage: adevs_set<devs*>
+safe_set: adevs_set<netExec*>*
nsim support
1
netExec
+netExec()
+getComponents(c:adevs_set<devs*>&): void
+route(pv:PortValue&,model:devs *,r:adevs_bag<EventReceiver>&): void
+gc_models(removed:adevs_set<devs*>&): void
+componentSetChanged(): bool
+~netExec()

Figure 1.9: Class Diagram of ADEVS

Figure 1.7 is the class inheritance diagram for DEVSJava. In this diagram we can see all the elements in DEVSJava. Besides the classes related to DEVS, it defines some common data structures for specifying algorithms used in DEVS modelling. The `simTrip` on the right bottom corner of the figure plays the role of the root coordinator, which starts a simulation process.

Figure 1.8 is the class diagram for the abstract models and the simulators of DEVSJava. An interesting feature of DEVSJava is that the simulator and coordinator have the same interfaces as an atomic DEVSJava model. This means that the simulator of DEVSJava itself is modeled in DEVS!

1.4.3 ADEVS

ADEVS is a DEVS modelling and simulation library developed by James J. Nutaro at the University of Arizona. ADEVS was developed in C++, and it supports both Parallel DEVS and Dynamic Structure DEVS (DSDEVS). Figure 1.9 is the class diagram of the structures of ADEVS's abstract DEVS models and simulators. These diagrams are drawn based on the source code of ADEVS at *http://www.ece.arizona.edu/˜nutaro/index.php*.

1.5 Conclusions

As a possible candidate for the standard of Discrete Event System modelling and simulation, DEVS provides not only specifications for building models of Discrete Event Systems, but also suggestions of building DEVS simulators. In this chapter, we first reviewed the DEVS specification. In the review, we followed the historical order of DEVS evolution, from Classic DEVS to Parallel DEVS. The major difference between Classic DEVS and Parallel DEVS is the mechanism for processing conflicting state transitions. In Classic DEVS, how to process simultaneous external events is not specified. External events have a higher priority than simultaneous internal events. Simultaneous internal events in a coupled model are processed by the order determined by the coupled model's *select* function. In Parallel DEVS, simultaneous external events are saved in a model's bag data structure. The conflict between simultaneous external events and internal events is resolved by the confluent function at the atomic model level. Because models are simulated in parallel at atomic model level, there is no internal state transition conflict in Parallel DEVS.

After reviewing the DEVS specification, we looked into algorithms for building DEVS simulators. Corresponding to DEVS model structure, DEVS simualtor has two data structures, simulator and coordinator. Simulators provide mechanisms for simulating atomic DEVS models, whereas coordinators have facilites for simulating coupled DEVS models. The root coordinator provides time progress information to all its subordinate coordinators and simulators.

Based on the same DEVS specification, many different DEVS modelling and simulation environments have been implemented. In this chapter, we briefly introduced Python DEVS, DEVSJava, and ADEVS. Python DEVS supports Classic DEVS. DEVSJava supports Parallel DEVS. ADEVS supports both Parallel DEVS and DSDEVS. Besides the functionalities, the structures of the different implementations are also different. To show the structural differences among different implementations, we presented the class diagrams of the different implementations.

Now, we have reviewed the DEVS specification and some existing implementations. In the next few chapters, we will discuss the design and implementation of our Infrastructure of DEVS Modelling and Experimentation, the main topic of this thesis.

2
Architecture and Design

2.1 Introduction

The reality that there are many available DEVS modelling and simulation envionments with different features significantly inhibits application of DEVS ([Zei05], [BV02], [Nut05]). Morever, due to the fact that the complex systems that need to be modelled are usually not trivial for most common users. DEVS, with its theoretical foundation, is not easy to grasp.

The main goal of this thesis research is to facilitate DEVS modelling and simulation, to promote DEVS standardization and application. The ultimate goal of DEVS modelling and simulation is to build a correct model for a system under study. The process of building a correct DEVS model usually includes the following phases, Modelling, Verification, Simulation, and Validation. So our question becomes how we can best support each step of the process.

Recently, visual modelling technology has become more and more popular in both software engineering and modelling and simulation industry. By representing models as graphs, visual modelling makes complex models easy to build, understood, and communicated. Many visual modelling languages, such as the Unified Modelling Laguange (UML, which includes both visual and textual notations), Entity Relationship Diagram(ERD), and Object Role Modelling(ORM), have been developed and applied in different domains.

DEVS is a formalism suitable for modular and hierarhical system modelling. So DEVS models are naturally represented graphically. Some beneficial research has been done on visual DEVS modelling. [PBV03] presents ideas on generating visual DEVS modelling environments using Meta-modelling technology. [PP93] proposes using Digraphs to visualize DEVS model. Though both of these have some limitations, they give helpful inspirations for visual DEVS modelling. We will enhance and improve these ideas to build a visual DEVS modelling environment to facilitate DEVS model creation and structural analysis at the Modelling stage.

When it comes to model verification, the first question is how models are represented. Some forms of representations are easy to be verified, while others are not. At present, almost in all available DEVS modelling and simulation environments, models are represented in a programming language. Models are represented in Java for DEVSJava, C++ for ADEVS, Python for Python DEVS, and so forth. Since the verification process usually needs some model structure information, programming languages are not a good choice of model representation language for verification purposes.

Modelling languages (also called model description language) are high-level declarative languages specificaly designed for hight-level abstract model representation. Recently, many modelling languages have been developed and applied successfully in different industries. The Specification Description Language (SDL) has been used in telecommunication industry for describing communication protocols; UML has been used by software engineers to specify software and business models; Modelica has been used in the modelling and simulation world for specifying models of physical systems. These success stories give us some indication that representing DEVS models in a high-level modelling language is a better choice for the purpose of model verification.

High-level modelling language emphasize on specifying models in a precise and concise way. Models represented in modelling languages usually cannot be run or simulated directly. A common pratice is that in the Modelling phase, models are represented in modelling languages. When it comes to simulation, high-level modelling representations are transfomed into models represented in programming languages, which are suitable for model execution or simulation. The transformation process is usually done by a model compiler. The model compiler usually at least has the following three capabilities. First, it checks whether the models are syntactically correct for a specific modelling language. Second, it verifies whether the models are structurally correct models for a certain formalism. And finally, it transforms the models to representations in a target programming language.

The model compiler can verify that a model is syntactically correct against a specific modelling language, and structurally correct against a certain modelling formalism. However, it cannot give one the confidence that the model reflects the behaviour of a real system correctly. Simulation is a process of running models with experimental data in a virtual environment to emulate the behaviour of a real system. By feeding the models with certain experimental input parameters, the virtual simulation environment can execute the models to generate output data. The output data combined with the simulation time gives the simulation trace. The behaviour of a model is reflected by the simulation trace. Comparing the simulation trace with experimental results obtained through experiments on the real system, a modeller can validate whether a model reflects the behaviour of the real system correctly.

From the discussion above, we can see that model validation is basically done by analyzing the simulation trace. In reality, the simulation trace of the models of a complex system is usually huge, so the task of model validation is not trivial. Two issues need to be considered for relieving the burden of model validation. First, representing the simulation result in an appropriate format. Second, developing tools to facilitate the validation process.

The Extensible Markup Language (XML) was originally designed for representing data for large-scale electronic publishing. Now it plays an increasingly important role for data exchange on the web and in other areas. In the software modelling industry, the Object Mangagement Group (OMG) has defined the standard XML format (XML Metadata Interchange) for exchanging meta-infomation for UML implementations of different vendors. In the web representation field, HTML follows the standard XML grammar since version 4.0. In the application domain, many applications, such as Dia, Inkscape, Eclipse, and so forth, save the graph files or meta-data in XML format. The major feature of representing data in XML format is that the structure of a XML file can be specified by a Document Type Definition(DTD) or an XML Schema. Based on the DTD or XML Schema, an XML file can be easily validated by an XML validator and transformed from one format to another using the eXtensible Stylesheet Language Transformations (XSLT). These features of XML make it a good candidate for representing DEVS simulation trace.

Though XML is flexible and easy to be validated, an XML representation of the simulation trace is still textual. For human consumption, a graphical data representation is much more expressive than a textual one. Trace visualization is the process of transforming textual information into graphical representation. Some research has been done on trace visualization for facilitating the trace analysis. [ST00] proposes ideas on building trace visualization and analysis tools for sypervisory control systems. [AHW05] presents proposals on trace visualization for interactive performance analysis of complex systems. These give us confidence that trace visualization is a practical and reasonable way for DEVS trace analysis.

In this chapter, we discuss the design and architecture of the DEVS modelling and simulation infrastructure. As we discussed above, the infrastrusture is designed to facilitate DEVS modelling and simulation in the following aspects. At the Modelling level, we use visual modelling technology to build a DEVS visual modelling environment. At the Verification stage, we represent DEVS models in a neutral modelling language, and use model compiler to verify the neutrally represented DEVS models and transform that neutral representation into a language-specific model representation. At the Simulation step, we use a standardized XML format to represent DEVS simulation traces. And finally, at the Validation phase, we build a DEVS visual trace plotter for facilitating the DEVS trace validation process.

The organization of this chapter is as followings. Firstly, we introduce some background concepts that we use to design and architect the system. Secondly, we discuss the motivations and purposes for designing and implementing the infrastructure. Thirdly, we present the architecture of the infrastructure, and give explanations for the choices we make at each step. And finally, we draw some conclusions for this chapter.

2.2 Modelling Language Concepts

The following describes the various aspects of modelling languages. In particular, a precise terminology is introduced. This section is based on [VdL03].

2.2.1 Modelling

Models are abstractions of reality. The structure and behaviour of systems we wish to analyze or design can be represented by models. These models, at various levels of abstraction, are always desribed in some formalism or modelling language. In addition to the syntax of a model (how it is represented), one needs to also specify its meaning (i.e., assign semantics).

One can, for example, specify on the one hand how a system dynamically evolves over time. On the other hand, it is possible to concentrate purely on the static structure of the system, without specifying its dynamic transitions between states. This demonstrates how, depending on the circumstances, one has to choose the right modelling abstraction.

So, after we get an abstraction from the reality, we have two questions that must be answered. The first is how we describe our models. And the second is how we can demonstrate that our model is right. For the first question, models are usually described by a model description language. The solution for the second is often by simulation.

2.2.2 Modelling Language

Models can be described graphically using visual modelling languages or textually with textual modelling languages. To "model" modelling languages and ultimately synthesize visual modelling environments for those languages, we will break down a modelling language into its basic constituents. This is illustrated in Figure 2.1. It is inspired by the description by Harel and Rumpe[HR00].

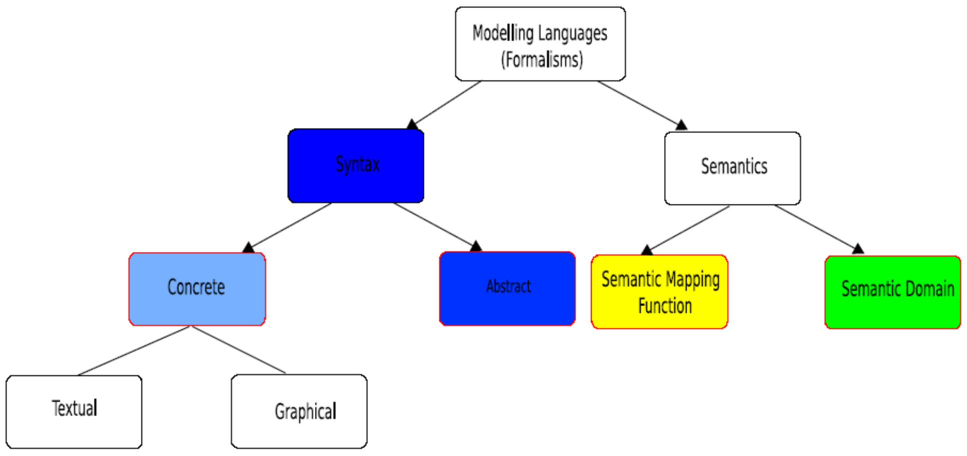


Figure 2.1: Modelling Language Breakdown

No matter which language you use to describe it, two main aspects of a model are its syntax (how it is represented) on the one hand and its semantics (what it means) on the other hand.

The syntax of modelling languages is traditionally partitioned into concrete syntax and abstract syntax. In textual languages for example, the concrete syntax is made up of sequences of characters taken from an alphabet. These characters are typically grouped into words or tokens. Certain sequences of words or sentences are considered valid (i.e., belong to the language). The (possibly infinite) set of all valid sentences is said to make up the language. Costagliola et. al. [Costagliola, Lucia, Orefice, and Polese 2002] present a framework of visual language classes in which the analogy between textual and visual characters, words, and sentences becomes apparent. Visual languages are those languages whose concrete syntax is visual (graphical, geometrical, topological, . . .) as opposed to textual.

For practical reasons, models are often stripped of irrelevant concrete syntax information during syntax checking. This results in an "abstract" representation which captures the "essence" of the model. This is called the abstract syntax. Obviously, a single abstract syntax may be represented using multiple concrete syntaxes. In programming language compilers, abstract syntax of models (due to the nature of programs) is typically represented in Abstract Syntax Trees (ASTs). In the context of general modelling, where models are often graph-like, this representation can be generalized to Abstract Syntax Graphs (ASGs).

Once the syntactic correctness of a model has been established, its meaning must be specified. This meaning must be unique and precise. Meaning can be expressed by specifying a semantic mapping function which maps every model in a language onto an element in a semantic domain. For example, the meaning of a Causal Block Diagram is given by mapping it onto an Ordinary Differential Equation. For practical reasons, semantic mapping is usually applied to the abstract rather than to the concrete syntax of a model. Note that the semantic domain is a modelling language in its own right which needs to be properly modelled (and so on, recursively). In practice, the semantic mapping function maps abstract syntax onto abstract syntax.

2.2.3 Meta-modelling

Meta-modelling is a heavily over-used term. Here, we will use it to denote the explicit description (in the form of a model in an appropriate meta-modelling language) of the abstract syntax set. Often, meta-modelling also covers a model of the concrete syntax. Semantics is however not covered. On the one hand, a meta-model can be used to check whether a general model (a graph) belongs to the abstract syntax set. On the other hand, one could, at least in principle, use a meta-model to generate all elements of the syntax set. This explains why the term meta-model and grammar are often used inter-changeably.

Several languages are suitable to describe meta-models. Two approaches are in common use:

1. A meta-model is a type-graph. Elements of the language described by the meta-model are instance graphs. There must be a morphism between an instance-graph (model) and a type-graph (meta-model) for the model to be in the language. Commonly used meta-modelling languages are Entity Relationship Diagrams (ERDs) and Class Diagrams (adding inheritance to ERDs). The expressive power of this approach is often not sufficient and an extra constraint language (such as the Object Constraint Language in the UML) specifying constraints over instances is used to further specify the set of models in a language. This is the approach used by the OMG to specify the abstract syntax of the Unified Modelling Language (UML).
2. A more general approach specifies a meta-model as a transformation (in an appropriate formalism such as Graph Grammars) which, when applied to a model, verifies its mem-

bership of a formalism by reduction. This is similar to the syntax checking based on (context-free) grammars used in programming language compiler compilers. Note how this approach can be used to model type inferencing and other more sophisticated checks.

Both types of meta-models (type-graph or grammar) can be interpreted (for flexibility and dynamic modification) or compiled (for performance).

Note that when meta-modelling is used to synthesize interactive, possibly visual modelling environments, we need to model when to check whether a model belongs to a language. In free-hand modelling, checking is only done when explicitly requested. This means that it is possible to create, during modelling, syntactically incorrect models. In syntax-directed modelling, syntactic constraints are enforced at all times during editing to prevent a user from creating syntactically incorrect models. Note how the latter approach, though possibly more efficient, due to its incremental nature - of construction and consequently of checking - may render certain valid models in the modelling language unreachable through incremental construction. Typically, syntax-directed modelling environments will be able to give suggestions to modellers whenever choices with a finite number of options present themselves.

The advantages of meta-modelling are numerous. Firstly, an explicit model of a modelling language can serve as documentation and as specification. Such a specification can be the basis for the analysis of properties of models in the language. From the meta-model, a modelling environment may be automatically generated. The flexibility of the approach is tremendous: new languages can be designed by simply modifying parts of a meta-model. As this modification is explicitly applied to models, the relationship between different variants of a modelling language is apparent. Above all, with an appropriate meta-modelling tool, modifying a meta-model and subsequently generating a possibly visual modelling tool is orders of magnitude faster than developing such a tool by hand. The tool synthesis is repeatable and less error-prone than hand-crafting.

As a meta-model is a model in an appropriate modelling language in its own right, one should be able to meta-model that language's abstract syntax too. Such a model of a meta-modelling language is called a meta-meta-model. It is noted that the notion of "meta-" is relative. In principle, one could continue the meta- hierarchy ad infinitum. Luckily, some modelling languages can be meta-modelled by means of a model in the language itself. This is called meta-circularity and it allows modelling tool and language compiler builders to bootstrap their systems.

A model m in the Abstract Syntax set needs at least one concrete syntax. This implies that a concrete syntax mapping function mapping an abstract syntax graph onto a concrete syntax model is needed. Such a model could be textual (e.g., an element of the set of all Strings), or visual (e.g., an element of the set of all the 2D vector drawings). Note that the set of concrete models can be modelled in its own right. It is noted that grammars may be used to model a visual concrete syntax. Also, concrete syntax sets will typically be re-used for different languages. Often, multiple concrete syntaxes will be defined for a single abstract syntax, depending on the user. If exchange between modelling tools is intended, an XML-based textual syntax is often used. If in such an exchange, space and performance is an issue, a binary format may be preferrable. When the formalism is graph-like as in the case of a circuit diagram, a visual concrete syntax is often used for human consumption. The concrete syntax of complex languages is however rarely entirely visual. When for example equations need to be represented, a textual concrete syntax is more appropriate.

Finally, a model m in the Abstract Syntax set needs a unique and precise meaning. As previously discussed, this is achieved by providing a Semantic Domain and a semantic mapping function. This mapping can be given informally in English, pragmatically with code or formally with model transformations. Natural languages are highly ambiguous and not very useful since they cannot be executed. Code is executable, but it is often hard to understand, analyze and maintain. It can be very hard to understand, manage and derive properties from code. This is why formalisms such as Graph Grammars are often used to specify semantic mapping functions in particular and model transformations in general. Graph Grammars are a visual formalism for specifying transformations. Graph Grammars are formally defined and at a higher level than code. Complex behaviour can be expressed very intuitively with a few graphical rules. Furthermore, Graph Grammar models can be analyzed and executed. As efficient execution may be an issue, Graph Grammars can often be seen as an executable specification for manual coding. As such, they can be used to automatically generate transformation unit tests.

2.2.4 Simulation

A model is an abstraction of a real system. We can ascertain that a model accurately reflects the structure and behaviour of the system it represents by means of simulation. Simulation of a model described in a certain formalism (such as Petri Net, Differential Algebraic Equations (DAE) or Bond Graph) produces simulation results: the dynamic input/output behaviour. Simulation may use symbolic as well as numerical techniques. Simulation, which mimics the real-world experiment, can be seen as virtual experimentation, allowing one to answer questions about (the behaviour of) a system. As such, the particular technique used does not matter. Whereas the goal of modelling is to meaningfully describe a system presenting information in an understandable, re-usable way, the aim of simulation is to be fast and accurate. Symbolic techniques are often favoured over numerical ones as they allow the generation of classes of solutions rather than just a single one. For example, $Asin(x) + Bcos(x)$ as a symbolic solution to the harmonic equation $\frac{d^2x}{d^2y}$ (with A and B determined by the initial conditions) is preferred over one single approximate trajectory solution obtained through numerical simulation. Furthermore, symbolic optimizations have a much larger impact than numerical ones thanks to their global nature. Crucial to the SystemExperiment/ModelVirtual Experiment scheme is that there is a homomorphic relation between model and system: building a model of a real system and subsequently simulating its behaviour should yield the same results as performing a real experiment followed by observation and codifying the experimental results. A simulation model is a tool for achieving a goal (design, analysis, control, optimisation, . . .). A fundamental prerequisite is therefore some assurance that inferences drawn from modelling and simulation (tools) can be accepted with confidence. The establishment of this confidence is associated with two distinct activities; namely, verication and validation.

2.2.5 Verification

There are two different aspects of meanings for the word verification in the modelling community. For a model represented in a specific modelling formalism, model verification means the process of checking whether a model satisfies the requirements of the formalism. To be specific, check whether a model has certain structrual features specified by the formlism. For model transformations, verification is the process of checking the consistency of a simulation program with respect to the lumped model it is derived from. More explicitly, verification is concerned with the correctness of the transformation from some intermediate abstract representation (the conceptual model) to the program code (the simulation model) ensuring that the program code faithfully reflects the behaviour that is implicit in the specification of the conceptual model. A model compiler may automate the transformation from conceptual model to simulation model (code). If this compiler can be verified, all transformations by the compiler are verified.

2.2.6 Validation

Validation is the process of comparing experimental measurements with simulation results within the context of a certain Experimental Frame[ZPK00]. When comparison shows differences, the formal model built may not correspond to the real system. A large number of matching measurements and simulation results, though increasing confidence, does not prove validity of the model however. A single mismatch between measurements and simulation results invalidates the model. For this reason, Popper has introduced the concept of falsification, the enterprise of trying to falsify or disprove a model. A model may state that water in a pot boils at 100^oC. Many experiments will confirm this model, until either the pot is closed or is taken to a different altitude. A falsified model should not lead to an outright rejection of the model. Rather, it should lead to a refinement of the model's experimental frame on the one hand and to an attempt to expand the model to the current experimental frame. In the case of the water boiling in a pot, a restricted experimental frame would state that the pressure must be constant (1atm). Expanding the model would express the relationship between boiling point and pressure and volume.

2.2.7 Modelling and Simulation Process

Modelling and simulation is an iterative process. One cannot build a correct model for a complex system in one go. Based on the concepts introduced above, a general process for modelling and simulaltion can be described as shown in Figure 2.2. This process includes four steps: `Modelling`, `Verification`, `Simulation`, and `Validation`. The process of creating a correct model for a real system is an iterative process of repeating these four steps.

In the `Modelling` step, the modeler analyzes the system in question. Based on the features of an appropriate interest and the nature of the problem, the modeler builds a tentative model using specific model formalism for the system. The tentative model can be expressed either in model description languages or in programming languages. Because this is a tentative model, it cannot be guaranteed correct both in syntax and in semantics.

In the `Verification` phase, the verification model takes the tentative model as input. Based on the features of the specific formalism, it checks whether the tentative model meets the syntax requirements of the fomalism used to build this model. If the model passes the verification, it will be forwarded to the next phase. The modeler may have to analyze the real system and the feedback from the verification to refine the tentative model until it passed the syntax verification.

At the `Simulation` level, a formalism specific simulator takes the tentative model that passes syntax verification as input and generate the simulation trace that reflects the behaviour of the model. The generated trace can be represented in some structured text file, or just a collection of data with specific type in memory.

At `Validation` time, the generated simulation trace is analyzed. The job of trace analysis can be done automatically by some software, or semi-automatically by some analysis facilitating tools, or totally by hand. By comparing the analysis result with data obained by observing the real system, the modeler can determine whether the tentative model is as desired. If the analysis results match the experimental or observed data, we get the model we want. The modeler may have to go back to the analysis phase to analyze the real system again and to refine the tentative model, until a validated model is obtained.

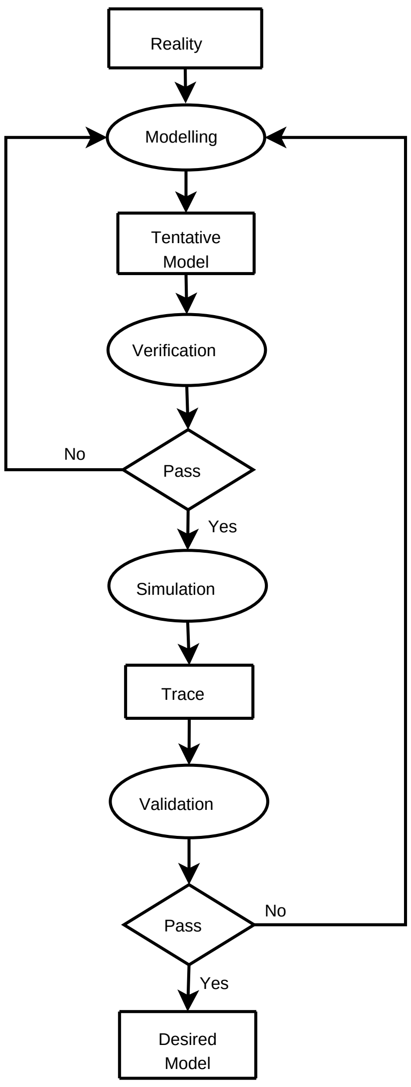


Figure 2.2: Modelling and Simulation Process

This four-step process, `Modelling`, `Verification`, `Simulation`, and `Validation`, is just a high-level description of the modelling and simulation procedure. How it works concretely depends on the modeler's preference and how the modelling and simulation tools are built.

2.3 Motivation and Purpose

In chapter 1, we used DEVS Java, Python DEVS, and ADEVS as examples to discuss the features of currently existing DEVS modelling and simulation environments. From that discussion, we can see that these existing systems share some common features from the four-step modelling and simulation process. Firstly, at the Modelling level, there are no tools available to facilitate the model creation. Secondly, at the model representation and verification level, models are represented in specific programming languages and the verification process is usually done by hand. Thirdly, at the simulation and validation level, simulation traces are represented in pure text format and no tools are available to facilitate trace analysis. The modelling and simulation process for most current existing DEVS modelling and simulation environments can be shown as in Figure 2.3.

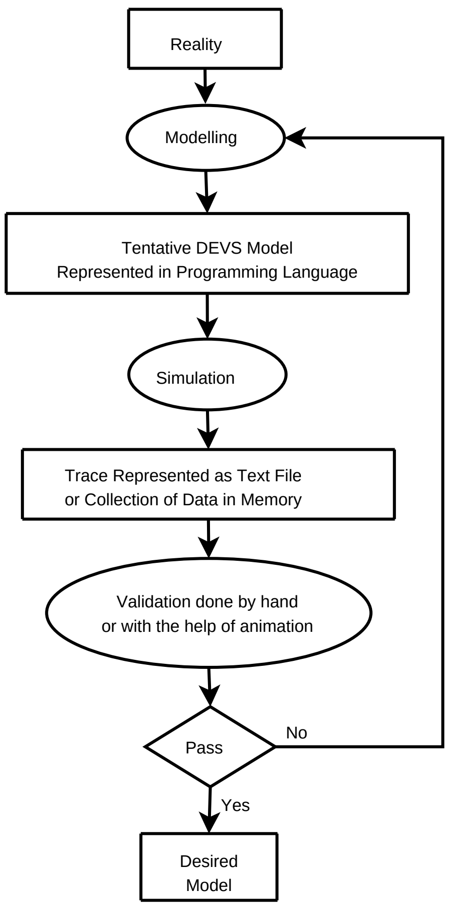

Reality
Modelling
Tentative DEVS Model
Represented in Programming Language
Simulation
Trace Represented as Text File
or Collection of Data in Memory
Validation done by hand
or with the help of animation
Pass
No
Yes
Desired
Model

Figure 2.3: Current General Practice of DEVS Modelling and Simulation Process

From the introduction of chapter 1, we can see that DEVS specifies a two-level structure for Discrete Event Sytem modelling and simulation. On the one hand, DEVS specifies the syntax and semantics of DEVS models; on the other hand, DEVS gives the algorithms and suggested structures for building DEVS simulators. The major benefit of DEVS specification is that it clearly separates the task of model creation from the task of building a DEVS simulator.

DEVS modellers concentrate on specifying models, while simulator builders focus on improving simulators' performance. The interfaces beween DEVS models and simulators are very clear. If a model is created following the DEVS specification, it should be possible to simulate it by means of any standard DEVS simulator. However, in reality, at least at this time, things are not that simple. Models written for DEVS Java cannot be simulated by the Python DEVS simulator. Models written for ADEVS are specific to ADEVS. Though those models are all DEVS models, the fact that DEVS simulators cannot simulate DEVS models (written for other simulators) is certainly not the intention of the DEVS specification.

There are many reasons causing the incompatibility among DEVS models and simulators. One major reason is the practice that DEVS models are represented in progamming languages. In most of the current existing DEVS modelling and simulation environments, models are represented in the language in which the simulator is coded. For example, models in Python DEVS are described in Python code, models for DEVSJava are represented in Java code, and models in ADEVS are specified by C++ code. Though programming language represented models may be more efficiently and smoothly simulated by the language-specific simulator. This approach has many drawbacks as explained below.

1. It blurs the boundary of labour division. Generally, modellers are domain experts who have a rich knowledge in specific domains. The main purpose of using DEVS for a modeller is to solve problems in his/her domain. One should not expect that domain experts are also computer programming professionals. On the other hand, simulator builders are computer scientists (or programmers) who are good at making simulators that can run efficiently and correctly. The simulator builders are not necessarily domain experts. Because DEVS models are represented in programming languages, the modellers (or DEVS users) first must be programmers, or they even do not know how to write a model, not to mention a correct DEVS model.
2. It makes model reuse difficult. DEVS is a formalism for high-level model description. DEVS models have nothing to do with computing implementation details. While programming languages are good at specifying execution and implementation details to make a program run effeciently and correctly, one must provide extra details or enforce extra constraints on a DEVS model in order to represent it in programming languages. High-level of model descripton is usually easier to reuse than low-level model specification with implementation details. For example, in Python DEVS, atomic DEVS models are defined as Python classes that inherit from the predefined Python class `AtomicDEVS`. From a software design and implementation perspective, this is a good solution. However, when it comes to model reuse, things become complex. One's original intention is to reuse a DEVS model, but now one has to consider not only how to reuse a DEVS model, but also how to reuse a Python class that inherits from another Python class `AtomicDEVS`.
3. It makes DEVS standardization more difficult. This statement is meaningful at two levels. First, representing DEVS models in programming languages lowers the high-level DEVS specification to programming language level with implementation details. The more details involved, the more difficult it is to standardize DEVS. Second, programming languages are similar at almost the same abstract representation level. Each language has its own specific features. It is hard to say which programming language is better for DEVS model representation. One cannot convince others that which programming language should be used as standard for DEVS model representation. This situation creates the deadlock today that DEVS simulators (in one language) cannot simulate DEVS

models (represented in another programming language). All existing DEVS modelling and simulation environments are built based on DEVS specification, they are all correct in their own regimes. No one can cross the programming language boundary.

4. It makes automatic model verification difficult. As we mentioned earlier, model verification is the process of checking whether a model satisfies the DEVS specification. It is much harder to get a model's structure information at the programming language level than at the model description level. Though in some languages like Java, this can be done through the reflection mechanism, the price is high. For other programming languages like C++, it is almost impossible.

These disadvantages of representing DEVS models in programming languages have severely impeded DEVS expansion and application. In order to facilitate the process of DEVS modelling and simulation, and to promote DEVS standardization and application, we are motivated to build an infrastructure for DEVS modelling and simulation. We hope that our efforts can solve or partially solve the problems we have in DEVS modelling and simulation today, and open more chances for future research.

The infrastructure concenstrates on issues in the following areas. Firstly, at the modelling stage, we build visual modelling tools to facilitate the process of DEVS model creation. Secondly, at the model representation and verification level, we represent DEVS models in high-level model description language, and use the model comiler to verify DEVS models automatically. Thirdly, at the simulation level, we represent simulation trace in standard XML documents. And finally, at the model validation level, we build visual trace plotting tools to facilitate the process of trace analysis.

2.4 The Overall Architecture

Figure 2.4 gives the overall architecture of the infrastructure for DEVS modelling and simulation. Corresponding to the four-step modelling and simulation process, we build tools or add functionalities at each step.

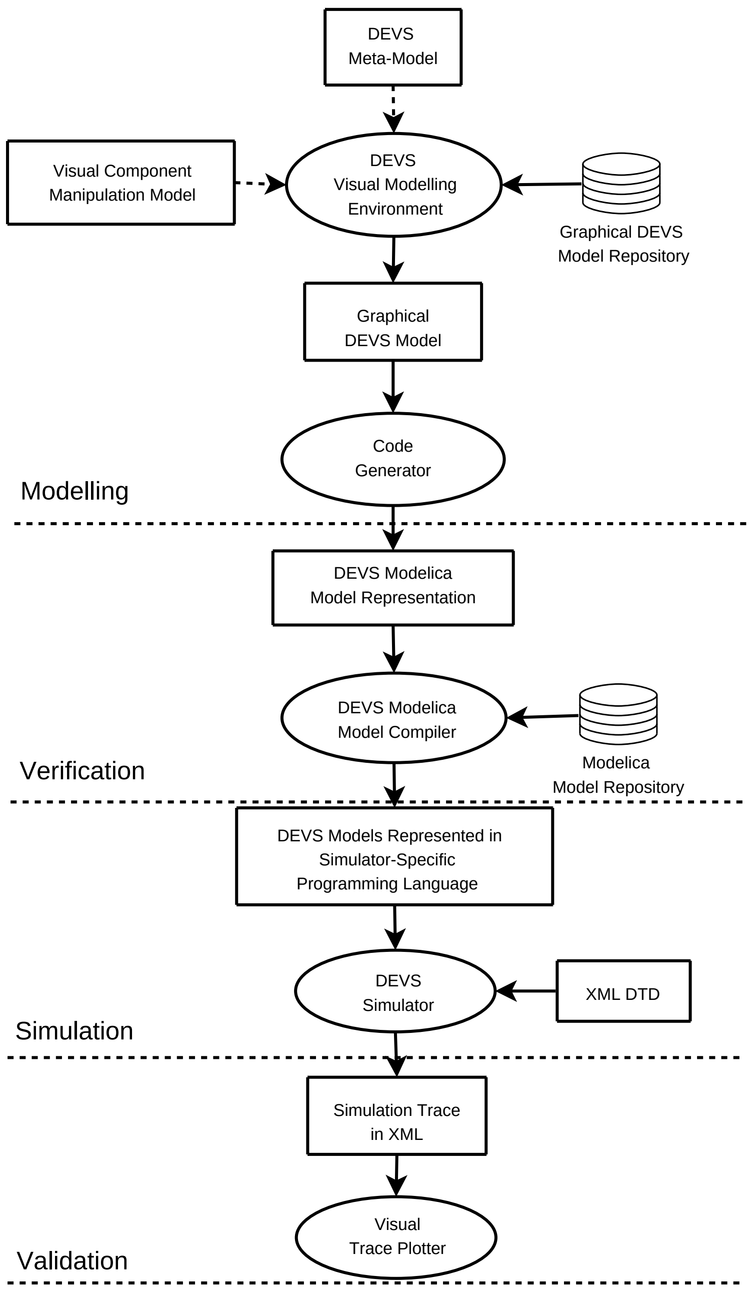

DEVS Meta-Model
Visual Component Manipulation Model
DEVS Visual Modelling Environment
Graphical DEVS Model Repository
Graphical DEVS Model
Code Generator
Modelling
DEVS Modelica Model Representation
DEVS Modelica Model Compiler
Modelica Model Repository
Verification
DEVS Models Represented in Simulator-Specific Programming Language
DEVS Simulator
XML DTD
Simulation
Simulation Trace in XML
Visual Trace Plotter
Validation

Figure 2.4: Architecture of the Infrastructure for DEVS Modelling and Simulation

Modelling At the `Modelling` stage, we provide a DEVS visual modelling environment, in which DEVS models can be created and manipulated graphically. The visual modelling environment has the following features.

1. The environment is automatically generated. Everything in the environment is modeled. Firstly, DEVS abstract syntax is meta-modelled. The DEVS meta-model specifies the properties and constraints on visual DEVS components and relations among DEVS components. So the visual modelling environment can do some model checking jobs at the visual modelling level based on the meta-model. Secondly, the operations that can be performed on each DEVS component are also modelled. This has two benefits. First, it limits the opportunities of misoperations for users. Second, it helps maintain model consistency at the visual modelling level. Because in the operation model, not only can operations on one component be specified, the actions for dealing with chained effects on other model components caused by operations on one model component can also be specified.
2. Graphical models created in the visual enviroment are transformed into the neutral Modelica representation rather than to models represented in programming languages. We have discussed in the previous section that programming languages are not a good choice for DEVS model representation. So, in the DEVS modelling and simulation infrastructure, we use the declarative object-oriented modelling language Modelica to describe DEVS models to counter the drawbacks of representing DEVS model in programming languages. Certainly, Modelica is not the only choice for neutral DEVS model representation. We choose Modelica for two major reasons. First, Modelica is a mature modelling language, it has been successfully used to describe continuous models in different domains. Second, Modelica has language features that are suitable for describing both continuous and discrete models. DEVS have been envisioned as a potential formalism for hybrid system modelling. Combining them may open more chances for future research. We counter the drawbacks of representing models in programming languages using Modelica in the following ways. First, Modelica is a declarative high-level model language. It is easier to be learned by domain experts. So modellers can concentrate on specifying models in Modelica. And simulator builders are focusing on building model compilers and model simulators. The labour division between modellers and simulator builders become clear. Second, because Modelica is a high-level model description language without implementation details, models represented in Modelica are easier to be reused and standardized than those represented in programming languages. Finally, the model compiler that compiles Modelica models into language-specific models can do the model verification automatically.
3. The visual modelling environment is automatically generated based on the DEVS meta-model, and the DEVS component manipulation model. This means that the environment is not hard coded. When the DEVS meta-model or the component manipulation model is changed, the environment can be easily updated.

Verification At the `Verification` level, a DEVS Modelica model compiler is provided. The model compiler has the following functionalities. Firstly, it makes sure that the models are syntactically correct Modelica models. Secondly, it checks that the models satisfy DEVS specification. And finally, it generates the simulator-specific DEVS model representations.

Simulation At the `Simulation` stage, simulation traces represented in XML are generated. The syntax and the meaning of the content of the XML represented simulation trace are specified an XML DTD. The benefits of using XML and DTD are obvious.

1. It defines a clear interface between DEVS simulators and trace plotting or analysis tools. Different simulators, DEVS Java, or Python DEVS, can generate simulation traces in the same format if they both follow the specification of the DTD. This opens many chances for further study of the simulation trace. First, trace plotting and analysis tool builders can concentrate on building tools without considering implementation details of different simulators. The trace representation is specified by the XML DTD. If a tool can understand the XML DTD, then all the XML files that satisfy the DTD can be plotted or analyzed by the tool. Second, it make trace comparison possible. For the same DEVS model, if it is simulated by different DEVS simulators using the same XML DTD, the simulation trace should be the same. This is significant for standardizing and certifying DEVS simulators.
2. The XML trace representation is specified by an XML DTD. So XML represented simulation trace files can be easily validated by XML validators against the DTD. Additionally, with the support of XSLT, one XML simulation trace file can be easily transformed to other formats for further study.
3. XML is a standard way of data representation. It has been widely used in many different fields. For users, the learning curve is smaller than using a private format of data representation. For tool developers, there are many standard libraries available in different progamming languages for accessing and manipulate contents of XML files, which can save both time and investment.

Validation At the `Validation` level, we provide a visual trace plotter for facilitating the process of human trace analysis. Validation is to check whether a model meets the goal it was created for. Trace analysis is one of the effective methods in validating a model since traces represent the actual flow of events in the systems [ST00]. There are many different ways for trace analysis. No matter what way one uses, and what purpose it is for, visualization tool is definitely an effective facilitator. Some people have successfully used this technique in their research. [ST00] introduces a trace visualization and analysis for supervisory control systems, in which they found the tool was effective in reducing the time for simplifying understanding of the dynamic behaviours of the system and finding problem areas. [AHW05] presents a trace visualization tool to interactively analyze the performance of parallel programs. Based on this understanding, we build a visual tace plotter to visualize DEVS model sequential state transitions and state variable changes along the simulation time. This plotter provides two different level user interfaces. For simple usage, one can simply browse the simulation trace visually by selecting interesting model properties. For complex application, one can customize the model state parser used by the plotter to reparse the raw model state information and providing customizable filtered traces.

Features of the Overall Architecture Besides the features at each stage of the four-step process, below are some special characteristics of the overall architecture.

1. Two-level of model reuse. From Figure 2.4, we can see that there are two model repositories, a graphical model repository and a Modelica model repository. This means that

model reuse can happen at two different levels of this architecture. This is meaningful in the following situations. First, in a complex system, some components may be suitable for graphical representation, while others are better represented in text. This way of design makes it possible for reuse of both the graphical and textual components. Second, this ensures the visual modelling environment and textual modelling envrionment can be used independently. Users can choose either one depending on their own preferences.

2. Clear boundaries between different steps. In the architecture, the boundaries between each step are very clear. The results of the `Modelling` step are Modelica DEVS representations. The results of the `Verification` are language-specific model representations. And the results of the `Simulation` step are XML represented simulation traces. The clear interfaces fetween different steps make the architecture very flexible and scalable.
3. Open structure, considering current existing DEVS modelling and simulation enviroments and opening chances for future research. This is closely related to the feature 2 above. This architecture is a open structure for DEVS modelling and simulation. It can be used combined with current existing DEVS modelling and simulation environments. But it does not depend on any language-specific DEVS simulator. Because models are represented in the high-level modelling language Modelica, with a proper model compiler, the models can be translated to any language-specific representation, and so be simulated by different simulators. Similarly, because the XML represented simulation trace is specified by the XML DTD, the trace plotter is also independent of specific simulators. Traces from different simulators can be plotted by the same plotter, and the same trace file can be plotted by different plotters with different functionalities.

2.5 Conclusions

In this chapter, we discussed the design and architecture of our infrastructure for DEVS modelling and experimentation. We first introduced some commonly used modelling and simulation concepts and gave a general description of a commonly adopted modelling and simulation process. Then we discussed the current practice of DEVS modelling and simulation, and some drawbacks of representing DEVS models in progamming languages. After that we presented the ideas of building the infrastructure for DEVS modelling and simulation. Finally, we introduced the architecture of the infrastructure and gave a simple explanation for each part of the infrastructure.

In the next few chapters, we will discuss each part of the infrastructure in more detail. Parts of the infrastrucure are presented in a bottom-up fashion. In chapter 3, we introduce the visual DEVS trace plotter. In chapter 4, we discuss the DEVS Modelica model representation and the model compiler. In chapter 5, we present the DEVS meta-model and the visual modelling environment. In chapter 6, we present a case study demonstrating the infrastructure at work.

3 Standardized Trace Representation and Trace Plotter

3.1 Introduction

Model building is an iterative process, which means that it is very rare that one can build a right model at the first time. The `Modelling`, `Verification`, `Simulation`, and `Validation` process will be executed again and again, until a desired model is obtained. So, after a model is created, the modeler must find ways to make sure that the model really reflects both the structure and behaviour of the system under study. The structure of the model can be checked by visual modelling tools or model compilers. The behaviour is checked by way of simulation and validation.

The basic idea of simulation and validation is that, by doing the model simulation, we can get a simulation trace. By analyzing the simulation trace, we can know the behaviour of the model. Due to the fact that systems modeled in DEVS are very complex, the task of analyzing simulation trace is not trivial. Research in the modelling and simulation has shown that trace visualization tools can significantly facilitate the process of trace analysis[AHW05][ST00]. Though there are many DEVS modelling environments available now, there is no DEVS-specific trace visualization tool available.

[AHW05] and [ST00] propose approaches for trace visualization for specific complex systems. They provide valuable information for DEVS trace plotting, however, as they are specific to applications, we cannot apply them directly to DEVS. DEVS is a neutral formalism for discrete event modelling and simulation. Though there are many different kinds of DEVS implementations without standardized interface, they have some features in common. First, each DEVS model has state variables to describe the model status. Second, at simulation time, all DEVS simulators are event driven, which means that a state trace record can only be generated at the time that an event occurs.When an event occurs, the state of the model usually changes accordingly. So the simulation trace of a DEVS model can be represented as a series of state values with a time stamp that marks the time the model state changes.

There are many different existing DEVS simulators and simulation traces generated by different simulators are represented differently. In order to make it possible for our trace plotter to plot traces generated by different DEVS simulators, the trace file that the plotter takes as input must be simulator independent. Based on this analysis, we decided to save the trace file in an XML format. There are many benefits of using the XML format. Firstly, XML as standardized way for data storage has been widely accepted by people both in industry and academic area. There would be less resistance and no big curve for learning. Secondly, XML can be defined by a DTD (Document Type Definition) or XML Schema, one can validate a XML file against its DTD or XML Schema. This makes it easier to verify a trace file syntactically right before plotting. Thirdly, with XSLT support, updating and evolution of a trace file are easy.

Usually, the simulation trace of a DEVS model is huge. So the task of parsing a huge XML file is not trivial. In order to make the parsing process more efficient, we use a two-level parsing method. In this approach, we parse the model structure information and separate the trace records into groups according to the model (or sub-model) a record relates to. At plotting time, the number of trace records processed at each plot is thus much smaller than the original one.

For the convenience of users with different levels of computer programming knowledge and users with different plotting requirements, we provided two plotters which are called simple plotter, *splotter.py*, and customized plotter, *cplotter.py*. The simple plotter assumes that the sequential state of a DEVS model is determined by one attribute of the model state. The user just needs to select the model of interest, and the attribute determining the sequential state. The plotter will then visualize the trace automatically. The customized plotter is more powerful and complex. It provides a programming interface, by which the users can program themselves to determine how the sequential state is calculated from the whole state, and which part of the state attributes should be shown and how to show them in the plotter. In the simple plotter, the state attributes shown on the plotter are taken from the trace file directly, which is generated by the simulator using the state's string conversion function specified by the modeler.

The following part of the report is organized as follows. First, we discuss the design architecture of the plotter. Then we look into each part of the plotter in more detail. After that, an example shows how to use the simple and customized plotters is given.

3.2 Design and Implementation

3.2.1 Architecture of the Trace Plotter

When it comes to designing a trace plotter for DEVS, one first should consider the following issues. 1. Where does the trace data come from? 2. What is included in the trace file? 3. How is the trace represented?

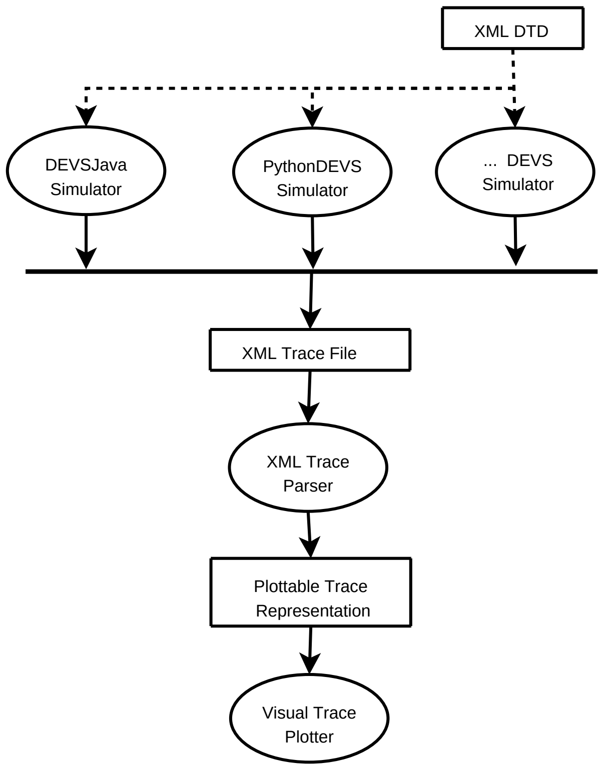


Figure 3.1: Architecture of the Trace Plotter

For the first question, a simple answer is that the trace data comes from a DEVS simulator. But the trick is that there are many different DEVS simulators (such as DEVSJava, ADEVS, and PythonDEVS) available now, and there are no commonly accepted rules to specify how the simualtion trace should be generated. This means that for different DEVS simulators, the simulation traces are different.

What is included in the DEVS simulation trace? This question faces the similiar situations as that for the first question. Because there is no standard for DEVS simulation trace output, so the contents of the simulation trace from different DEVS simulators are different. For some simulators, they only output values of state variables interested for every simulation, while others may output all state variables. Without knowing what is included in the trace output, it is implossible to develop tools to plot it.

At present, almost all the DEVS simulators are using private data structure to represent the simulation trace. Because the DEVS simulators are developed using different programming languages, the simulator-specific data structure usually has language-specific features. If we use the specific format used by one simulator, then it will be very difficult to use this plotter to plot a trace generated by other DEVS simulators.

Bearing these questions in mind, we design the architecture of the trace plotter as in Figure 3.1. In this architecture, the answers for the three questions are as follows. For the first question, the answer is that the data can come from any DEVS simulator. For the second question, the answer is that what can be included in the simulation trace file is sepcified by the XML DTD. And for the third question, simulation traces are represented in XML files, and how XML files are organized is specified by the XML DTD. So, for any DEVS simulatior, if it can generate XML reprenented simulation trace conforming to the XML DTD in Figure 3.1, then the simulation trace can be plotted by the visual DEVS trace plotter.

From Figure 3.1 we can see that the Trace Parser takes the XML represented trace file as input, and parses the XML file into a represention that can be recongnized by the trace plotter. Then, the plotter does the plotting job. Before we discuss how the XML trace is parsed and plotted, let us look into the XML DTD to see what is specified in a XML represented simulation trace.

DTD for XML Trace Representation

1. <!ELEMENT trace (event+)>
2. <!ELEMENT event (model, time, kind, port*, state)>
3. <!ELEMENT model (#PCDATA)>
4. <!ELEMENT time (#PCDATA)>
5. <!ELEMENT kind (“IN”|“EX”|#PCDATA)>
6. <!ELEMENT port (message)>
7. <!ELEMENT message (#PCDATA)>
8. <!ELEMENT state (attribute+)>
9. <!ELEMENT attribute (name, type, value+)>
10. <!ELEMENT name (#PCDATA)>
11. <!ELEMENT type (#PCDATA)>
12. <!ELEMENT value (#PCDATA|attribute)*>
13. <!ATTLIST port name CDATA #IMPLIED>
14. <!ATTLIST port category (“I”|“O”) #REQUIRED>
15. <!ATTLIST attribute category (“P”|“C ”|“ PC ”|“ CC”) #REQUIRED>
16. <!ENTITY title “DEVS Simulation Trace”>
17. <!ENTITY publisher “MSDL”>
18. <!ENTITY copyright “Copyright 2006 MSDL”>

3.2.2 XML DTD for Trace Output

We have mentioned earlier that one of the benefits of using XML is the capability for file validation. The validation is done by checking whether an XML file conforms to a DTD or XML Schema. XML Schema and DTD serve for a same purpose, the major difference is that the expressiveness of XML Schema is more powerful than DTD. Correspondingly, the XML Schema is more complex than DTD. For simplicity reasons, we used DTD rather than XML Schema to specify the trace file.

The DEVS formalism is used to model discrete event systems. The DEVS simulators basically are all event-driven simulators. So the DEVS simulation trace is an ordered collection of events. For each event, which time the event occurs, what type the event is, internal or external, what the model ports' status is, and what the model state is, basically are the concerns a trace analysist wants to know. Based on these assumptions, we design the XML DTD for DEVS trace representation as below.

From the DTD, we can see that a trace file consists of a `trace` element. The `trace` element is composed of a collection of `event` elements, which includes at least one `event`. This is meaningful in the sense that a trace file without any event is useless. Based on line 1 of this DTD, a valid XML trace file will look like below.

```
<trace>
    <event> ... </event>
    <event> ... </event>
    .......
    <event> ... </event>
</trace>
```

Line 2 defines the `event` element of the trace file, which means that an `event` is composed of a `model` element, a `time` element, a `kind` element, and zero or more `port` elements, and a `state` element. The `model` element is the name of the DEVS model, in which the `event` occurs. The `time` means the simulation time at which the `event` occurs. The `kind` element is the type of the `event`, where "`IN`" means internal event, and "`EX`" means external event. The `port` element includes information about model port status when the `event` occurs. The `state` element provides the state information of the model when the current event occurs. So an `event` element in the XML trace file can be represented as follows.

```
<event>
    <model> ... </model>
    <time> ... </time>
    <kind> ... </kind>
    <port> ... </port> * optional
    <state> ... </state>
</event>
```

The model name specified by the `model` element is the fully qualified name of a DEVS model. A full qualified model name is a model name that can reflect the composition relation among parent and its child models. For example, if we simulate a coupled model `A` that has a structure as shown in Figure 3.2, then the fully qualified name for model `A` is still `A`. And for sub-models `B`, `C`, `D`, and `E`, their fully qualified names are `A.B`, `A.C`, `A.B.D`, and `A.B.E` respectively.

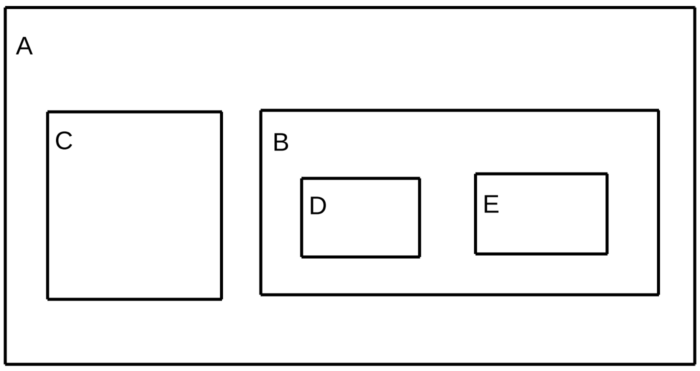


Figure 3.2: Fully Qualified Model Name

Line 5 in the DTD specifies values that are allowed for the `kind` element. For an atomic model, the kind for an event can be "`IN`" or "`EX`", where "IN" means internal event, "EX" means external event. For an coupled model, the `kind` can be empty or anything allowed in XML PCDATA. The reason we specify the value of kind for atomic models is that it is reasonable to distinguish external events from internal events when we plot a simulation trace.

Lines 6, 7, 13 and 14 specify the structure of the `port` element. The `port` element has two attributes, `name` and `category`, and one `message` element. The `name` attribute is just the text name of the `port` in question. The values of the `category` attribute of a `port` element can be "`I`" or "`O`", where "`I`" means input port and "`O`" means output port. The `message` element is the current message (or event) in the DEVS port. The represention of the message is in textual format. If the message structure is complex, the modeler should provide a function to transform it into a text format. Below is an example of a valid `port` element, which means that there is a message "`Job id=1, size=3`" at an input port named "`in`".

```
<port name="in", category="I">
        <message>Job id=1, size=3</message>
</port>
```

Lines 8, 9, 10, 11, 12, and 15 specify the structure of the `state` element. A `state` is composed of one or more `attributes`. Each `attribute` of the `state` has an attribute called `category`, and three elements `name`, `type`, and `value`. The `category` attribute of the `attribute` element can be "`P`", "`C`", "`PC`", or "`CC`", where "`P`" means primitive type, "`C`" means customized (or user defined) type, "`PC`" means a collection of values of a primitive type, and "`CC`" means a collection of values of a customized type. The `type` for an `attribute` can be any text that is meaningful to represent the `attribute` data type.

Below is an example of a customized `attribute` whose `category` attribute is "`C`". Its `name` is `currentJob`, and the `type` is `Job`, which is a user defined type. The `Job` type has two attributes, `ID` and `size`. In this case, the `currentJob's value` is `ID=1` and `size= 4.2687284882`.

```
<attribute category="C">
    <name>currentJob</name>
    <type>Job</type>

    <value>
        <attribute category="P">
            <name>ID</name>
            <type>Integer</type>
            <value>1</value>
        </attribute>
        <attribute category="P">
            <name>size</name>
            <type>Float</type>
            <value>4.26872848822</value>
        </attribute>
    </value>

</attribute>
```

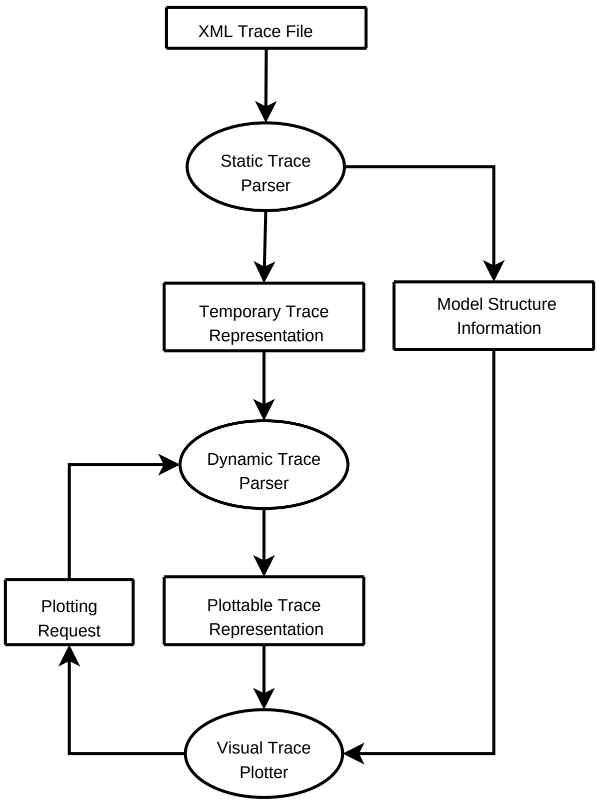


Figure 3.3: Two-level Trace Parser

3.2.3 Trace Parser

After we define the format of the trace file, it is the Trace Parser's job to parse the XML file into a format that can be plotted by the Trace Plottter. As we mentioned earlier, the number of records (events here) in a simulation trace file is usually huge. So, if we parse the file every time a user wants to plot a different property of a model, the plotting process will not be very efficient. In order the improve the plotting performance, we design a two-level parsing architecture, in which we use two parsers, a static parser and a dynamic parser to finish the trace parsing job. The structure is shown in Figure 3.3.

The idea of the two-level parsing architecture comes from the fact that the structure of the model will not change at plotting time. For the trace of a coupled DEVS model, if we can group the events into collections according to the model that an event belongs to, the number of events that need to be parsed will be significantly decreased. So the task of the Static Parser is to traverse a whole XML trace file to get the structural information of the model, and to split trace events into groups based on the model (or sub-model) that an event belongs to. In order to regroup the trace events, the static parser partialy parses trace events and transforms them into a temporary trace representation. The temporary trace representation will be further parsed by the dynamic parser to do the real plotting task.

Temporary Trace Representation and Plottable Trace Representation

The temporary trace representation and the plottable trace representation have the same information. The main difference is the representation of the model state information. In the temporary trace representation, the state inforamation is still in XML format, while in the plottable trace representation, the state information is represented in pure text format. There are two reasons for keeping the state information in XML format in the temporay representation. First, the task of the static parser is to parse the structure information of the model. The model structure information can be obtained without parsing the state of the model. The second, and the most important reason is that because we support the dynamic plotting function, the meta-information, like the attributes of a state and the type of each attribute of the state must be kept.

The temporary trace representation is discribed in Figure 3.4. The XML trace file is parsed into a collection of `TmpEvent` object ordered by the `time` attribute, which represents the simulation time that an event occurs. The `type` property of the `TmpEvent` represents the type of the event, namely external event or internal event. The `portInfo` is the textual representation of the ports status when this event occurs. This information is parsed from the `port` element of the XML representation and it includes both input and output ports' status. The `xmlState` is the state element of a trace event in the XML representation. It is still kept in XML node format. How to parse the state part of the XML is the job of the Dynamic Parser.

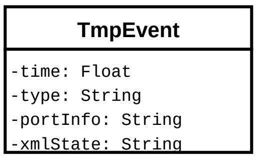


Figure 3.4: Temporary Trace Representation

Figure 3.5 is the class diagram of the plottable trace representation. The `time` and `type` property have the same meanings as that of the temporary representation. The `state` attribute comes from the original intention of plotting sequential state of a DEVS model. Now, it means a special value obtained from the `xmlState` of a `TmpEvent` using specific plotting (or parsing) rules. It can be a value of a specific state variable or a value obtained from several state variables using a specific parsing rule. How to get this value will be introduced in the following section when we discuss the `State Parser`. The `content` is the combination the textual representation of port information and the textual representation of the state information. Here, the XML represented state information has been parsed into textual format.

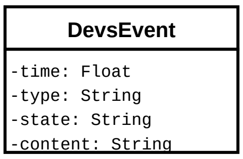


Figure 3.5: Plottable Trace Representation

Static Parser

The class diagram of the static trace parser is shown in Figure 3.6. The static parser takes a trace file name as its parameter. After the static parsing process, the results are save into `modelSet`, `modelEvents`, `modelState`, `modelType`, and `coupleComp` properties.

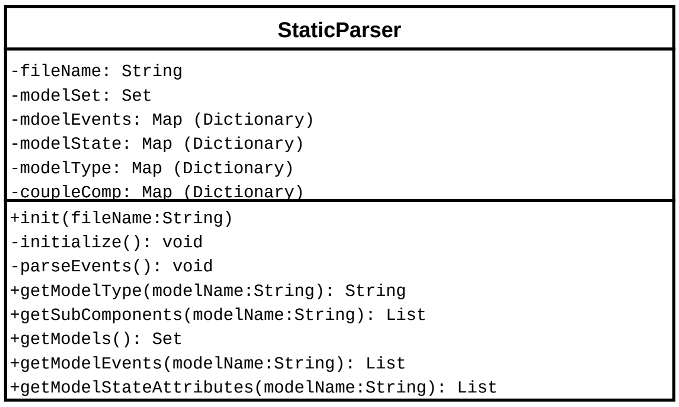

StaticParser
-fileName: String
-modelSet: Set
-mdoelEvents: Map (Dictionary)
-modelState: Map (Dictionary)
-modelType: Map (Dictionary)
-coupleComp: Map (Dictionary)
+init(fileName:String)
-initialize(): void
-parseEvents(): void
+getModelType(modelName:String): String
+getSubComponents(modelName:String): List
+getModels(): Set
+getModelEvents(modelName:String): List
+getModelStateAttributes(modelName:String): List

Figure 3.6: Class Diagram of the Static Trace Parser

The `modelSet` is the set of sub-models included in the simulated model. Each sub-model is identified by its full qualified name. For coupled model `A` in Figure 1.5, the `modelSet` of `A` is {`A, A.B, A.C, A.B.D, A.B.E`}. The events of each model are saved into a map structure called `modelEvents`, in which model names are used as keys and a list of model events as the value of a corresponding key. For a trace file like Example 3.1, the key and value pairs of `modelEvents` as given in Table 3.1.

```
<trace>
        <event><model>A.C</model>e1</event>
        <event><model>A.B.E</model>e2</event>
        <event><mdoel>A.B.D</model>e3 </event>
        <event><model>A.C</model>e4</event>
        <event><model>A.B.E</model>e5</event>
        <event><model>A.B.E</model>e6</event>
</trace>
```

Example 3.1: Trace Example for Static Parser

Keys	Value
A	e1, e2, e3, e4, e5, e6
A.B	e2, e3, e5, e6
A.C	e1, e4
A.B.D	e3
A.B.E	e2, e5, e6

Table 3.1: Values of modelEvents for Example 3.1

Both the events for coupled models and events for atomic models are saved in the same map structure. The structure of a coupled model event and the structure of an atomic model event are different. Events for coupled models are parsed into `DevsEvents`, however, events for atomic models are parsed into `TmpEvents`. The reason for this difference is that we do not know which part of the state will be used for plotting, so we should keep all the state information for atomic models, so we can dynamically parse the state when users change their plotting criteria.

The `modelState` is also a map structure, which saves the state information about each model indexed by the model name. The state information for each model is simply a list of attribute names of the model state. Because the coupled models have no state information, it only works for atomic models. `ModelType` is used to indicate whether a model is an atomic model or a coupled model. An atomic model is indicated as "`A`", and a coupled model is represented by "`C`". For coupled models, the map structure `coupleComp` maps from a model name to a list of its atomic models. For the model `A` in Figure 1.5, the `coupleComp` will look like Table 3.2.

Keys	Value
A	A.C, A.B.D, A.B.E
A.B	A.B.D, A.B.E

Table 3.2: Values of coupleComp

The operations of the TraceParser class just return the values as that are indicated by function names. The meanings are obvious.

Dynamic Parser

Figure 3.8 is the class diagram of the dynamic parser. This class' constructor takes a list of trace events and a state parser as its parameters. The trace events are of type `TmpEvent`, the results of the static parser. We mentioned earlier that in the temporary trace representation the state information is still in XML format. So the state parser's task here is to parse the state information in the XML representation. For each event in the event list, the dynamic parser gets a `state` (or value) using the state parser provided. Then it translates the XML represented state information into a pure textual representation. The textual state information together with the port information in the temporary representation are used to compose the `content` value of the plottable trace representation. The value parsed from the XML state information using the state parser is saved in the `state` property in the plottable trace representation. The mapping relation is shown in Figure 3.7.

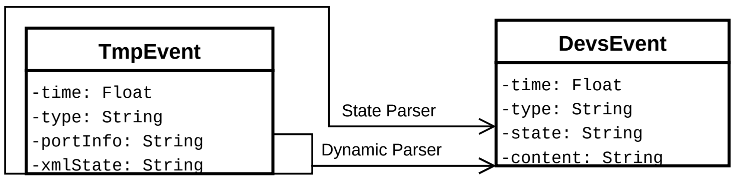


Figure 3.7: Parsing TmpEvent into DevsEvent

The `events` attribute of the dynamic parser is a list of plottable events, the result of the dynamic parsing. The `states` attribute is a set of values obtained from the XML state representation using the state parser for all events. These two are the information needed by the visual plotter.

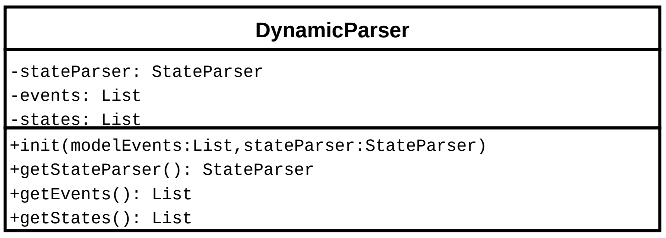

DynamicParser
-stateParser: StateParser
-events: List
-states: List
+init(modelEvents:List,stateParser:StateParser)
+getStateParser(): StateParser
+getEvents(): List
+getStates(): List

Figure 3.8: Class Diagram of Dynamic Parser

Besides the constructor, there are three public functions of this class. The `getStateParser` returns the current state parser used; the `getEvents` returns a list of generated `DevsEvents`; and the `getStates` returns a set of states (or values) parsed from all the events.

State Parser and Simple State Parser

Figure 3.9 is the class diagram of the state parser. There are two tasks for a state parser. First, get a value from the XML represented state for interesting state variables. Second, translate the XML represented state into a pure textual presentation. The first task is done by `getSeqState` function and the second is done by the `getString` function. Both these functions take an XML represented model state as parameter. The `getSeqState` returns a textual represented value, and the `getString` returns the textual representation of the model state.

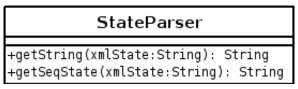

StateParser
+getString(xmlState:String): String
+getSeqState(xmlState:String): String

Figure 3.9: State Parser

For a given XML represented state, how to display it on the screen, and how to get a value for plotting are dependent on the user's interest. This means that only the user knows how to define these functions. So it is the user's task to provide a valid state parser for dynamic state parsing. For different plotting purposes, the parsers will be different.

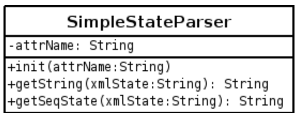

SimpleStateParser
-attrName: String
+init(attrName:String)
+getString(xmlState:String): String
+getSeqState(xmlState:String): String

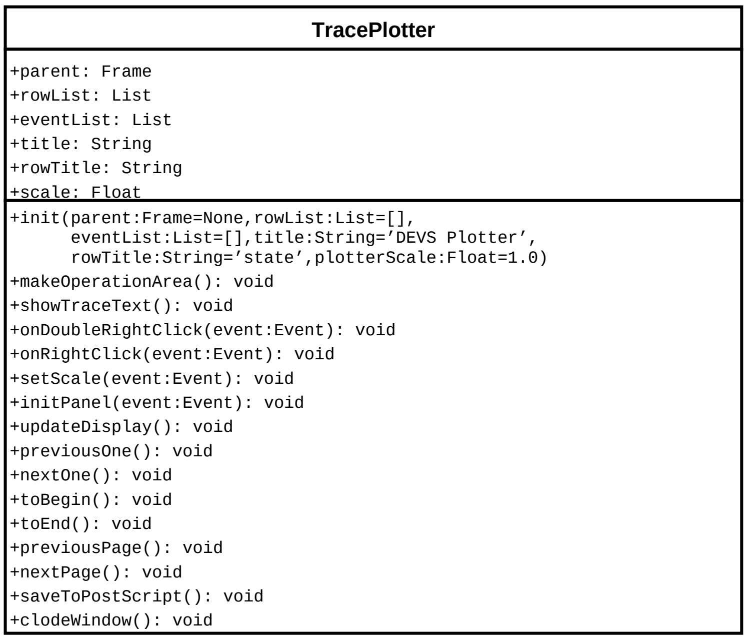
Figure 3.10: Simple State Parser

In order to make it easy to use, we provide a predefined version of the state parser called simple state parser. The class diagram of a simple state parser is shown in Figure 3.10. The `StateParser` and the `SimpleStateParser` provide the same interface to users. The difference is that the simple state parser takes an attribute name as parameter. This means that the `getSeqState` function of the simple state parser can only get the value of a single state variable. For some simple plotting tasks like watching the value change of a state variable along the simulation time, the simple parser is very useful. The user only needs to specify the state variable that is of interest, and the system takes care of the rest. The weakness of this state parser is that it gets the value of the state variable from the trace without any processing. If one wants to do some pre-processing before plotting, the normal state parser is the only choice.

3.2.4 Visual Trace Plotter

The XML represented trace has been parsed. How to plot it is the job of the visual plotter. The class diagram of the trace plotter is shown in Figure 3.11.

There are six parameters needed to construct a visual plotter. The `parent` is a frame window in which the plotter will be displayed. The meaning of `rowList`, `title`, and `rowTitle` are shown in Figure 3.12. The `eventList` is a list of `DevsEvents` to be plotted. It corresponds to the `events` property of the `Dynamic Parser`. The `scale` parameter specifies how to map the pixel of the screen to the time unit of events. The default is 1.0, which means that one pixel on the screen corresponds to one time unit. The trick for DEVS event plotting is in the values of the `rowList`. Originally, the `rowList` only includes the sequential states of the model being plotted.



```
TracePlotter
+parent: Frame
+rowList: List
+eventList: List
+title: String
+rowTitle: String
+scale: Float
+init(parent:Frame=None,rowList:List=[],
      eventList:List=[],title:String='DEVS Plotter',
      rowTitle:String='state',plotterScale:Float=1.0)
+makeOperationArea(): void
+showTraceText(): void
+onDoubleRightClick(event:Event): void
+onRightClick(event:Event): void
+setScale(event:Event): void
+initPanel(event:Event): void
+updateDisplay(): void
+previousOne(): void
+nextOne(): void
+toBegin(): void
+toEnd(): void
+previousPage(): void
+nextPage(): void
+saveToPostScript(): void
+clodeWindow(): void
```

Figure 3.11: Visual Trace Plotter

The nature of DEVS is such that a model can only change its state when an event occurs. So, besides the sequential state, the values of any state variable can be used as `rowList` to be plotted. When the sequential states are used, we can see the sequential states change along the occurring time of events. While the values of a state variable or the combined values of some state variables are used as the values of the `rowList`, we can see when the values change, and which events cause the value changes. The `rowList` can be obtained from the `Dynamic Parser's states` property.

Figure 3.12 is an screen shot of the event plotter with some marks, from which one can see how the `rowTitle`, `rowList`, `title`, `scale`, and `eventList` are displayed on the screen.

Simple Plotter and Customized Plotter

Corresponding to two kinds of state parsers, two different user interfaces are provided. The `Simple Plotter` provides interfaces for using the simple state parser, and the `Customized Plotter` gives interfaces for the customized state parser.

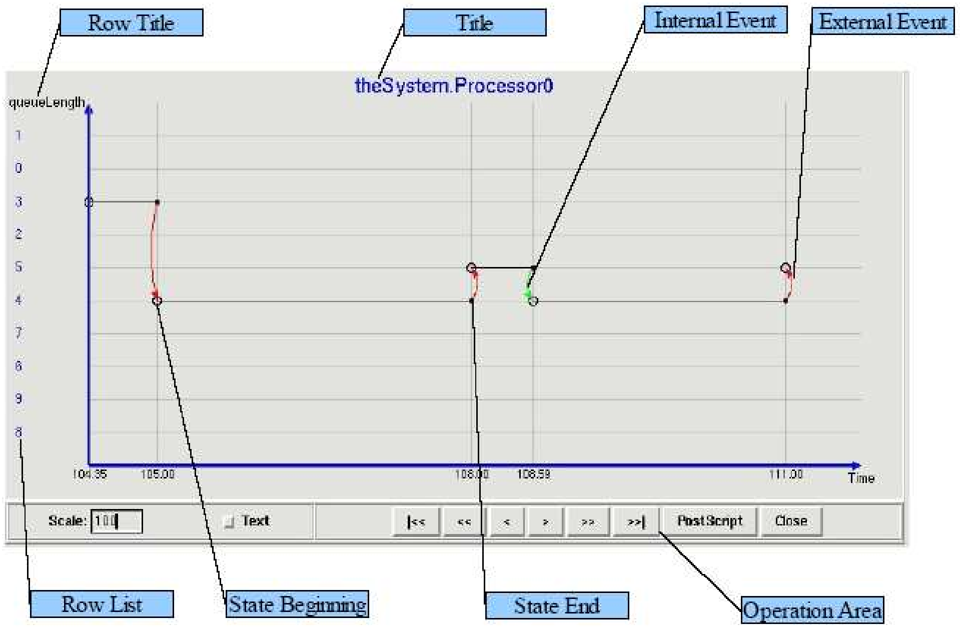


Figure 3.12: Terms Used in Trace Plotter

3.3 Case Study

In the previous two sections, we have discussed the design and implementation of the trace plotter. Now let us look at a case study to see how to use the trace plotter and what it looks like. The model used in this case study is the `Processor` model in chapter 6.

```
<trace>
    ... ...

    <event>
        <model>theSystem.Processor0</model>
        <time>3.0</time>
        <kind>EX</kind>

        <port  name="IN"  category="I">
            <message>(job 2, size 5.527549)</message>
        </port>
        <state>
            <attribute  category="P">
                <name>processorStatus</name>
                <type>String</type>
                <value>BUSY</value>
            </attribute>
            <attribute  category="P">
                <name>queueLength</name>
                <type>Integer</type>
                <value>1</value>
            </attribute>
            <attribute  category="C">
                <name>currentJob</name>
                <type>Job</type>
                <value>
                    <attribute  category="P">
                        <name>ID</name>
                        <type>Integer</type>
                        <value>1</value>
                    </attribute>
                    <attribute  category="P">
                        <name>size</name>
                        <type>Integer</type>
                        <value>4.26872848822</value>
                    </attribute>
                </value>
            </attribute>
        </state>
    </event>

    ... ...
```

Example 3.2: Simulation Trace of the Processor Model

3.3.1 XML Represented Trace for the Processer Model

In order to use the plotter, we first have to generate the simulation trace. Depending on the length of the simulation time and the complexity of the model, the simulation trace can be very huge. Example 3.2 represents one event in the simulation trace. It can be read as "at simulation time 3.0, an external event occurs at the model `System.Processor0`. At that time, the port and state status of the model are ...... (as shown by the port and state element in XML)".

3.3.2 Plotting Trace Using the Simple Plotter

Figure 3.13 is the graphical user interface for the simple plotter. From the file menu, one can select the trace file to be plotted. After a file is selected, the plotter calls the static parser to parse the file, and the model structure information is displayed. Users can select the model they are interested in. Once a model is selected, the model state attributes are displayed. One can select an attribute from the attributes list. Once an attribute is selected, the plotter uses the selected attribute as parameter to construct a simple state parser, which can then be used by the dynamic parser to parser the trace state. And finally, the results are displayed. Figure is a screen shot for plotting `processorStatus` and `queueLength` attributes of `theSystem.Processor0`.

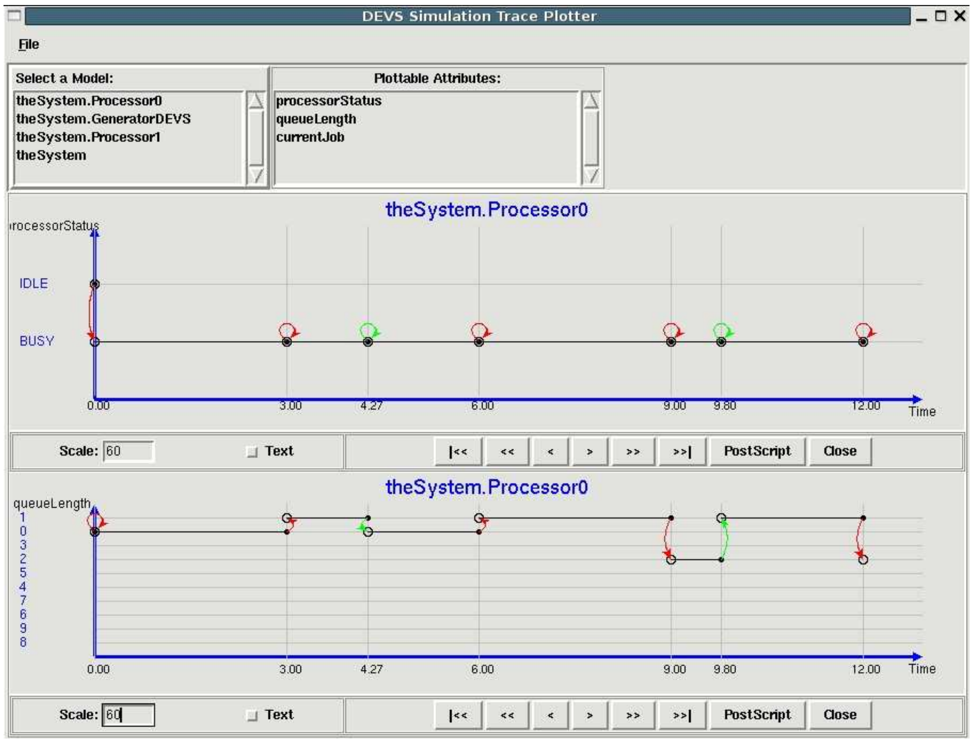


Figure 3.13: Plotting Trace Using the Simple Plotter

3.3.3 Plotting Trace Using the Customized Plotter

Figure 3.14 is the graphical user interface for the cutomized plotter. The file menu and the model selection part have the same meanings as that of the simple plotter. Unlike in the simple plotter, after a model is selected, a state parser must be given. This state parser is written by the user. After a state parser is given, the plotter calls the dynamic parser using this state parser. The results of the dynamic parsing are then displayed. Figure 3.14 shows the screen shot of using a customized state parser named `PStateParser.py` for a `Processor` model `theSystem.Processor0`.

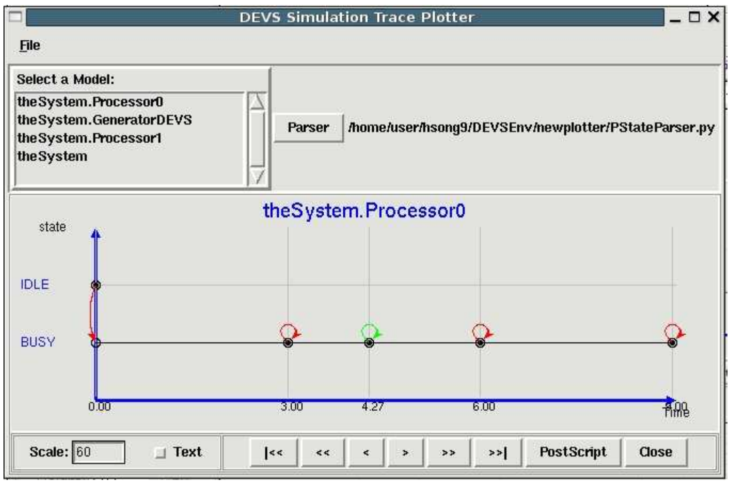


Figure 3.14: Plotting Trace Using the Customized Plotter

The main difference between the `Simple Plotter` and the `Customized Plotter` is that one must provide a `State Parser` for using the `Customized Plotter`. Example 3.3 is the customized state parser `PStateParser.py` used in Figure 3.14. This example is very simple. It is only used for demonstration purpose. The idea is the same as writing a complex one.

```
from xml.dom.minidom import *

class PStateParser(object):

    def getString(self, xmlState):
        for node in xmlState.childNodes:
            if node.nodeType == Node.CDATA_SECTION_NODE:
                return node.nodeValue #node.childNodes[0].data

        return "No state string"
```

continue ......

```
    def getSeqState(self, xmlState):
        attributes = xmlState.getElementsByTagName("attribute")
        for attribute in attributes:
            attrName = attribute.getElementsByTagName("name")[0]
            txtName=attrName.childNodes[0].data
            #print txtName, "–", self.attrName
            if (txtName=="processorStatus"):
                attrValue = attribute.getElementsByTagName("value")[0]
                txtValue=attrValue.childNodes[0].data
                return txtValue
            return "Error"
```

Example 3.3: A Customized State Parser for the Processor Model

3.4 Conclusions

In this chapter, we discussed the design and implementation of the DEVS Visual Trace Plotter. The main features of the Plotter include: 1. Standardized XML trace representation; 2. Two-level trace parsing architecture; 3. Simple predefined state parser and customizied state parser.

The standardized XML trace representation makes it possible for the plotter to be used to plot not only the simulation trace generated by Python DEVS, but also simulation traces generated by other simulators that follow the XML DTD. The simulation trace for a complex system is usually huge. The two-level trace parsing architecture, which first parses the model structure information and separates event trace records to groups according to the model a record belongs to, makes the trace parsing process more efficient. The two types of state parsers make the plotter easy to use.

4 Modelica Representation and Model Compiler

4.1 Introduction

With the development of computer software and hardware technologies, modelling and simulation technology are used more extensively. As a consequence of this trend, more and more modelling languages appear for describing models in different application domains. SDL[EHS97] has been used in the telcommunication industry for describing protocol models, UML has been used by software engineers for specifying business and software models, Modelica [Fri04] has been used for describing models of physical systems, and so forth.

A modelling language, also called a model description language, is ideally a declarative language for describing model specifications. Though they share many common features with programming languages, modelling languages and programming languages serve different purposes. Generally speaking, modelling languages stress high-level solution specifications in some appropriate formalism(s). The main task of a modelling language is to make a solution specification concise, precise, and easily understood. Programming languages emphasize on the execution of a solution, whose main task is to represent the solution in a way that can be executed correctly and effeciently by a computer.

DEVS is a formalism for discrete event system modelling. Because of its mathematical rigour and capability of hierarchical modelling, DEVS has been applied in many different fields [Zei03] [ZKB99][ZV93]. However, in many available DEVS modelling environments, like DEVSJava [Zei05], PythonDEVS[BV02], ADEVS[Nut05] and so forth, models are still described in programming languages. In fact, DEVS capabilities are commonly "grafted" on an object-oriented programming language. This gives the modeller access to the full expressive power of the programming language such as inheritance, lexical scoping, and libriaries. The weaknesses of using programming languages to represent models (see chapter 2) have severely affected the application and standardization of DEVS.

[HK06] proposes a high level formalism-specific model description language DEVSpecl for DEVS model representation. This language is designed for DEVS, so it has features specific for DEVS model descriptions. However, DEVSpecl is a purely new language. It has few users and is difficult to use for specifying non-DEVS models. DEVS has been evisioned as a suitable formalism for integrating systems that have both continuous and discrete components, as discribed in [ZPK00]. Using a language that lacks the capability of describing continuous models may undermine DEVS' potentials.

In this chapter, we discuss our effort of using the declarative object-oriented modelling language Modelica to describe DEVS models. Modelica is an object-oriented model description language that has been successfully used to specify models of systems in many physical domains. Though Modelica is originally designed for describing physical models, it has enough constructs for discrete event model description.

The following features lead us to Modelica as the model description langage for DEVS model description. Firstly, Modelica has been developed and applied for over a decade, it is a relatively mature modelling language. Secondly, Modelica has a large model repository for diffent industries and a large group of users. Thirdly, DEVS has been envisioned as a formalism that is suitable for hybrid system modelling. Modelica has been successfully used for describing continuous models. If DEVS models are also described by Modelica, we are one step further to the goal of hybrid system modelling using both DEVS and continous model formalisms. Lastly, constructs such as (computationally) non-causal parameter-coupling equations may be used to increase the usability of DEVS.

The organization of this chapter is as follows. In the first section, we present a simple introduction of Modelica to get a basic idea of Modelica language features. In the second section, we discuss how to represent DEVS model components using Modelica constructs. In the third section, we look into issues of compiling a Modelica DEVS model representation into Python DEVS. In the fourth section, we give a case study of representing a DEVS model in Modelica and subsequently translating the Modelica representation into Python DEVS. Finally, in the fifth section, we draw conclusions for this chapter.

4.2 Modelica and Its Model Description Constructs

Modelica is an object-oriented model description language. It provides a structured, computer-supported way of doing mathematical and equation-based modelling. The main objective of Modelica is to make it easy to exchange models and use model libraries. The design goal of Modelica is to build a modelling language based on the Differential Algebraic Equation (DAE) formalism with discrete-event features to handle discontinuities and sampled systems. This design goal allows Modelica to be a multi-formalism, multi-domain, general-purpose modelling language. The approach built on non-causal modelling with true ordinary differential and algebraic equations and the use of object-oriented constructs facilitate reuse of model knowledge[Fri04].

Though Modelica is object-oriented, the Modelica view on object-orientation is different from normal object oriented languages. Since Modelica emphasizes structured mathematical modelling, object-orientation is viewed as a structuring concept for describing complex large systems. There are three ways for structure description in Modelica, which are hierarchies, component-connections, and inheritance. Hierarchy and inheritance are basically done in the same ways as normal object-oriented languages. The component-connection is a special feature of Modelica. In normal programming languages, interactions between components are done through function calls or message passing mechanisms. In Modelica, components interactions are done through connections. The idea is that each component has connectors (or ports) through which the component can send output and accept input. Connectors of different components can be connected together to compose connections. Then, the output of one component can be linked as an input of another component via connections. Modellers do not need to specify how output of one component can become input of the other component. This can be deduced automatically by the model compiler based on model equations.

Modelica is a declarative language. The concept of declarative description is that Modelica models are primarily mathematical descriptions. Dynamic behaviours of a system are declaratively specified by mathematical equations. Rather than specifying specific details on *how* to achieve the result of problems, Modelica models stress on *what* kind of mathematical relations hold among model members. This way of model description is at a higher level of abstraction than normal object-oriented modelling, since some implementation details can be omitted [Fri04].

Many other features make Modelica suitable for high-level model description. For space reasons, we do not discuss them here. Since our purpose is to represent DEVS models in Modelica, let us look into some Modelica constructs that are related to DEVS representation.

Class Like any object-oriented programming language, the basic structural element in Modelica is a class. Almost everything in the real technical world can be represented as a Modelica class, and a complex model can be hierarchically composed of classes through class inhertance and composition. But the structure of a Modelica class is different from classes in other languages. The significant feature making Modelica different from other programming languages is how Modelica uses classes for describing model behaviour. In common object-oriented programming language, system behaviour is described by methods. In Modelica, in most cases model behaviours are described by equations. Tough equations in Modelica are represented in the same way as assignments in other languages, Modelica equations only reflect the equality relationship among variables. There is no assignment relation among equation member variables. For example, in Java, ‘‘a = b + c’’ means assigning the addition result of variable b and c to variable a. In Modelica, equation ‘‘a = b + c’’ only means the relationship among variable a, b, and c. It has the same meaning as equation ‘‘b + c = a’’. Variables in an equation can be written in any order if the relationship reflected in the equation is not changed. Besides the equation feature for supporting non-causal modelling, Modelica provides function and algorithm constructs for normal causal modelling. The difference between functions in Modelica and functions in normal programming languages is that Modelica functions are also classes. They are special classes that can either stand alone or be embedded into other classes.

Restricted Classes Class is the fundamental structure element in Modelica. A class in Modelica can be defined using the keyword class. But under certain conditions, the keyword class can be replaced by one of five other, more specific keywords: `model`, `connector`, `record`, `block`, and `type`. On the one hand, the restricted class mechanism makes Modelica code easier to read and maintain. It is also modeller-friendly since the modeller does not need to learn several different language constructs, but just the class concept. On the other hand, all properties of a general class are identical to all kinds of restricted classes. For example, the syntax and semantics of definition, instantiation, inheritance, and general properties are defined in the same way for all kinds of classes. Such orthogonality simplifies the construction of a Modelica compiler since only the syntax and semantics of the class construct, along with some validity checks on a restricted class need to be implemented. The following summarizes the restrictions and usage of each kind of restricted class in terms of some examples [Xu05].

model The only restriction of a model restricted class is that it may not be used in connections. Its semantics are identical to the general class construct in Modelica, and it is most commonly used.

record The record class is used to describe structured data. No equations are allowed in the definition or in any of its components. For example:

```
record Address
    String streetName;
    Integer apartmentNumber;
    String city;
    String zip;
end Address;
```

type A type is a class that is an alias or extension of an existing class. A type restricted class may only be an extension to a predefined type, `enumeration`, `record`, or `array` of type. Basically, the purpose of using type is to identify a data structure by a meaningful name. Below is a simple example of using type.

```
type Point = Real[2];
class class1
    Point p = {1.0, 2.0};
    ......
end class1;
```

connector The restrictions of connector classes are identical to those of record classes, except that connector classes are designed to be used in connections.

block The block restricted class is used to model causal (input/output) block diagrams. In Modelica, the two keywords, input and output, are used as component prefixes to postulate the data flow direction. All declared variables in a block must either have the prefix input or output. A block class may not be used in connections. Below is a simple example of the block construct.

```
block RectangleArea
      input Real width;
      input Real height;
      output Real area;
equation
      area = width * height;
end RectangleArea;
```

Package Classes in Modelica can be organized into packages. Package is a restricted class with some enhanced capabilities. It is restricted in the sense that it may only contain class definetions and constant declarations. In a Modelica package, variable and parameter declarations are not allowed. Without variables and parameters, the equation part is meaningless and certainly not allowed. However, the Modelica package has some enhanced functionalities that normal classes do not have. First, a package can import name spaces or components from other packages. Second, a package can be marked as encapsulated, which means that this package is an independent unit, usage or reference of components in other packages must be explicitly imported. A simple example of Modelica package is shown below.

```
package packexample
      import pack1.*;
      import pack2.Class1;
      constant Real pi = 3.14159;

      class DemoClass
            ......
      end DemoClass;

      ......

end packexample;
```

Function Functions are natural parts of mathematical models. So it is natural to have structures in the Modelica language allowing users to define mathematical functions. The body of a Modelica function is an algoritm that specifies the execution behaviour when the function is called. The parameters for a function are specified by the keyword `input`, and results are saved to variables marked by the `ouput` keyword. There are some constraints on Modelica functions. Firstly, at most one `algorithm` clause is allowed in a function's body. No `equations` and `initial algorithms` are allowed. Secondly, calling a function requires either an algorithm or an external function interface. And thirdly, no calls to the Modelica built-in operators `der`, `initial`, `terminal`, `sample`, `pre`, `edge`, `change`, `reinit`, `delay` and `cardinality` are allowed in a function as their arguments are time-varying signals as opposed to instantaneous values. Below is a simple example of a Modelica function.

```
function Multiply
    input Real x;
    input Real y;
    output Real result;
algorithm
    result := x * y;
end Multiply;
```

Built-in Types There are four built-in types in Modelica, which correspond to primitive types Real, Integer, Boolean, and String. These built-in types are all classes. They have most of the features of normal Modelica classes. The only difference is that the value of the variable of a built-in class can be accessed directly through the variable's name, rather than using the dot notation [Fri04]. For example, if we have a variable declared in a class as `‘‘Real x;’’`, then we can use `x` as this `‘‘y := x;’’`. Rather than using `x.value`, the value of `x` is accessed directly and assigned to variable `y`'s value attribute.

4.3 Design and Implementation

We have discussed the language features of Modelica and benefits of high-level model representation using Modelica. Now let us see how we can use Modelica to facilitate DEVS modelling and simulation. In this topic, we discuss two issues. The first is how DEVS models can be represented in Modelica. The second is how DEVS models represented in Modelica are translated for simulation purposes.

4.3.1 The Architecture

Modelica is a high-level model description languge. Models described in Modelica cannot be exectued or simulated directly. In order to further study or analyze the correctness and precision of a model, model simulation or execution is necessary. This requires some way to transform Modelica model representations into formats that are suitable for execution or simulation. As a common way of practice, the task of transforming high level model representations into executable or simulatable formats is done by model compilers.

As in the case of compilers for programming languages, a model compiler has at least two major tasks, syntax checking and target code generation. Syntax checking is to make sure that the source code satisfies the source language grammar. Syntax checking is the premise of correct target generation. Syntactically correct source makes correct target generation possible. Target generation is to translate the source code into a desired representation, be it a representation in another language or machine code.

When it comes to compiling DEVS Modelica representations, the following tasks need to be done by a DEVS Modelcia compiler. Firstly, it is the compiler's responsibility to make sure that those models are syntactically correct Modelica models. Secondly, the compiler has to check that those DEVS models syntactically satisfy the DEVS Modelica specification. Thirdly, Modelica is a typed language, so the model compiler must check that those models satisfy Modelica's type rules. And finally, the model compiler must have the capability to generate simulator-specific model representation. Here the simulator-specific model representation can be any format that can be simulated by a DEVS simulator. It could be Java code for DEVSJava, C++ code for ADEVS, or Python code for Python DEVS, and so forth.

Due to the fact that there are many existing DEVS simulation environments, the Modelica representation should be expressive enough, so it can be transformed into any DEVS simulator-specific representation. Figure 4.1 is the software architecture for using Modelica in DEVS modelling and simulation. We envision Modelica as a standard DEVS model representation. We can build a Modelica model repository for models in different fields. The models in the Modelica repository are not necessarily pure DEVS models. When one models a new system, for components that are already in the model repository, the existing models can be reused directly. The modeller only needs to build models for components without existing models. The newly created models and reused existing models in the Modelica repository are combined together by the model compiler.

This way of using Modelica for DEVS modelling and simulation has the following benefits. First, it promotes high-level model knowledge reuse. Since a Modelica model representation is at a higher level of abstraction than a programming language model representation, model reuse in a Modelica repository is at a higher level than model reuse at programming language library level. High level models usually do not change as often as low level models, so the benefit is obvious. Second, it promotes DEVS standardization. High-level models omit many implementation related details, so they are easy to be standardized. And thirdly, Modelica has the capability for multi-domain, multi-formalism model representation, while DEVS is a potential formalism for integrating discrete and continuous systems for hybrid modelling and simulation. Combining Modelica and DEVS opens many chances for future development.

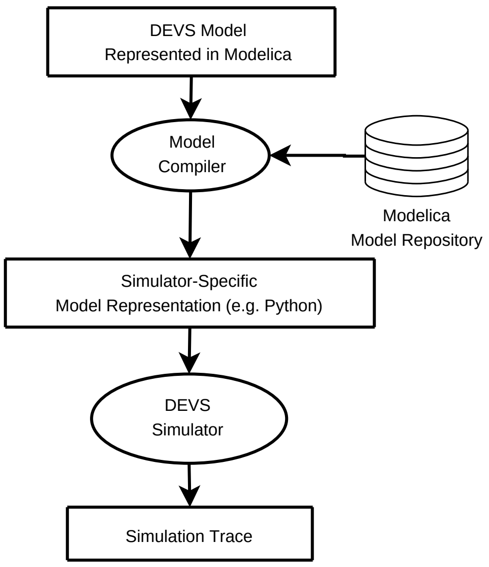


Figure 4.1: Architecture for Modelica based DEVS Modelling and Simulation

4.3.2 Representing DEVS in Modelica

Modelica Constructs vs. DEVS Components

As discussed in chapter 1, there are two types of DEVS models, atomic DEVS models and coupled DEVS models. An atomic DEVS model has inputs, outputs, sequential states, external transition function, internal transition function, time advance function, and output function. Coupled models have inputs, outputs, ports, and sub-models. As in a programming language, atomic and coupled DEVS models can be represented as Modelica classes. Sub-models are described by class instances. Structural inputs and outputs can also be described as Modelica classes. Transition functions can be represented as Modelica functions. For the sequential states, Modelica has an enumeration type, which can be used to represent enumerated values.

Now we know which Modlica constructs can be used to represent which DEVS components. The question is, if we just describe DEVS models using Modelica classes, how we can know which class is `AtomicDEVS` and which class is `CoupledDEVS`. To solve this problem, we use "prototypes". We define a DEVS package, in which all the DEVS components are specified. All user-defined Modelica constructs, if they extend from the predefined DEVS classes, are treated as DEVS components. For example, if we find a Modelica class definition like this: `‘‘class Car extends AtomicDEVS;’’`, then we assume the `Car` class defines a atomic DEVS model. The `Car` class is a direct sub-class of the `AtomicDEVS`. If the `Car` class is further sub-classed by other classes, then those sub-classes of the `Car` class are also assumed to be atomic DEVS models. This way of sub-classing is called indirect sub-class. Similarly, all other DEVS components can be identified in this way.

Predefined DEVS Modelica Elements

Figure 4.2 is the class diagram of predefined DEVS Modelica classes. The root of the diagram is `DEVSElement`. All other classes are extended from the root class `DEVSElement`. The classes are in two groups. Classes extending from `DEVSComponent` represent DEVS components, and classes extending from `DEVSCollection` represent collection data structures.

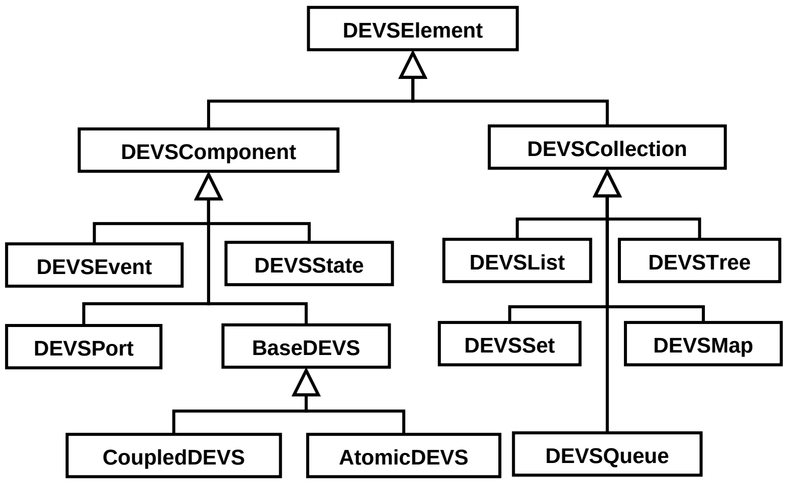


Figure 4.2: Predefined DEVS Elements in Modelica

There are three basic purposes for using predefined classes. First, classes extending from `DEVSComponent` are mainly used for identification purpose, which means that DEVS components' Modelica representation should extend from corresponding predefined classes. For example, if one wants to define an atomic DEVS model in Modelica, then the Modelica class representing that model must extend from the predefined `AtomicDEVS` class. Second, Modelica has enough constructs for continuous model description. However, for discrete event system, the array element in Modelica is not very convenient for operations on complex data collection. So classes extending from `DEVSCollection` are mainly for simplifying collection data operation. And finally, Modelica is a typed language. This means that a Modelica function can only work with parameters with the same types as in the function's declaration. For example, if we define a function accepting a parameter of `Integer` type, we cannot call this function with a parameter of `Real` type. Because there is no a common root class (like `Java's object` class) for all Modelica classes, the type rule makes building collection structure for holding data with different types very difficult. For example, we have a list of mixed DEVS component objects, which include DEVS port instances, model objects and events. If these objects have no a common ancester class, it will be difficult to hold them in one collection data structure. If the collection data structure has a `add` function accepting a parameter of `DEVSPort` type to add a DEVS port object, because of the type rule, you cannot use this `add` function to add an object with types other than `DEVSPort`. You have to define other functions to add other objects to the collection. With the support of the predefined hierarchical classes, things become easier. For the same `add` function, if we define it accepting a parameter of `DEVSElement` type, then, all the objects of sub-classes of `DEVSElement` can be valid parameters to this function.

Predefined BaseDEVS Except the BaseDEVS class, all other predefined DEVS component classes are empty classes purely for identification purpose. The BaseDEVS defines two utility functions for event processing. The definition of the BaseDEVS is shown as below.

```
class BaseDEVS

    function poke
        input DEVSPort outPort;
        input DEVSEvent evt;
    end poke;
    function peek
        input DEVSPort inPort;
        output DEVSEvent evt;
    end peek;

end BaseDEVS;
```

The poke function sends an event to an output port of a DEVS model. It takes two input parameters, evt the DEVS event that is being sent, and outPort, the DEVS port that the event is being sent to. The peek function retrieves an event from an input port. It takes two parameters, inPort, the port from which an event will be retrieved, and evt, the event that is retrieved from the inPort.

Predefined DEVSList We presented several collection data structures in Figure 4.2. We only define the DEVSList here for demonstration purposes. Below is the definition of the DEVSList class. Other classes can be defined in a similar way, when they are needed.

```
class DEVSList

    function append
        input DEVSElement de;
    end append;
    function pop
        input Integer index;
        output DEVSElement de;
    end pop;

end DEVSList;
```

We take the semantics of Python List as the semantics of the DEVSList functions. For simplificity reasons, we only defined two functions for the DEVSList. Append means append an element to the end of a list. It takes an input parameter of DEVSElement type. So all predefined DEVS classes and their children are valid elements to be appended to the list. The pop function takes an input parameter index of Integer type, and an output parameter de of DEVSElement type. This function pops a DEVSElement de at the position of the list indicated by the input parameter index to the output parameter de. Other list operations can be added to the DEVSList definition in a similar way.

Predefined DEVSState There is no DEVS state element in the DEVS specification. The DEVSState class represents the state variables of an atomic DEVS model. We represent the state variables in a separate class called `DEVSState`. This is also an empty class. However, the real state class for an atomic DEVS model must have at least one variable `seqState`, which represents the sequential state of that model with the type of that model's `SeqStates`. Each atomic DEVS model must have a variable called `state` representing its model state variables and sequential states. For example, we have an atomic DEVS model `AModel`. Then the definitions of `AModel` and its model state are as below.

```
class AModelState
    AModel.SeqStates seqState=AModel.SeqStates.initialState;
    ... ...
end AModelState;


class AModel
    extends AtomicDEVS;
    type SeqStates = enumeration("sq1", "sq2", ......);
    AModelState state();
    ... ...
end AModel;
```

Modelica Keywords with Transformed Meanings

`Input` and `output` keywords in Modelica are used to specify that a parameter of Modelica functions is an input or output parameter. In DEVS Modelica, we borrow these two words in DEVS model definitions with different semantics. The `input` in a DEVS Modelica class means an input DEVS port, and `output` means an output DEVS port. For example, the statement `‘‘input DEVSPort p_in();’’` means the declaration of a input DEVS port variable `p_in`. These meanings are only effective for port declarations in atomic or coupled DEVS model definitions. In functions, the `input` and `outout` keywords still have their original Modelica semantics.

In Modelica, functions should not have side effects on model state, which means that a Modelica function can only manipulate its parameters and local varilables, not the global model state variables. In DEVS Modelica, we deliberately loose this rule for state transition functions, output function, and time advance function of the atomic DEVS models. Because these four functions are inherent to DEVS atomic models, and the reason of their existance is to change the model state. It is very cumbersome to describe the model behaviour if we do not loose this function rule. Except these four functions, other functions in DEVS Modelica follow the normal Modelica function rules.

4.3.3 Representing DEVS Components in Modelica

Combining our above discussions, we get the Modelica prototypes for DEVS components.

Atomic DEVS Models An atomic DEVS model can be represented by a Modelica class like below.

```
class modelname
    extends AtomicDEVS;
    parameter declarations { parameter Integer i; ......}
    sequential states declaration { type SeqStates = enumeration("sq1", "sq2", .......);}
    model state declaration { ModelState state(); }
    input port declarations { input DEVSPort p_in1(); ......}
    output port declarations { output DEVSPort p_out1(); ......}

    function extTransition //external transition function
    end extTransition;
    function intTransition; //internal transition function
    end intTransition;
    function outputFunction //output function
    end outputFunction;
    function timeAdvance //time advance function
        output Real timespan;
    end timeAdvance;

end modelname;
```

The atomic model include parameters, sequential states, model state, input and output ports, and behaviour functions. In the parameter declarations part, parameters needed for instantiating a model are specified. Sequential states are represented using the Modelica enumeration type. The model state is declared as a normal Modelica class instance variable. Input and output ports are declared using input and output keywords and predefined class `DEVSPort`. All the functions are defined as normal Modelica functions. The `timeAdvance` function requires a output parameter `timespan`, through which the value of the time interval for specific sequential state is returned.

Coupled DEVS Models Similar to atomic models, the prototype of Modelica representation for coupled DEVS models is shown below.

```
class modelname
    parameter declarations
    input port declarations
    output port declarations
    sub-model declarations

    equation
        port connections {connect(m1.p_out, m2.p_in); ......}

end modelname;
```

A coupled DEVS model has parameters, input and output ports, sub-models, and port connections. Ports and parameters are declared in the same way at those for atomic models. Sub-models are declared as normal Modelica class instances. Port connections are defined by Modelica's `connect` function. Originally, the `connect` function is used to connect two Modelica connectors, and here we use it to connect two DEVS ports.

DEVS Event, Port, and Model State DEVS event, port, and model state are just represented as normal classes that extend from corresponding predefined DEVS classes. These classes are like the Modelica restricted class `record` in that they only have data members without an equation part. An extra requirement for the model state is that it must have a `seqState` variable as we mentioned above.

4.3.4 Using Language-Specific Library Functions in Modelica

When we do DEVS modelling in Modelica, we use utility functions very often. In most cases there are corresponding library functions in programming languages for utility functions. If we can use these functions, rather than implementing them again in Modelica, we can save a lot of time. There are two issues with using language-specific library functions in Modelica. First, the signatures of these functions must be declared in Modelica format. So we can make sure these funcions are used syntatically right at Modelica level. And second, there must be a way for the Modelica model compiler to know where the definitions of these functions are when the compiler transforms the Modelica model represnetations into simulator-specific models.

Declaring Language-Specific Functions in Modelica

For the first issue, we designed a special Modelica package called `externalfunctions`, in which the utility functions that can be found in programming languages are declared in Modelica format. Below is an example of the delcaration of a function `randint` in the `externalfunctions` package.

```
package externalfunctions

    function randint
        input Integer ia;
        input Integer ib;
        output Integer randvalue;
    end randint;
    ......

end externalfunctions;
```

Mapping Functions from Modelica to a Specific Programming Language

For the second issue, we define an XML file called `funcmapping.xml`, in which the mappings between utility functions' Modelica declarations and language-specific library locations are defined. So the model compiler can import appropriate libraries in target language when Modelica models are being compiled.

Two kinds of mappings can be defined in the `funcmapping.xml` file, package mapping and function mapping. Package mapping indicates a corresponding package name in the target language for a Modelica package. Function mapping specifies the target funciton name and package for a Modelica function. So during target generation, the compiler only needs to import a specific function in the target package rather than the whole package. Below is part of the `funcmapping.xml` file with a package mapping and a function mapping example.

```
<?xml version="1.0"?>

<mappings>

    <package-mapping>
        <name>devs</name>
        <mappackage>DEVS</mappackage>
    </package-mapping>
    <function-mapping>
        <name>randint</name>
        <mapname>randint</mapname>
        <mappackage>whrandom</mappackage>
    </function-mapping>
    ......

</mappings>
```

4.3.5 The Model Compiler: from Modelica to PythonDEVS

As we mentioned earlier, Modelica is a model description language. In order to do simulation, Modelica model representations must be translated into models that are represented in a programming language that can be simulated by a specific DEVS simulator. This task is done by the Modelica model compiler. Because Modelica is a general purpose modelling language, models represented in Modelica can be translated into models represented by most general purpose programming languages. Here we use Python as target language to see how we can compile DEVS Modelica models into Python DEVS models.

This part of our work is based on Steven Xu's μModelica compiler [Xu05]. μModelica focuses on the continuous parts of the Modelica language. It does not support constructs such as `function`, `algorithm`, `if ... else` statement and so forth. The code generated from μModelica is Octave code, which can be used by GNU Octave for numerical computation and experimentation. We enhance the μModelica compiler with the capability of processing programming language features of the Modelica language, so it can compile DEVS Modelica models.

Architecture of the DEVS Modelica Compiler

Figure 4.3 gives the architecture of our DEVS Modelica compiler. Like any normal language compiler, the DEVS Modelica compiler includes four parts, parsing, scoping (building symbol tables), type checking, and code generation. Even though the μModelica compiler only supports continuous features of the Modelica language, its parser supports the full language features of Modelica. So at the parser level, we use the parser of μModelica directly. At the scoping level, in addition to the programming language features of Modelica, the μModelica compiler does not support the import statement. Due to the fact that import is a important mechanism in Modelica for supporting model reuse, we add this function to our DEVS Modelica compiler at the scoping level. For the continuous language features, we reuse the functionalities of the μModelica compiler. Due to time limitations, we did not do much on type checking. We simply reuse μModelica's type checker for simple name look up. Since Steven Xu has discussed the μModelica compiler in detail in his thesis [Xu05], we do not spend much time on those issues in this thesis. The following discussions focus on the code generator that generates code from DEVS Modelica to Python DEVS.

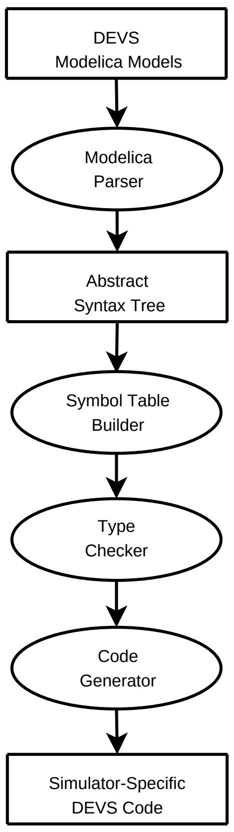


Figure 4.3: The Architecture of DEVS Modelica Compiler

Mapping between DEVS Modelica Representation and Python DEVS

Python DEVS is a DEVS modelling and simulation framework written in Python. As we discussed in chapter 1, Python DEVS has defined three DEVS components, atomic DEVS model, coupled DEVS model, and DEVS port. So the DEVS Modelica representation of atomic models, coupled models, and ports can be translated into corresponding Python DEVS components. In Python DEVS, events and model states are simply represented by normal Python classes. So DEVS events and model states in Modelica representation are simply translated into Python classes. We now know the mappings between DEVS Modelica components and Python DEVS elements. Let us see how to generate each Python DEVS component from DEVS Modelica.

Atomic Models As we mentioned above, an atomic Modelica model representation has parameters, ports, sequential states, model state and transition functions. The task of generating a Python DEVS representation for an atomic model Modelica representation is to generate each part of the atomic Modelica representation model into Python DEVS.

In Python, the construction of a class instance is done by the `__init__` function. Each Python class that needs parameters to initialize its instances must have an `__init__` method through which the initialization value for the parameters are passed to the class instances' attributes. The parameters of the `__init__` function of a Python class play the same role as the parameters of a Modelica class. So the parameters of a Modelica class are translated into parameters of the `__init__` function of a Python class when we translate a DEVS Modelica representation into Python DEVS.

In Python DEVS it is not necessary to declare ports. Input and output ports are added to models through method calls. There are two methods for adding ports to a DEVS model, `addOutPort` and `addInPort`. `AddOutPort` adds an output port to a DEVS model, and `addInPort` adds an input port to a DEVS model. Both these methods take a string representation of a port name *name* as parameter and return a port instance variable that points to the newly added port. These two methods are defined in the `BaseDEVS` class of the Python DEVS prototypes, so they can be called in `AtomicDEVS` and `CoupledDEVS` directly. For example, in a Python DEVS atomic model class' `__init__` function, ``self.out = self.addOutPort(``p_out'')'' means this model has an output port called ``p_out'', and this port can be accessed by `self.out` at runtime. In DEVS Modelica, the owner relationship that a model has with some ports is handled through port declarations. For example, ``input DEVSPort p_in();'' in Modelica means that a model has an input port ``p_in''. To translate port declarations in Modelica models into ports in Python DEVS models we use the following policy. Input port declarations are translated into Python DEVS' `addInPort` calls, and output port declarations are translated into `addOutPort` calls. The port name in Modelica declaration is translated as both port instance variable and port name. For example, the Modelica declaration of ``input DEVSPort p_in();'' is translated into ``self.p_in = self.addInPort(``p_in'')'' in Python DEVS.

Sequential states of an atomic DEVS model in Modelica are represented as members of a Modelica enumeration type. In Python, there is no enumeration type data structure. However, Python has the concept of class variables. The sequential state of an atomic DEVS model are class variables, because all model instances of the same kind of model have the same sequential states. So, the enumeration of sequential states in DEVS Modelica is translated into class variables in Python. For example, if atomic DEVS model `AModel` in Modelica representation has a statement ``type SeqStates = enumeration(``Idle'', ``Busy'');''. Then its Python DEVS representation will have code as shown below.

```
class AModel:
    Idle = "Idle"
    Busy = "Busy"
    ......
```

In DEVS Modelica, model state of a atomic model is declared as a variable of the atomic model. We treat it the same way when we translate a Modelica model into Python DEVS model. For example, ``ModelState state();'' in Modelica will be translated into ``self.state = ModelState()'' in Python DEVS.

In Modelica, functions are also classes with limited functionalities. In Python, methods/functions are members of a specific class. So, the embedded transition functions of an atomic DEVS Modelica model are translated into member functions of a Python DEVS class. The input parameters of a Modelica function are translated into parameters of a corresponding Python function. The output parameters of a Modelica function are translated to variables that are returned from the corresponding Python function. Modelica representation of internal transition function and external transition function of an atomic model have no output parameters. However, in Python DEVS, both the internal and external transition functions are required to return the model state after the transitions. So, the statement ``return self.state'' is added to the end of both these functions in Python DEVS. In Modelica, the time advance function has an output parameter `timespan`, so the last statement of the generated time advance function in Python DEVS is ``‘‘return timespan’’``. The `timespan` is a local variable of the function, there is no self qualifier before it. The output function in Modelica has neither input parameters nor output parameters, so there are no parameters and returns in the corresponding generated Python DEVS output function.

Coupled DEVS Model In DEVS Modelica, coupled models have parameters, ports, model instances (or sub-models), and connections. Parameters and ports can be translated into Python DEVS in the same way as that for ports and parameters of an atomic model. In Modelica, DEVS model instances are represented as normal Modelica class instances in the form ``‘‘model-name model-instance-variable( parameters );’’``. However, in Python DEVS, besides the normal parameters, each model has a textual name for instance identification in a simulation environment. In order to satisfy this requirement of Python DEVS, the model-instance-variable above is treated as both variable name and textual instance name when the Modelica representation is translated into Python DEVS. For example, the model instance declaration in Modelica ``‘‘AModel a1(x=1, y=2);’’`` is translated into ``‘‘self.a1=AModel(x=1, y=2, name=‘‘a1’’)’’`` in Python DEVS. The Modelica connect function for connecting DEVS ports in Modelica representation of coupled model is simply replaced by Python DEVS's `connectports` function, which is a member function of the Python DEVS coupled model prototype.

DEVS Events We have translated atomic models and coupled models. Next it is the turn of DEVS events. Events are much simpler in Modelica representation than models. Events only have parameters and attributes, which are simply translated to Python code as what we do for parameters and attributes of atomic models.

Primitive Data Types Modelica has four primitive data types, Real, Integer, Boolean, and String. They are translated into Python's Float, Integer, Boolean, and String respectively.

4.3.6 Python DEVS Code Generator

As we mentioned earlier, the μModelica parser supports the full language features of Modelica. The result of the μModelica parser is the Abstract Syntax Tree (AST) of the parsed model. A special feature of the AST generated by μModelica is that it is generated based on the Visitor Design Pattern [Xu05]. So generating code for the AST is just to create a new visitor to traverse the AST and generate code for each node of the tree.

The Structure of the Modelica Abstract Syntax Tree

Figure 4.4 shows a few elements on the top of the AST. Here the `ClassFile` corresponds to a Modelica source file. An AST may include one or more source files. Two situations can lead an AST to have more than one `ClassFile`. First, the compiler compiles over one file at a time. Second, one Modelica source imports classes or packages in other source files.

Each `ClassFile` is composed of zero or more `RegularClassDefinitions`. There is a short syntax for defining a class in Modleica, which is called `Short Class Definition`. This is usually used to give a informative type name to an existing class [Fri04]. For example, ``‘‘class Age = Integer;’’`` defines a new class `Age`, which has the same meaning as `Integer`. So, the `RegularClassDefinition` here is a general concept. It means all the Modelica class definitions that do not use the short syntax. It can be a `package`, a `class`, a `function` or a `restricted class` definition. The `RegularClassDefinition` is composed of zero or one `ElementList`, zero or one `EquationPart`, and zero or one `AlgorithmPart`. If a `RegularClassDefinition` refers to a `package` or `record`, it may only have an `ElementList`. If it is a `function`, it may have a `ElementList` and an `AlgorithmPart`. If it is a `class`, it may have an `ElementList` and an `EquationPart`. A `RegularClassDefinition` can be empty, none of its components is mandatory.

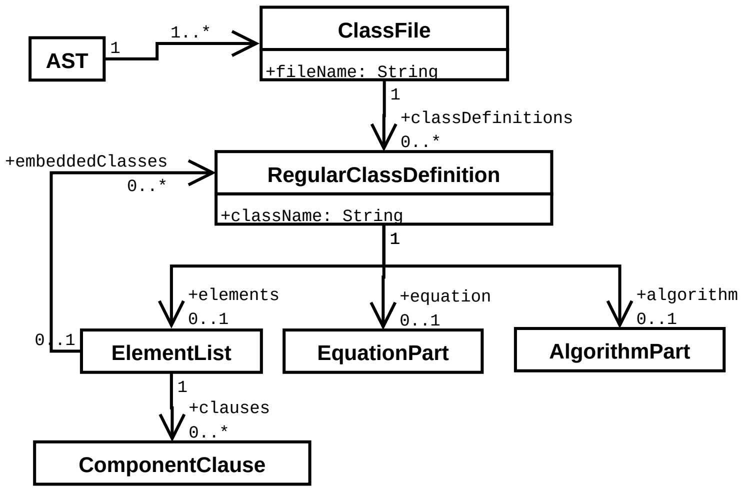


Figure 4.4: The Modelica Abstract Syntax Tree Structure

In Modelica, embedded class definitions are allowed, which means one class can be defined inside another class definition. So the `ElementList` may have zero or many `RegularClassDefinitions` and zero or many `ComponentClauses`. `ComponentClause` is also a general term. Basically, it means all the Modelica declarations. It can be a variable declaration, a parameter declaration, and so forth. The `EquationPart` and `AlgorithmPart` are the constructs for model behaviour description in Modelica. Equations are used to specify equality relations among state variables for continuous models. Algorithms are used to describe programming language features of a model.

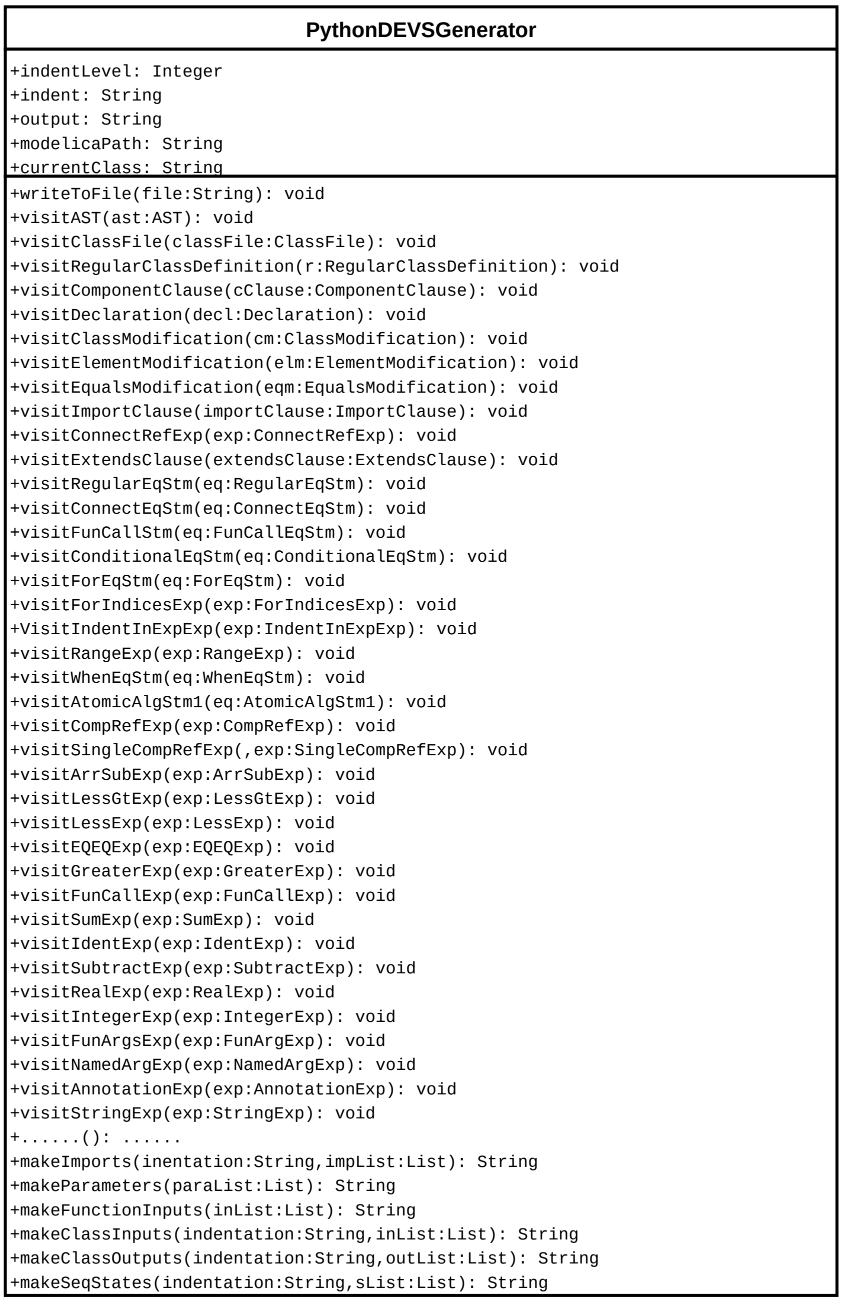
For space reasons, we only present these components in a μModelica AST. More information can be found in [Xu05].

The Python DEVS Code Generator

Once we have a model's AST, the task of code generation for the model is just to generate code for each node of the AST and put code for all the nodes of the AST together following the grammar of the target language.

Figure 4.5 is the class diagram of the Python DEVS code generator. It is a visitor class to the AST, so it has corresponding visiting methods to the nodes of the AST. Because we do not support all the Modelica language features, it only includes the visiting methods corresponding to constructs related to DEVS Modelica representation.

Obviously, the responsibility of each visiting method is to generate code for the corresponding AST node. On the bottom of Figure 4.5, there are some non-visiting methods. The reason of the existence of non-visiting funcitons is related to the different positions of some constructs in the source language and target language. Most of the Modelica language features can be translated into corresponding Python DEVS constructs following the order they are visited by the visitor. However, for some other Modelica features, like the import statement, input and output parameters, the positions of their Python counterparts are significantly different. For these language features, the code are generated by the methods beginning with the prefix ``‘‘make’’``. Because the visiting methods' meanings are obvious, let us introduce the methods that begin with the ``‘‘make’’`` prefix.



PythonDEVSGenerator

```
+indentLevel: Integer
+indent: String
+output: String
+modelicaPath: String
+currentClass: String
```

```
+writeToFile(file:String): void
+visitAST(ast:AST): void
+visitClassFile(classFile:ClassFile): void
+visitRegularClassDefinition(r:RegularClassDefinition): void
+visitComponentClause(cClause:ComponentClause): void
+visitDeclaration(decl:Declaration): void
+visitClassModification(cm:ClassModification): void
+visitElementModification(elm:ElementModification): void
+visitEqualsModification(eqm:EqualsModification): void
+visitImportClause(importClause:ImportClause): void
+visitConnectRefExp(exp:ConnectRefExp): void
+visitExtendsClause(extendsClause:ExtendsClause): void
+visitRegularEqStm(eq:RegularEqStm): void
+visitConnectEqStm(eq:ConnectEqStm): void
+visitFunCallStm(eq:FunCallEqStm): void
+visitConditionalEqStm(eq:ConditionalEqStm): void
+visitForEqStm(eq:ForEqStm): void
+visitForIndicesExp(exp:ForIndicesExp): void
+VisitIndentInExpExp(exp:IndentInExpExp): void
+visitRangeExp(exp:RangeExp): void
+visitWhenEqStm(eq:WhenEqStm): void
+visitAtomicAlgStm1(eq:AtomicAlgStm1): void
+visitCompRefExp(exp:CompRefExp): void
+visitSingleCompRefExp(,exp:SingleCompRefExp): void
+visitArrSubExp(exp:ArrSubExp): void
+visitLessGtExp(exp:LessGtExp): void
+visitLessExp(exp:LessExp): void
+visitEQEQExp(exp:EQEQExp): void
+visitGreaterExp(exp:GreaterExp): void
+visitFunCallExp(exp:FunCallExp): void
+visitSumExp(exp:SumExp): void
+visitIdentExp(exp:IdentExp): void
+visitSubtractExp(exp:SubtractExp): void
+visitRealExp(exp:RealExp): void
+visitIntegerExp(exp:IntegerExp): void
+visitFunArgsExp(exp:FunArgExp): void
+visitNamedArgExp(exp:NamedArgExp): void
+visitAnnotationExp(exp:AnnotationExp): void
+visitStringExp(exp:StringExp): void
+......(): ......
+makeImports(inentation:String,impList:List): String
+makeParameters(paraList:List): String
+makeFunctionInputs(inList:List): String
+makeClassInputs(indentation:String,inList:List): String
+makeClassOutputs(indentation:String,outList:List): String
+makeSeqStates(indentation:String,sList:List): String
```

Figure 4.5: The Class Diagram of Python DEVS Generator

makeImport In Modelica, there are two ways to map a package to the storage of a computer file system, mapping a package to a directory and mapping a package to a file. If a package is mapped to a directory, the package name becomes the directory name, and classes inside the package are saved as individual files with extention name ``mo'' under the directory. If a package is mapped to a file, then the package name becomes the file name, and all class definitions inside the package are saved in the same file. For example, we have a package ``modelica.msdl.devs'' package. For the first method, it is mapped to the ``$MODELICAPATH/modelica/msdl/devs/'' directory, under which each class definition in the package is saved as an individual file, such as AtomicDEVS.mo, CoupledDEVS.mo and so forth. For the second approach, the package is mapped to ``$MODELICAPATH/modelica/msdl/devs.mo''. In our implementation, we only support the second way of package mapping. The MODELICAPATH is a environment variable for Modelica, which includes predifined paths for Modelica libraries.

When you want to use definitions in other packages, there are four ways in Modelica to import resources in other packages [Fri04]. They are shown as below.

1. import <packagename>; //qualified import
2. import <packagename>.<definitionname>; //single definition import
3. import <packagename>.*; //unqualified import
4. import <shortpackagename> = <packagename>; //renaming import

The packagename here is the fully qualified name of the imported package including possible dot notation. Due to time limitation, we do not support the full import semantics in the DEVS Modelica compiler. Only the second and third options above are supported.

In Python, there are two forms of syntax for importing modules. One is similar to Modelica's qualified import, which imports the module name into the current namespace. The syntax is ``import modulename''. The other is in the form of ``from modulename import definitionname or *''. The first form of import can only import the module name into the importing namespace. The second one can implement the semantics of Modelica's single definition import and unqualified import. However, the restriction on the second form import in Python is that this kind of import can only appear at the beginning of a source file before any definition or declararation. Modelica's import statement has no such constraint. So the task of the makeImports method of the PythonDEVSGenerator is to collect all the import statements in a Modelica file, translate them into Python's second form of import statements and put them at the beginning of the target Python DEVS file.

makeParameters In Modelica, parameter declarations for a class follow the same way as class variable declarations. The only difference is that parameter declarations are modified by the parameter keyword. While in Python, parameters for a Python class are declared as the parameters of the class's constructer method __init__. So the makeParameters method's responsibility is to translate parameters of a Modelica class to the parameters of the __init__ method of a corresponding Python class. Below is an example of translating Modelica class A's parameters p1, p2 into parameters of corresponding Python class A's __init__ method.

```
Modelica Class:

class A
    parameter Integer p1;
    parameter Real p2;
    ......
end A;

Python Class:

class A:

    def __init__(p1, p2):
        ......

    ......
```

makeFunctionInputs In Modelica, functions are also classes. Input parameters of a Modelica function are specified by modifying a variable declaration with the `input` keyword. While in Python, declarations of function parameters are different from declarations of a class's variables. So the task of the `makeFunctionInputs` method is to translate the input parameters of a Modelica function into parameters of a corresponding Python function. Below is an example.

```
Modelica function:

function foo
    input Integer a;
    input Integer b;
    ......
end foo;

Python function:

def foo(a, b):
    ......
```

makeClassInputs and makeClassOutputs As we mentioned earlier, we use Modelica input and output keywords with transformed meanings to declare input and output ports in the atomic and coupled DEVS models. These port delcarations are translated into Python DEVS's `addInputPort` function and `addOutputPort` funciton respectively. This task is done by the `makeClassInputs` and `makeClassOutputs` method. As indicated by their names, `makeClassInputs` is responsible for translating input port declarations in Modelica into Python DEVS's addInput-Port and `makeClassOutputs` is responsible for translating output port declarations.

4.4 Case Study

We have discussed issues of representing DEVS models in Modelica and how to translate Modelica representations into Python DEVS models. Now let us see a real example of Modelica DEVS model representation.

Our example is to model an event generator that produces an output event after a random time interval. The model is represented by two Modelica classes, `GeneratorState` and `Generator`. As indicated by the names, the `GeneratorState` class represents the model state, and the `Generator` class describes the model. The model has two sequential states, `G_IDLE`, and `G_GENERATING`. This is represented by the enumeration type `SeqStates` in the `Generator` class. Corresponding to the two sequential state, `SeqStates` has two enumeration values. Also the `Generator` class has a variable `state` of type `GeneratorState`. The `GenerateState` class has a `seqState` variable, which in fact is used at simulation time to reflect the behaviour of the sequential state. Below is the Modelica representation of the `Generator` model.

```
class GeneratorState
    extends DEVSState;
    Generator.SeqStates seqState(start=Generator.SeqStates.G_IDLE);
end GeneratorState;



class Generator
    extends AtomicDEVS;
    parameter Integer ia=0;
    parameter Integer ib=0;
    parameter Integer szl=0;
    parameter Integer szh=0;
    output DEVSPort g_out;
    GeneratorState state();
    type SeqStates = enumeration(G_IDLE, G_GENERATING);

    function intTransition
    algorithm
        if ( state.seqState == SeqStates.G_IDLE ) then
            state.seqState := SeqStates.G_GENERATING;
        elseif ( state.seqState == SeqStates.G_GENERATING ) then
            state.seqState := SeqStates.G_IDLE;
        end if;
    end intTransition;
    function outputFnc
        DEVSEvent evt = null;
```

```
continue ...

    algorithm
        if ( state.seqState == SeqStates.G_GENERATING) then
            evt := Job(szl, szh);
            poke(g_out, evt);
        end if;
    end outputFnc;
    function timeAdvance
        output Integer timespan;
    algorithm
        if ( state.seqState == SeqStates.G_IDLE) then
            timespan:=randint(ia, ib);
        elseif ( state.seqState == SeqStates.G_GENERATING) then
            timespan := 0;
        end if;
    end timeAdvance;

end Generator;
```

The DEVS Modelica compiler takes the two Modelica classes above and generates the Python DEVS representation below. Corresponding to the model and model state classes in Modelica, two Python classes with the same name `GeneratorState` and `Generator` are generated. Besides the elements described in `Modelica` models, two extra functions `__str__` and `toXML` are generated for the Python DEVS classes. These are two utility functions for generating model state information when the model is simulated.

```
class GeneratorState:

    def __init__(self):
        self.seqState = Generator.G_IDLE
    def __str__(self):
        strRep = ''
        strRep = strRep + "\nseqState: " + str(self.seqState)
        return strRep
        def toXML(self):
        strRep = ''
        strRep = strRep + "\n<attribute category=\"P\">"
        strRep = strRep + "\n\t<name>seqState</name>"
        strRep = strRep + "\n\t<type>Generator.SeqStates</type>"
        strRep = strRep + "\n\t<value>"+str(self.seqState)+"</value>"
        strRep = strRep + "\n</attribute>"
        return strRep
```

```
class Generator( AtomicDEVS ):
    G_IDLE = 'G_IDLE'
    G_GENERATING = 'G_GENERATING'

    def __init__(self, ia, ib, szl, szh, name):
        AtomicDEVS.__init__(self, name)
        self.ia = ia
        self.ib = ib
        self.szl = szl
        self.szh = szh
        self.name = name
        self.g_out = self.addOutPort("g_out")
        self.state = GeneratorState()
    def intTransition( self ):
        if (self.state.seqState == Generator.G_IDLE):
            self.state.seqState = Generator.G_GENERATING
        elif (self.state.seqState == Generator.G_GENERATING):
            self.state.seqState = Generator.G_IDLE
        return self.state
    def outputFnc( self ):
        evt = None
        if (self.state.seqState == Generator.G_GENERATING):
            evt = Job(self.szl, self.szh)
            self.poke(self.g_out, evt)
    def timeAdvance( self ):
        if (self.state.seqState == Generator.G_IDLE):
            timespan = randint(self.ia, self.ib)
        elif (self.state.seqState == Generator.G_GENERATING):
            timespan = 0
        return timespan
    def __str__(self):
        strRep = ""
        strRep = strRep + "\nstate: " + str(self.state)
        return strRep
    def toXML(self):
        strRep = ''
        strRep = strRep + "\n<attribute category=\"C\">"
        strRep = strRep + "\n\t<name>state</name>"
        strRep = strRep + "\n\t<type>GeneratorState</type>"
        if (self.state!=None):
            strRep = strRep + "\n\t<value>"+self.state.toXML()+"</value>"
        else:
            strRep = strRep + "\n\t<value>None</value>"
        strRep = strRep + "\n</attribute>"
        return strRep
```

4.5 Conclusions

In this chapter, we first briefly introduced the object-oriented modelling language Modelica. Then we discussed issues of representing DEVS models in Modelica. Then we introduced the Modelica model compiler, which compiles the Modelica DEVS representation into Python DEVS. Subsequently, we presented a case study to see how we represent a real DEVS model in Modelica, and introduced the generated Python DEVS representation from the Modelica model representation.

From this chapter, we can draw the following conclusions. First, the way of representing DEVS models in Modelica, and then tanslating the Modelica model representation into simulator-specific model representation is feasible. Second, Modelica is a suitable choice for high-level model description, and it has enough features for describing DEVS models. Third, the model compiler plays a important role in the model transformation process. It can not only check the correctness of model syntax, but also generate utility functions (such as trace output) to relieve the burden of modellers.

5 Visual DEVS Modelling Environment

5.1 Introduction

Research and application of visual modelling technology in the software engineering industry has made great progress in recent years. Compared to hand written textual model specification, visual models are more intuitive and easier understood. Visual model representations can be used not only as model specifications but also as effective tools for communication and collaboration. Furthermore, visual models can effectively reflect the interactions and the structural relations among different models. It makes model analysis and design more effecient.

One of the key issues of visual modelling is how to specify the visual modelling rules, which means how to specify constraints, space arrangements, and connections on visual constructs. In textual languages, grammars are used to describe language syntax. For visual languanges, meta-models play a similar role as grammars in textual languages. The idea is to build a meta-model for a modelling formalism, in which the formalism's visual syntax is specified. When one create models using that formalism, one must follow the rules specified by the meta-model of the formalism.

For DEVS, some research has been done on DEVS visual modelling. [PP93] proposes ideas on visual modelling DEVS behaviour using higraphs. [PBV03] demonstrates the possibility of visual modelling DEVS in AToM^3, and implements a code generator that can generate Python-DEVS code from visual DEVS models. [Dub06] uses meta-modelling and State-charts to build visual modelling environments for DEVS, in which both the DEVS formalism and the visual modelling environment are modelled. The DEVS formalism is modelled by Entity Relation (ER) diagrams and the modelling environment is modelled by State-charts.

AToM^3 is a multi-formalism modelling and meta-modelling environment. In AToM^3, formalisms are modelled by a common meta-meta-model, ER diagram. Based on the common meta-meta-model, models specified by one formalism can be easily transformed into models represented in another formalism. The meta-modelling technology used in AToM^3 can specify both the visual grammar for a formalism and visual appearance of components of the formalism. After building a meta-model for a formalism, the AToM^3 meta-model compiler takes the meta-model as input, and generates a visual modelling environment for that formalism. The operations that can be executed in the environment are specified by State-charts.

In this chapter, we look into an enhanced version of the visual modelling enviroment of [PBV03]. The enhancements have been done mainly in the following ways. First, at the meta-model level, the new version presents the model instance concept, which means that, for a coupled DEVS model, its sub-models are instances of defined DEVS models. This enhancement makes DEVS visual models more compact and easy to understand. Second, new components, event and model state, have been modeled in the DEVS meta-model. And finally, the Modelica DEVS code is generated from the visual models rather than code for Python DEVS.

The organization of this chapter is as following. In section 1, we discuss modelling and meta-modelling in ATоM^3. In section 2, we present the architecture of the visual modelling environment. In section 3, we discuss meta-modelling DEVS in ATоM^3. In section 4, we look into the code generator that translates visual models into Modelica model representations. In section 5, a case study demonstrates how to use the visual modelling environment. Finally, in section 6, we draw conclusions for this chapter.

5.2 Modelling and Meta-Modelling in AToM^3

We have discussed modelling and meta-modelling technologies in chapter 2. A meta-model is a model for a modelling formalism. There are two common purposes for using meta-modelling technology. First, a meta-model specifies the syntax of a modelling language. So the meta-model can be used to generate the components of the modelling language and check whether a generated model is a syntatically valid model for the modelling language that is defined by the meta-model. This feature of meta-modelling is usually used to build visual modelling environments. By building a visual meta-model for an modelling language, the modelling environment system can dynamically check whether the visual syntax is correct while one is drawing a visual model. Second, meta-modelling can facilitate the process of model transformation. Model transformation means transforming models specified in one formalism to corresponding models, posibly described in another formalism. A complex dynamic system may contain diverse components. At the modelling level, each component may be modelled in a different formalism. However, at the simulation level, in order to study and analyze the system as a whole, it may be required to transform the diverse models specified in different formalisms into models described in one formalism that is suitable for simulation. When it comes to model transformation, a key issue is how to find an appropriate construct in the target formalism to represent a component in the source formalism. A effective way to solve this problem is to meta-model all the formalisms involved using another formalism. Since all the formalisms involved in the transformation are meta-modelled by the same formalism, and components of all the formalism can be generated from the same meta-modelling language, finding matched components in a target formalism to represent constructs in source formalisms will be not a problem.

These two advantages of meta-modelling inspire us to use meta-modelling technology to build a visual modelling environment for DEVS. The benefits of using meta-modelling in building such an DEVS modelling environment are obvious. Firstly, meta-modelling provides the capability of model checking at visual modelling level. Because the visual DEVS modelling environement is not a free-drawing, we need to add some rules and constraints to enforce the visual DEVS syntax. This can be perfectly done by meta-modelling DEVS. Secondly, DEVS has been envisioned as a generic formalism suitable for both modelling and simulation. Many modelling formalisms that are not suitable for simulation can be tansformed to DEVS and then be simulated. Meta-modelling for DEVS can make the transformation easier.

In this chapter we will discuss building a visual modelling environment for DEVS formalism using the meta-modelling technology in AToM^3. Before we build a meta-model for DEVS, let us first look at how meta-modelling is done in AToM^3.

AToM^3 (A Tool for Multi-formalism Meta-Modelling) is a interactive tool for building meta-models. AToM^3 provides not only the facilities for specifying the grammar of a modelling language, but also the utilities for defining the appearance for constructs of the modelling language. At the meta-meta-modelling level, the modelling language used in AToM^3 is the ER (Entity Relationship) formalism with extended constraints. The core part of AToM^3 is a meta-model processor. When an instance of AToM^3 starts, the meta-model processor loads its own ER model to bootstrap the system. After it is ready, it provides a visual modelling (or meta-modelling) environment, in which one can build and manipulate ER meta-models for other formalisms. Also one can load meta-models for other formalisms to do multi-formalism modelling and meta-modelling.

When modelling using the ER formalism at the meta-meta-level, the modelled object and its features are specified by entities with attributes. AToM^3 can specify two kinds of attributes, regular and generative [VdL05]. Regular attributes are used to specify the characteristics of the components at meta-model level modelled by the current meta-meta-level ER formalism. Generative attributes are used to specify the properties of constructs at model level that are modelled by the currently defined meta-model. If the model level model is used further to describe other models, it will have its own generative attributes. Besides specifying meta-model attributes, it is also possible to specify the graphical appearance at the meta-meta-level for each entity of the lower meta-level. This graphical appearance is in fact a special kind of generative attribute, which specifies how instances of meta-level components and their attributes will be visually displayed. This feature provides AToM^3 the capability of supporting visual model manipulation at the lower meta-level.

In order to fully specify a modelling formalism, the meta-meta-level language must have the capability of specifying not only the structural and graphical notations, but also the constraints and operations. In AToM^3, the ER formalism has been extended to work with the Object Constraint Language (OCL) and Python functions. ER is mainly used to express structural and graphical notations. Constraints and actions are described by OCL or Python functions.

Two major steps one needs to go through for building a new modelling environment for a specific formalism in AToM^3. First, build a ER meta-model for the formalism, in which you specify the structure, visual apperance, and constraints for the components of the formalism. Second, load the State-charts formalism and build a State-chart specifying the behaviour for visual manipulation of entities in the formalism. After these two steps, the AToM^3 meta-modelling processor generates a visual modelling environment. In the newly generated modelling environment, one can build models using the specific formalism graphically.

5.3 Design and Implementation

5.3.1 Architecture

The architecture of the DEVS visual modelling environment is given in Figure 5.1. First, we build a DEVS meta-model, in which DEVS components are specified. Then a GUI behaviour model is defined, in which the operations that can be executed on visual DEVS model components are specified. After that, the ATоM^3 meta-model processor generates a visual DEVS modelling envionment.

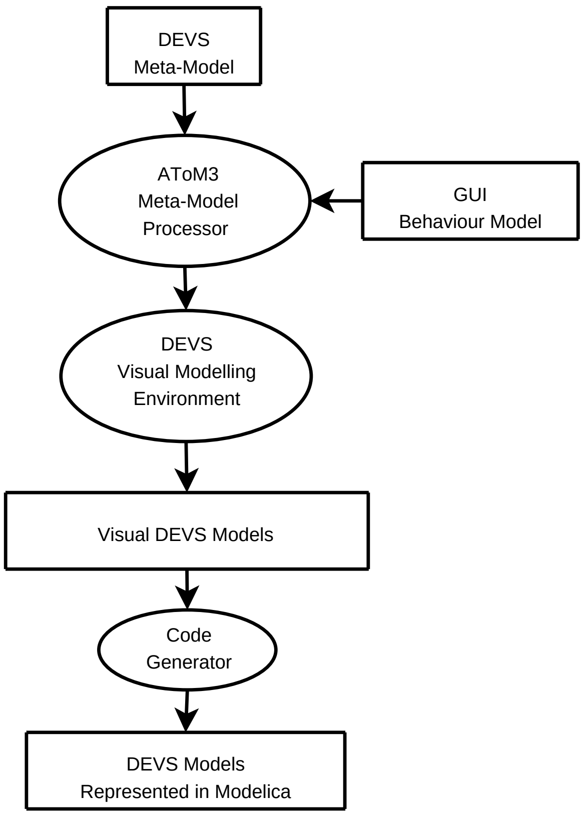

DEVS Meta-Model
AToM3 Meta-Model Processor
GUI Behaviour Model
DEVS Visual Modelling Environment
Visual DEVS Models
Code Generator
DEVS Models Represented in Modelica

Figure 5.1: Architecture of DEVS Visual Modelling Environment

In the generated visual modelling environment, DEVS models can be drawn graphically. At this time, the visual models are represented as ASG (Abstract Syntax Graph) objects in Python. In the graph, each component is represented as a graph node. The code generator's task is to travel the ASG graph, and translate this model representation into a Modelica representation mentioned in chapter 4.

5.3.2 Meta-Modelling DEVS in AToM^3

Figure 5.2 is the meta-model for DEVS. Meta-models in AToM^3 are represented as ER diagrams. In this figure, the "V2" following entity and relation names means version 2 of the DEVS meta-model. The ER model in Figure 5.2 can be understood as follows. We begin from atomic models.

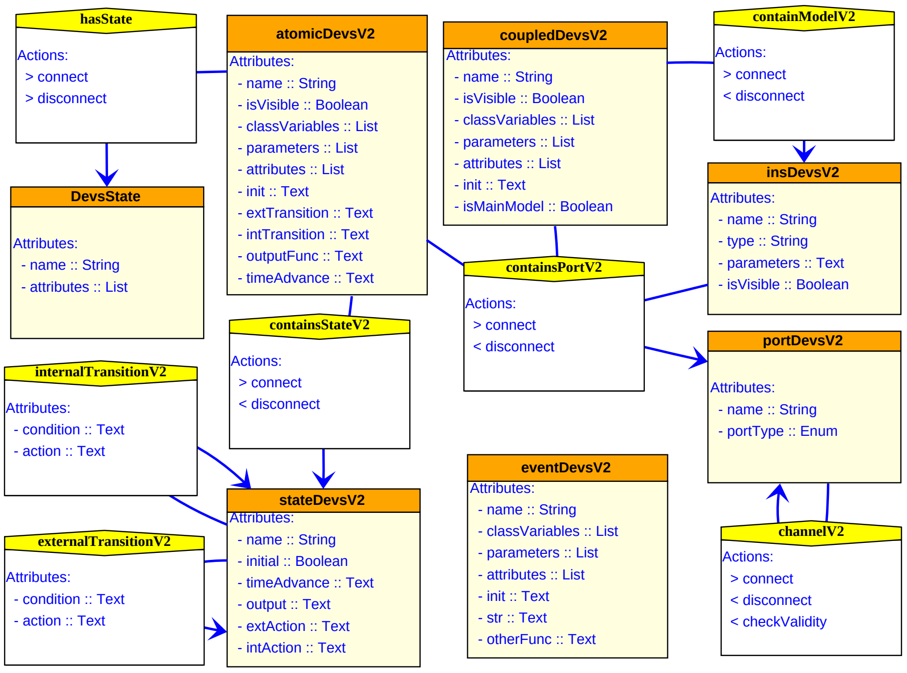


Figure 5.2: DEVS Meta-Model

Atomic DEVS Model For atomic DEVS we model the following properties. Each atomic model has a `name` of string type. This `name` is not the fully qualified name mentioned above. It is just a nomal string name, which will be used to instantiate the atomic model. An atomic DEVS may have a list of class variables. Here the class variable has the same meaning as that in common object-oriented langages. The class level variables can be accessed using class (model) names without instantiation of the class (or model). Parameters models the values needed to initialize a model instance. Attributes represents the state variables of an atomic model. Besides properties inside the entity rectangle, there are two other properties described by the relation arrow, `stateDevsV2` and `portDevsV2`. `stateDevsV2` represents the sequential states of an atomic model, and `portDevsV2` describes the input and output ports of an atomic DEVS model. Each atomic DEVS has one or more sequential states and one or more ports.

Sequential State As one of the most important concepts in DEVS, the Sequential State (stateDevsV2 in Figure 5.2) has been modelled as follows. Each sequential state has a name, which is used to identify it. The name of the sequential state is unique inside an atomic DEVS model. A sequential state has a `output` property, which specifies what will be sent out and which port the output should be sent to. The `timeAdvance` attribute describes how long a model can stay in this sequential state. The `extAction` and `intAction` properties specify the common actions that needs to be taken before the model transit from this sequential state to another sequential state. There are two ways to transit a sequential state to another sequential state, via internal transition or external transition. Internal transitions are triggered by internal events, and external transitions are triggered by external events. The model behaviour of transition from one sequential state to another is modelled by the relation between two sequential states, `internalTransitionV2` and `externalTransitionV2`. Both of these relations have two properties, `condition` and `action`. `Condition` specifies in which circumstances the internal or external transition occurs, and `action` specifies the operations that need to be done before the transition.

Port Each DEVS model may have one or more ports, by which the model can receive external events and send out its output events. Ports are modelled by entity `portDevsV2` in Figure 5.2. A port has a `name` and a `type`. The `name` is a string used to identify the port. And the `type` indicates that the port is an input or output port.

Event Entity `devsEventV2` models the input and output events in DEVS models. An event has a `name` and a list of `attributes`. The `name` is used to identify the type of event instances. `Attributes` specify the properties of an event. Each attribute is a `<name, type, value>` triple. In order to make it possible to provide instantiation information at visual model level, an event is modelled with two extra properties, `classVariables` and `parameters`, where the `classVariables` represents an event's type-level properties, and the `parameters` means parameters needed to initialize an event instance.

Coupled DEVS Model As we mentioned in chapter 1, a coupled DEVS model is composed of a set of sub-models, and sub-models can be either coupled models or atomic models. A coupled DEVS model is specified by `coupledDevsV2` in Figure 5.2. Like atomic models, each coupled model has a `name`, a list of `parameters`, `classvariables`, and `attributes`, which have the same meanings as those in `atomicDevsV2`. The fact that each coupled model may have one or more sub-models is specified by the `containModelV2` relationship. This relationship starts from a `coupledDevsV2` entity and ends at `insDevsV2`. The constraints (which are not shown on the figure.) on this relationship specify that each `coupledDevsV2` entity contains one or more `insDevsV2` entities. The `insDevsV2` entities here actually represent sub-models. Why we use instances rather than sub-models themselves will be discussed later. Besides sub-models, a coupled DEVS model may or may not contain ports. This is specified by the `containsPortV2` relationship.

Model Instance For large modelling and simulation projects, model reuse plays a important roles at both design and implementation level. Because model reuse can not only save money and time, by using existing tested standard models, it can also improve a system's scalability and compatibility. In the visual DEVS modelling environment, we use the model instance concept to support model reuse.

In DEVS specification, each coupled model can have both coupled or atomic sub-models. There are two basic possiblities of reusing sub-models. First, the sub-models have been defined, tested and used in other projects. Second, the sub-models are not defined, but you will use the sub-models many times in your current projects. For the first situation, reproducing existing models is certainly not a effecient way of working. For the second situation, redrawing or redefining the same model many times is also not a good way of working. So we use the model instance concept in the DEVS meta-model to facilitate the model reuse process in DEVS visual modelling.

The model instance concept is represented by the `insDevsV2` entity in Figure 5.2. Each `insDevsV2` entity has three attributes, `name`, `type`, and `parameters`. `Name` is used to identify a model instance; `type` specifies which model the instance is of; and `parameters` defines parameters that are needed to initialize this instance. With the support of the model instance, model reuse becomes very easy in DEVS visual modelling. For both situations mentioned above, a model only needs to be defined once. When one uses it for coupling, one only needs to specify a model instance and couple it with other model instances or the current coupled model that you are building. Because model coupling is done through port connections, `insDevsV2` also has ports, and the ports have the same defintions as they have in the `atomicDevsV2's` model definition. The fact that a visual model instance may also have ports is specified by the relationship `containPortV2` between the `insDevsV2` entity and `portDevsV2` entity.

This practice of using model instances for DEVS model coupling can bring two main benefits. First, it facilitates model reuse and hence saves time and money for model building. Second, because the definition and application of a model are separated, the hierarchical structure of visual DEVS models becomes very clear and easily understood.

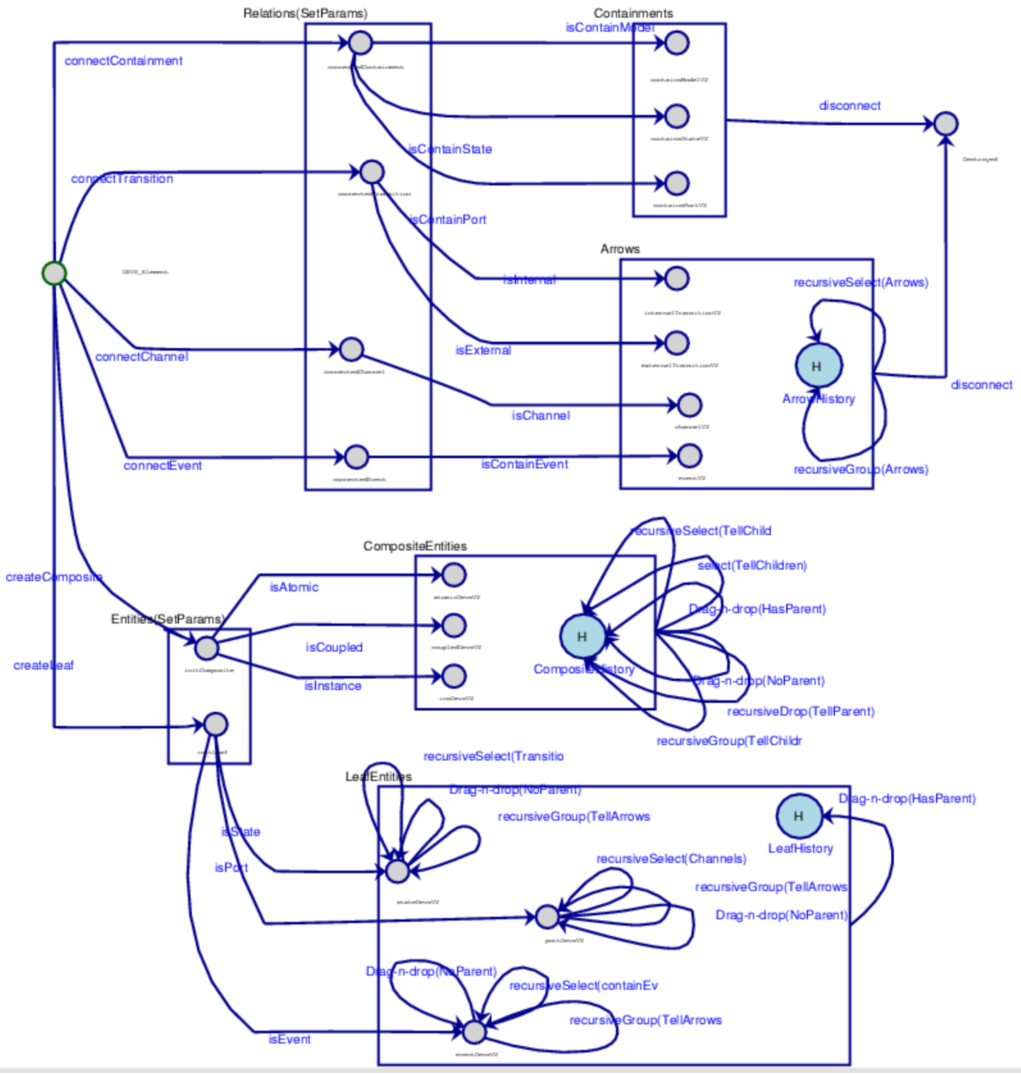


Figure 5.3: State Chart for DEVS Visual Modelling GUI Interface

5.3.3 User Interface Model - State-chart

The design principle of AToM^3 is "Modelled everything". In AToM^3, not only are all the formalisms and the system kernel modelled, the behaviour of the graphical user interface of AToM^3 is also modelled. The behaviour model is specified in the State-chart formalism. The idea is that there is a common model in State Chart that specifies all the common visual constructs manipulation behaviour in AToM^3. When you build a modelling environment for a new formalism, if the visual modelling behaviour is covered by the common State Chart in AToM^3, then everything is fine. If there are some new operations in the newly created visual modelling environment that are not covered by the common State-chart, one needs to build a State-chart to specify the new behaviour. The formalism-specific State-chart can communicate with the common State-chart, and they work together to specify the graphical manipulation behaviour of the newly created modelling environment.

Figure 5.3 is the State-chart for DEVS visual modelling environment. This State-chart is originally created by Denis Dube. It is modified for supporting the enhanced functionalities.

5.4 Code Generator: from Visual Model to Modelica

We have discussed how to represent DEVS models in Modelica in chapter 4. In this chapter, we look into how to translate the visual DEVS models mentioned above into Modelica representation.

5.4.1 Mapping Visual Model Components to Modelica Representation

In chapter 4, we have discussed the Modelica DEVS constructs. We have defined Modelcia prototypes for atomic DEVS models, coupled DEVS models, DEVS states, DEVS events, and DEVS ports. So it is very easy to find corresponding Modelica representations for the entities `atomicDevsV2`, `coupledDevsV2`, `DevsState`, `eventDevsV2`, and `portDevsV2` defined in Figure 5.2. There are two other entities `insDevsV2` and `stateDevsV2` in Figure 5.2. For a coupled DEVS model, `insDevsV2` just means the instance of an existing model. This means that when we do model coupling, it is the model instances, not the model types that are coupled together. So the `insDevsV2` entities are translated into model instantiations. For an atomic DEVS Modelica representation, the sequential states are described as elements of an enumeration type. In the visual meta-model, each atomic DEVS may have one or more sequential states. So the name of each `stateDevsV2` instance can be translated into an element of the `SeqStates` enumeration. The behaviour of a model is reflected on the other properties of the `stateDevsV2` entities.

5.4.2 Atomic DEVS Code Generator

In Chapter 4, we have defined a prototype for atomic DEVS models. Basically, the Modelica representation of an atomic DEVS model looks like this.

```
class ModelName
      extends AtomicDEVS;
      parameter p1, p2, ....;
      ModelState state;
      type SeqStates=enumeration(sq1, sq2, ...);
      input inPort1, inPort2, ....;
      output oPort1, oPort2, ....;
      function externalTransition
      end externalTransition;
      function internalTransition
      end internalTransition;
      function timeAdvance
      end timeAdvance;
      function outputFunction
      end outputFunction;
end ModelName
```

So the task of generating Modelica representation from the visual model is to get the corresonding parts of the prototype from the visual DEVS model. From Figure 5.2, we can get Figure 5.4, which includes all the components that are related to an atomic DEVS model.

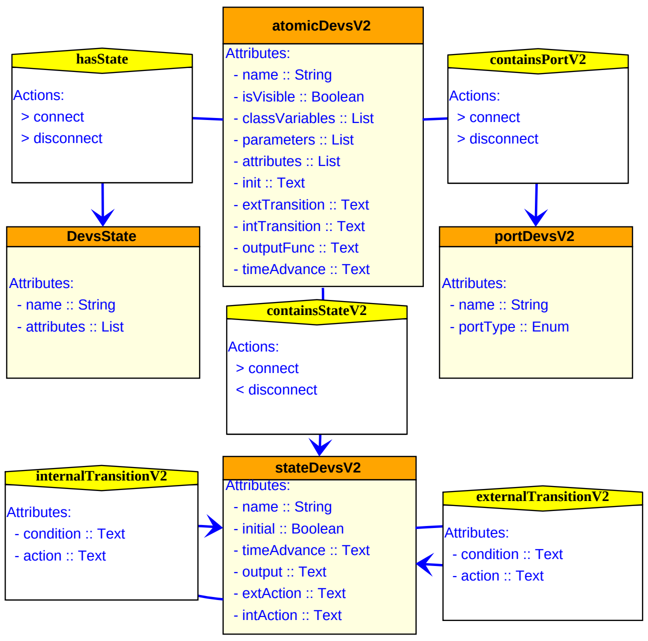


Figure 5.4: Meta-Model for Atomic DEVS Model

Based on Figure 5.4 and the prototype of Modelica representation for an atomic DEVS, let us see how we can get the Modelica componets from the properties of the atomic DEVS meta-model.

Model Name From Figure 5.4 we can see that in the `atomicDevsV2` entity, there is a name attribute, which specifies that each atomic DEVS model built from this meta-model has an attribute name. This name is the name of the newly created model's name. Given a newly created visual atomic model `atmDEVS`, a model's name is translated into Modelica representation as below.

```
class atmDEVS.name
    ......
end atmDEVS.name;
```

For example, if the value of the `atmDEVS's` name attribute is "`Processor`", the code like below will be generated.

```
class Processor
    ......
end Processor;
```

Parameters There is a *`parameters`* attribute in the `atomicDevsV2` entity. The `parameters` is a list of AToM^3 `Attribute` type. AToM3 `Attribute` is a generative type, which is usually used to define the properties of models that are created from a meta-model. Here we use the `Attribute` to describe the initialization parameters for a model. An instance of the AToM^3 `Attribute` type has three member variables, `name`, `type`, and `initial value`. So the parameters of the `atmDEVS's` Modelica representation can be generated using the following template. For each parameter `param` in `atmDEVS's` parameters we can get a Modelica statement such as,"`parameter param.type param.name = param.initValue;`". For example, if a visual model has an parameter `num` with type of Integer and initial value 1. It is translate to "`parameter Integer num = 1;`" in Modelica.

Model State In DEVS Modelica representation, we define a prototype class DevsState. Each atomic DEVS's Modelica representation has a `state` attribute, which is of type `DevsState`. The `DevsState` entity in Figure 5.4 means the same thing as the predefined `DesvState` class in DEVS Modelica. We separate the model state variables from an atomic DEVS model and represent them in a specific entity. This separation makes it very clear to see in the visual model which part reflects a model's state and which part reflects a model's behaviour. A DEVS model state is very simple. It has a `name` and a list of `attributes`. The `name` is simply a string that identifies the state. The `attribute` is of type generative AToM^3 `Attribute` type, which is further composed of `type`, `name`, and `initial value`. In Modelica representation, the model state is represented by a Modelica class that extends from the predefined `DevsState` class. The `name` of the `DevsState` entity is translated into the Modelica class name, and the `attributes` are translated into class properties. Besides the explicitly specified model state `attributes`, in DEVS Modelica, each model state has an attribute `seqState` of type `SeqStates` that represents the current sequential state of the model. This attribute has an initial value that represents the model's initial sequential state. In visual DEVS models, an atomic model may or may not have a explicitly defined state. If an atomic model has no explicitly defined model state, a Modelica class with the name of the model's name concatenated with "`State`" is generated. The generated class has only one attribute `seqState` with the type of the model's `SeqStates`. So the Modelica representation for a model state will look like below.

```
class state.name
    extends DevsState;
    state.attributes;

end state.name;
```

Sequential States In DEVS Modelica, sequential states of an atomic DEVS model are represented by the Modelica `enumeration` type. In the meta-model shown in Figure 5.4, the visual DEVS sequential states are described as `stateDevsV2` entity. The Modelica enumeration type is composed of a set of strings. Each `stateDevsV2` entity has an attribute `name`. So the Modelica representation of a model's sequential state are generated from names of the `stateDevsV2` entities in a visual DEVS model. For example, if a visual DEVS model has two `stateDevsV2` entities, `Sa`, and `Sb`, then the Modelica representation of the model's sequential states will be "`type SeqStates = enumeration(Sa.name, Sb.name);`".

Input and Output Ports As shown in Figure 5.4, visual DEVS ports are specified by the `portDevsV2` entity. Each port has two attributes, `name` and `type`. `Name` is used to identify a port, and `type` specifies whether a port is an input or output one. In DEVS Modelica, `input` and `output` keywords are used to specify the type category of a port. And the predefined class `DevsPort` is used to indicate a variable as a DEVS port. So, for a given visual DEVS port `p`, the Modelica representation for `p` is "`p.type DevsPort p.name();`". For example, if `p` is a input port with `name` "`in`", then its Modelica representation is "`input DevsPort in();`". Similarly, if `p` is an output port named "`out`", then the Modelica representation is "`output DevsPort` out();".

Internal Transition Internal transition represents the process of an atomic DEVS model changing its sequential state from one state to another, when the time interval specified for the former state expires. In DEVS Modelica, an internal transition of a model is described by an `if else` statement in the model's internal transition function. As shown in Figure 5.4, in visual DEVS models, internal transitions are represented by `internalTransitionV2`, which actually represents the relationship between two visual sequential states. Each relationship `internalTransitionV2` has two attributes, `condition` and `action`. `Condition` specifies the condition under which the internal transition occurs, and the `action` describes the operations that will be done on the model state if that transition happpens. So the `condition` of a visual internal transition can be translated into Modelica's `if` statement's `condition` expression, and the action is the `if's` body. Because of the nature of exclusiveness of internal state transitions, only the internal transition for one sequential state can be translated to Modelica's `if` statement, and others must be translated into `elseif` statements. Besides the `internalTransitionV2` itself, the `intAction` attribute of the `stateDevsV2` entity is related to the internal transition beginning from the sequential state specified by that `stateDevsV2` entity. The `intAction` of a sequential state specifies the common operations that need to be done on the model state for all internal transitions begins from this sequential state. For example, if an atomic DEVS model has three internal transitions in its visual model. The first internal transition has an start sequential state `Sa` and an end state `Sb`, the second one starts at `Sb` and ends at `Sc` with condition `c1` and action `a1`, and the third one starts at `Sb` and ends at `Sa` with condition `c2` and action `a2`. The Modelica representation for these internal transitions will be generated like this.

```
if (state.seqState == Sa) then
    Sa.intAction;
    state.seqState :=Sb;

elseif (state.seqState == Sb) then
    Sb.intAction;
    if (c1) then
        a1;
        state.seqState := Sc;
    end if;
    if (c2) then
        a2;
        state.seqState := Sa;
    end if

end if;
```

External Transition Similar to internal transitions, external transitions are used to specify the behaviour of a DEVS model changing its sequential states when an external event occurs. In DEVS Modelica, external transitions have the same statement structure as that of internal transitions. In visual DEVS model, external transitions also have a similar visual structure as that of the visual internal transitions. The only difference is that the common operations for external transitions of a sequential state is specified by the `extAction` attribute of the sequential state. The `condition` and `action` attributes of an external transition have the same meanings as those for an internal transiton. So if we treat the example for the internal transition above as external transitions. Then we can a similar Modelica representation as below. The major difference is that this piece of code is part of the model's external transition function.

```
if (state.seqState == Sa) then
    Sa.extAction;
    state.seqState :=Sb;
elseif (state.seqState == Sb) then
    Sb.extAction;

    if (c1) then
        a1;
        state.seqState := Sc;
    end if;
    if (c2) then
        a2;
        state.seqState := Sa;
    end if

end if;
```

Time Advance Function In DEVS, the time advance function is used to specify how long a model can stay in a specific sequential state. In DEVS Modelica, the time advance behaviour for each sequential state is described by an `if` Modelcia statement. In visual DEVS models, the time advance behaviour of a sequential state is specified by the `timeAdvance` attribute of the `stateDevsV2` entity. So, for a visual DEVS sequential state `Sa`, the Modelica representation of its time advance behaviour can be generated as below.

```
if (state.seqState == Sa) then
    Sa.timeAdvance

end if;
```

Because the time advance function is used by DEVS simulators to determine the time units for staying at different sequential states, it has to return a number to indicate the time interval the model will stay for each sequential state of a model. In chapter 4, we have given a output parameter `timespan` for this special purpose. So the `timeAdvance` attribute of a visual sequential state can have a block of Modelica code, but the final result must be assigned to the `timespan` parameter.

Output Function The output function for an atomic DEVS model is used to specify the behaviour of generating an output event for the model. Similar to a model's time advance function, at different sequential states, a DEVS model may produce different output events. In the Modelica representation, the output behaviour of a model is described by an `if` statement. And in the visual model, the output is represented by the `output` attribute of the `stateDevsV2` entity. So the Modelica representation of the output behaviour for a sequential state Sa can be generated like this.

```
if (state.seqState == Sa) then
    Sa.output
end if;
```

5.4.3 CoupledDEVS Code Generator

Coupled DEVS models make hierarchical DEVS modelling possible. Figure 5.5 is the part of DEVS meta-model for describing coupled DEVS. A coupled DEVS model consists of ports, instances of other models, which can be both coupled and atomic DEVS model instances. The ports of components of a coupled model and the ports of the coupled model itself can be connected via channels.

We have discussed in chapter 4 that, in DEVS Modelica, coupled DEVS models are represented by a Modelica class that extends from predefined class `CoupledDEVS`. The `name`, `classvariables`, `parameters`, `attributes`, and `ports` can be translated into Modelica representation following the same way as that of in atomic models. Model instances can be represented as Modelica class instantiations. For example, instance `Ia` has `name` "`ia`", `type` "`ModelA`", and `parameters` "`a=2, b=3`". Then the Modelica representation for `Ia` is "`ModelA ia(a=2, b=3);`". Channels are described by the Modelica function `connect`. For example, if there is a channel between instance `Ia's` output port "`out`" and instance `Ib's` input port "`in`", then the Modelica representation for this channel is "`connect(Ia.out, Ib.in);`"

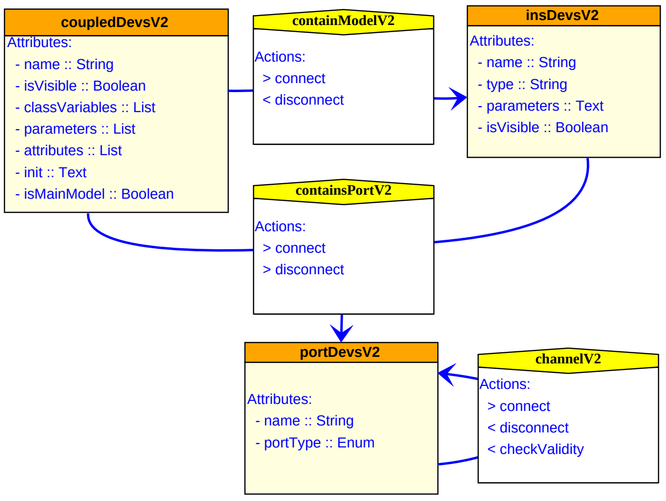


Figure 5.5: Meta-Model for Coupled DEVS Model

5.4.4 Event Generator

Figure 5.6 is the meta-model for DEVS events. In DEVS Modelica, events are represented by Modelica classes that extend from predefined Modelica class `DevsEvent`. So the Modelica representation for a visual DEVS event's attributes can be generated as the same way that for attributes of an atomic DEVS. The only difference is that the Modelica representation for events extending from `DevsEvent` class while the atomic model representation extends from `AtomicDEVS`.

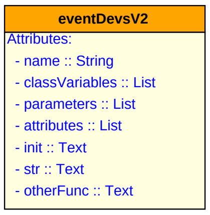


Figure 5.6: Meta-Model for DEVS Event

5.5 Case Study

We have discussed meta-modelling DEVS and Modelica code generation. Now let us see a case study. Figure 5.7 is the visual model of an atomic DEVS model named `Generator`. The `Generator` has two sequential states, `G_IDLE` and `G_GENERATING`. At the beginning, the model is in `G_IDLE` state. After a random time interval specified by `timeAdvance` attribute "`timespan:=randint(ia, ib)`", the sequential state is changed to `G_GENERATING`. At the `G_GENERATING` state, the model produces a output specified by the `G_GENERATING` state's `output` attribute, then the model sequential state changes back to `G_IDLE`. The model will continue this kind of transitions until the simulation experiment ends.

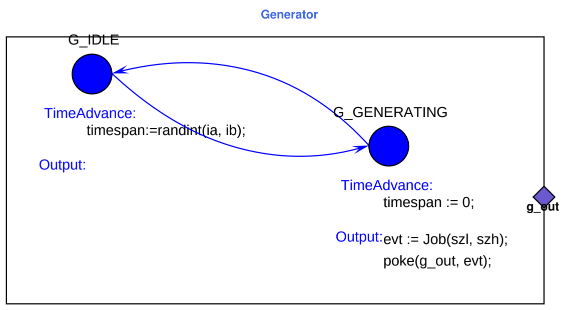


Figure 5.7: An Example of Visual Atomic DEVS Model - Generator

Below is the generated Modelica representation for the `Generator` model in Figure 5.7. The Modelica representation includes two classes, `GeneratorState` and `Generator` class. Note that, for this model, no corresponding `DevsState` entity appears. So the generated model state `GeneratorState` has only one property `seqState`, which is a variable of type `Generator.SeqStates`. The rule is that for each visual atomic DEVS model, if the model state is specified explicitly, a Modelica class with a name specified by the model state will be generated, and the `seqState` will be added as its property. If there is no explictly defined model state for a visual atomic model, a class named by the model's `name` concatenated with "`State`" is generated. And the generated class has only one attribute `seqState` with the type of the atomic model's `SeqStates`.

```
class GeneratorState
    Generator.SeqStates seqState(start=Generator.SeqStates.IDLE);
end GeneratorState;
```

```
class Generator
    extends AtomicDEVS;
    parameter Integer ia=0;
    parameter Integer ib=0;
    parameter Integer szl=0;
    parameter Integer szh=0;
    output DevsPort gout;
    GeneratorState state();
    type SeqStates = enumeration(IDLE, GENERATING);

    function intTransition
    algorithm
        if ( state.seqState == SeqStates.G_IDLE ) then
            state.seqState := SeqStates.G_GENERATING;
        elseif ( state.seqState == SeqStates.G_GENERATING ) then
            state.seqState := SeqStates.G_IDLE;
        end if;
    end intTransition;
    function outputFnc
        DevsEvent evt = null;
    algorithm
        if ( state.seqState == SeqStates.G_GENERATING) then
            evt := Job(szl, szh);
            poke(g_out, evt);
        end if;
    end outputFnc;
    function timeAdvance
        output Integer timespan;
    algorithm
        if ( state.seqState == Generator.SeqStates.IDLE) then
            timespan:=randint(ia, ib)
        end if;
        if ( state.seqState == Generator.SeqStates.GENERATING) then
            timespan:=0
        end if;
    end timeAdvance;

end Generator;
```

For space reasons, we do not present examples for coupled DEVS models here. For an example of visual modelling of coupled DEVS models, please see the case study in the next chapter.

5.6 Conclusions

AToM^3 is a tool for multi-formalism modelling and meta-modelling. In this chapter, we first discussed the mechanisms of meta-modelling in AToM^3. Then we introduced issues of meta-modelling and building a visual modelling environment for DEVS in AToM^3. Meta-models in AToM^3 can specify not only the structure for model components, but also the visual appearance, constraints, and actions of the components. Incoporated with the State-charts that specify the visual model manipulation behaviour, the meta-model can be used by AToM^3's meta-model processor to generate a visual DEVS modelling environment. In the visual modelling environment, graphical DEVS models can be easily built and be translated into neutral Modelcia representation, which will be further compiled by the compiler introduced in chapter 4.

Now we have finished the discussion of all three parts of the Infrastructure for DEVS Modelling and Experimentation. In next chapter, we will give a case study of how the Infrastructure works for the full DEVS modelling and simulation process.

6 Case Study

6.1 Introduction

In the previous chapters, we have introduced the different parts of the DEVS modelling and simulation infrastructure. In this chapter, we present a case study as an example of how to model and simulate a system using this infrastructure.

Our example is to model and simulate a chained processor system [BV02]. This system includes a series of processors connected together. The processors can process jobs. Each processor has a queue, which can hold a limited number of jobs waiting to be processed. Each processor has three ports, an “in” port, an “out” port, and a “discard” port. Jobs enters a processor through the “in” port. If the processor is busy, the newly arrivng job is put into the processor’s queue. If the queue is full, the new job is discarded through the “discard” port. If the processor is not busy when a new job arrives, the job is processed immediately. The inter-arrival-time of jobs is uniformly distributed. It takes time for the processer to process a job. How long it takes depends on the size of the job. A chained processor system is composed of a number of such processors connected sequentially.

The organization of this chapter is as follows. In the first section, we discuss the design of DEVS models of the processor and the chained processors. In the second section, we demonstrate the visual models of the system built using the visual DEVS modelling envrionment. In the third section, the Modelica representations generated from the visual models are presented. In the fourth section, Python DEVS code generated from the Modelica representation is discussed. In the fifth section, the simulation trace and some plotting results using the Visual Trace Plotter are discussed. Finally, in the sixth section, we draw some conclusions for this chapter.

6.2 DEVS Models of the Chained-Processor System

As described in the system specification, there are mainly two types of components in the chained-processor system, the processor, and processors chained together. The processor can be described as an atomic DEVS model, and the chained processor can be specified as a Coupled DEVS model. In order to simulate the chained processor system, we need another DEVS model which simulates the process of generating jobs. We call this model `Generator`, which is also an atomic model.

We have discussed the design of the `Generator` model in chapter 4. We do not repeat it here, and we begin with the `Processor` model.

6.2.1 The Processor Model

The `Processor` has two sequential states, `P_IDLE` and `P_BUSY`. When there is no job to process, the `Processor` is in the `P_IDLE` state; when a new job arrives, the `Processor` becomes busy and is in the `P_BUSY` state. The specification states that the Processor can discard a job when the current processor's queue is full. In DEVS, the output events correspond to sequential states. Only when the time interval for a specific sequential state expires, can an output event be produced. For the `Processor` model, there is no output event for the `P_IDLE` state. The processed jobs will be sent out when a time interval expires for the sequential state `P_BUSY`. In order to discard jobs that cannot be processed by the current `Processor`, we need another sequential state `P_DISCARDING` whose output event is the job that needs to be discarded. So the set of sequential states of the `Processor` model is `S = {P_IDLE, P_BUSY, P_DISCARDING}`.

A processor can accept unprocessed jobs, output processed jobs, and discard jobs when the current processor's capacity is full. So the `Processor` model has three ports, an input port `p_in` for accepting jobs, an output port `p_out` through which processed jobs are sent out, and an output port `p_discard` that is used to discard arriving jobs when the processor's queue is full. Events arriving at `p_in` port, and events sent to `p_out` and `p_discard` ports are all jobs. We can distinguish the input, output, and discarded jobs with timestamp and the ports that jobs pass through. For simplification reason, we do not distinguish them. So both input and output events for the `Processor` model are jobs. Each job has a size indicating how long a processor takes to process it, and a identification number used to distinguish jobs. So the input events and output event sets for the `Processor` model are `X = {instances of Job}`, `Y = {instances of Job}`, and the `job` is described as `Job(id, size)`.

The behaviour of the `Processor` model can be described as follows. At the beginning, the `Processor` is in the `P_IDLE` state. If there is no job arriving, the `Processor` will stay in the `P_IDLE` state forever. When a new job arrives, the `Processor` transits its sequential state to the `P_ BUSY` state and begins processing the newly arrived job. The `Processor` will stay in the `P_BUSY` state until the time interval specified by the job's size expires. During this period time, when a new job arrives, the `P_BUSY` state will be interrupted. The `Processor` has to remember how long it had been in the `P_BUSY` state and then check the status of its queue. If the queue is not full, the new job is put into the queue, and the `Processor` continues in the `P_BUSY` state, until the time left specified by the size of the job that is being processed expires. If the queue is full, the `Processor` goes to the `P_DISCARDING` state, and discards the incoming job, and then goes back to the `P_BUSY` state immediately. The `Processor` will stay in the `P_BUSY` state until the job that is being processed when the `Processor` is interrupted is finished. Just before the time specified by the job's size expires, the `Processor` outputs the current processing job onto the `p_out` port. Then, the `Processor` checks the status of its queue. If the queue is empty, the `Processor` goes to the `P_IDLE` state waiting for new jobs. If the queue is not empty, the `Processor` takes the first job of the queue as its current job, and goes to the `P_BUSY` state again, and repeats the previous process. Based on the above analysis, we obain the model state variables and the behaviour of the `Processor` model.

Basically, we need state variables to express the following information, the job that is being processed, the queue of the `Processor`, the size of the queue, and the time spent on the `P_BUSY` sequential state when a new job interrupts the `P_BUSY` state. So the `Processor` model has four state variables, named `currentJob`, `queue`, `queueSize`, and `timeElapsed`. Each of the variables represents the corresponding information mentioned above.

The behaviour functions for the `Processor` model can be described as below.

External Transitions

$$\begin{aligned}\delta_{ext}(P\_IDEL) &= P\_BUSY\\ \delta_{ext}(P\_BUSY) &= P\_DISCARDING, \text{ if the queue is full}\\ \delta_{ext}(P\_BUSY) &= P\_BUSY, \text{ if the queue is not full}\end{aligned}$$

Internal Transitions

$$\begin{aligned}\delta_{int}(P\_BUSY) &= P\_IDLE, \text{ if the queue is empty}\\ \delta_{int}(P\_BUSY) &= P\_BUSY, \text{ if the queue is not empty}\\ \delta_{int}(P\_DISCARDING) &= P\_BUSY\end{aligned}$$

Output Function

$$\begin{aligned}\lambda(P\_BUSY) &= job, \text{ the job currently being processed}\\ \lambda(P\_DISCARDING) &= job, \text{ the incoming job}\end{aligned}$$

Time Advance

$$\begin{aligned}t_a(P\_IDEL) &= INFINITY\\ t_a(P\_BUSY) &= currentJob.size - timeElapsed\\ t_a(P\_DISCARDING) &= 0\end{aligned}$$

6.2.2 The Chained-Processor Model

The chained-processor model is a coupled DEVS model with atomic `Processor` models connected together. Once the `Processor` has been designed, coupling individual processors together is straightforward. It is simple to make channels that connect corresponding ports together. The connections for the chained-processor model are shown in Figure 6.1.

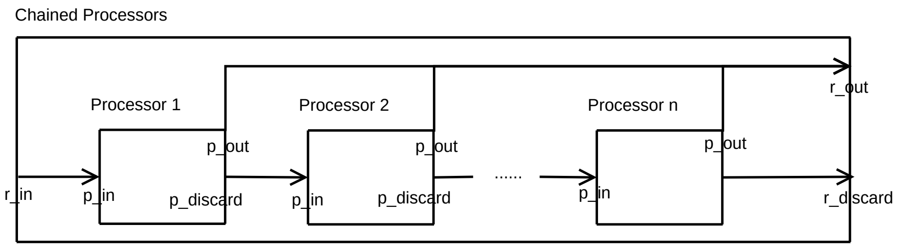


Figure 6.1: Coupled Model for the Chained-Processor System

The coupled model has three ports corresponding to the ports of the `Processor` models. The first `Processor` model's input port `p_in` is connected with the coupled model's input port `r_in`. The last `Pocessor's` output port `p_discard` is connected with the coupled model's output port `r_discard`. For all other `Processors`, each model's input port `p_in` is connected to its previous model's output port `p_discard`. And the output ports `p_out` of all the `Processors` are connected to the output port `r_out` of the coupled model.

6.3 The Visual Models

We have discussed the specification and design of the chained-processor system. Now let us see how the models can be represented in the Visual DEVS Modelling Environment introduced in chapter 5.

6.3.1 Job Event

The visual representation of the `Job` event is shown in Figure 6.2. The `Job` event has two parameters, `szl` and `szh`, and two attributes, `id` and `size`. The two parameters `szl` and `szh` represent the `Job's` size lower boundary and high boundary. The `id` and `size` have the same meanings as we discussed above. The `size` attribute is initialized with a random integer value uniformly chosen between size boundaries `szl` and `szh`.

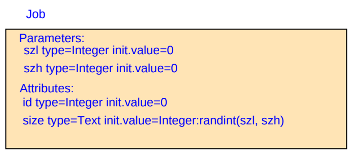

Job
Parameters:
szl type=Integer init.value=0
szh type=Integer init.value=0
Attributes:
id type=Integer init.value=0
size type=Text init.value=Integer:randint(szl, szh)

Figure 6.2: DEVS Event - Job

6.3.2 Atomic Processor

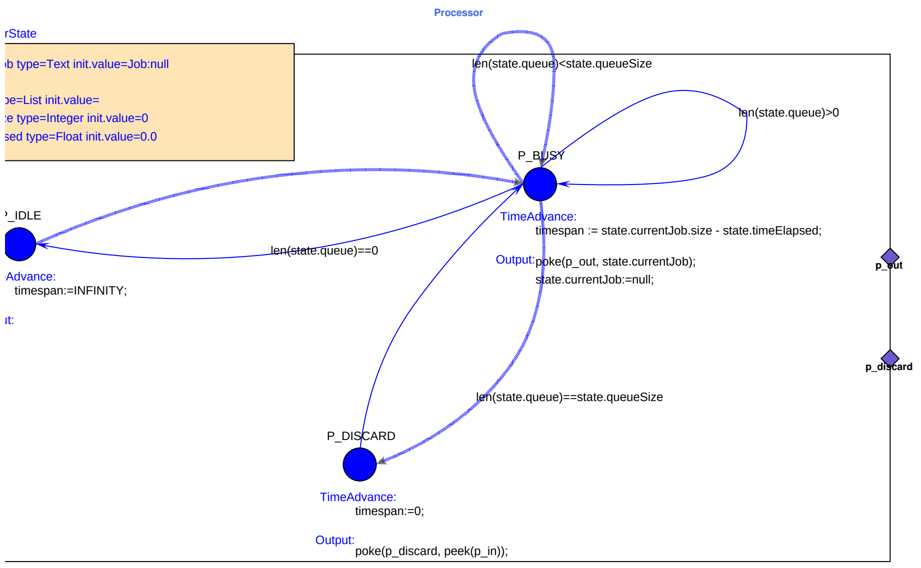

Processor
rState
b type=Text init.value=Job:null
e=List init.value=
e type=Integer init.value=0
sed type=Float init.value=0.0
len(state.queue)<state.queueSize
len(state.queue)>0
P_BUSY
TimeAdvance:
timespan := state.currentJob.size - state.timeElapsed;
Output:
poke(p_out, state.currentJob);
state.currentJob:=null;
_IDLE
len(state.queue)==0
Advance:
timespan:=INFINITY;
t:
p_out
p_discard
len(state.queue)==state.queueSize
P_DISCARD
TimeAdvance:
timespan:=0;
Output:
poke(p_discard, peek(p_in));

Figure 6.3: Atomic DEVS Model - Processor

The atomic `Processor` model is shown in Figure 6.3. This model includes two parts, the `ProcessorState` part and the model behaviour part. The `ProcessorState` discribes the state variables for this model, and the behaviour part uses the state diagram describing the sequential state transitions, time advance behaviour, and output in each state.

6.3.3 The Coupled DEVS Model - Chained Processors

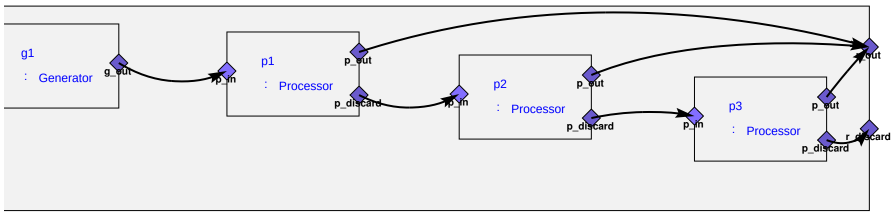
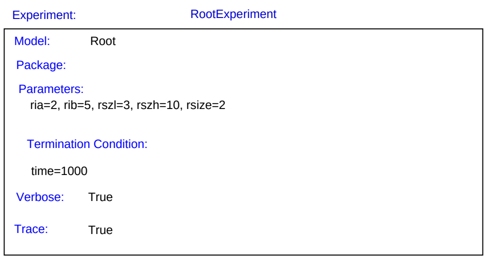


Figure 6.4: Coupled DEVS Model - Chained Processors

Figure 6.4 is an example of the chained-processor model named `Root` with three `Processors`. For simulation purposes, we include a `Generator` model `g1`, which is responsible for generating jobs in the coupled model. So the first `Processor p1's` input port `p_in` is connected with `g1's` output port `g_out`, rather than the coupled model's input port `p_in`.

6.3.4 Experiment Model



Experiment: RootExperiment

Model: Root

Package:

Parameters:
ria=2, rib=5, rszl=3, rszh=10, rsize=2

Termination Condition:
time=1000

Verbose: True

Trace: True

Figure 6.5: An Example of a DEVS Simulation Experiment

For completeness and convenience reasons, we include a visual experiment model component in the visual DEVS modelling environment. This model is used to specify simulation experiment parameters for a specific model. Figure 6.5 is an example of using the visual experiment specifying simulation parameters for the coupled model `Root` mentioned above.

6.4 The Modelica Representation

Models have been designed and created graphically. The next step is to generate Modelica representations for the visual models. The following are Modelica representations for the visual models mentioned in the previous section.

6.4.1 Job Event

The Job event is represented by a Modelica class. To express the fact that it is a DEVS event. This class extends the predefined model DevsEvent. The id property of the Job class has a special comment ''ID'', which is used to indicate the uniqueness of the id attribute for each Job instance. This property will be treated differently when a Modelica representation is translated into a computing language-specific model.

```
class Job
    extends DevsEvent;
    parameter Integer szl;
    parameter Integer szh;
    Integer id = 0 "ID" ;
    Integer size = randint(szl, szh);
end Job;
```

6.4.2 Atomic Processor

The visual atomic Processor model is represented by two Modelica classes, ProcessorState and Processor class. The ProcessorState class represents the model state variables, and hence it extends from the predefined DevsState class. The Processor class reprsents the real DEVS model of a processor, and so it extends from AtomicDEVS class.

Processor State

```
class ProcessorState
    extends DevsState;
    Job currentJob = null;
    DevsList queue();
    Integer queueSize = 0;
    Real timeElapsed = 0.0;
    Processor.SeqStates seqState(start=Processor.SeqStates.P_IDLE);
end ProcessorState;
```

Here, the state variable queue is represented as a DevsList object. The DevsList is a predefined class for DEVS in Modelica. It has the same semantics as the List type in Python. The automatically generated attribute seqState represents the model's sequential state. It is a type of Processor.SeqStates and is intialized with the model's initial sequential state.

Processor Model

Below is the Modelica code for the Processor model. Besides the standard atomic DEVS model functions, there is a function called initialization. This function comes from the init function of the visual atomic DEVS model. One can use it to initialize a model's state variables.

```
class Processor
    extends AtomicDEVS;
    parameter Integer qSize=0;
    parameter String name="a";
    input DevsPort p_in;
    output DevsPort p_out;
    output DevsPort p_discard;
    ProcessorState state();
    type SeqStates = enumeration(P_IDLE, P_BUSY, P_DISCARD);

    function initialization
    algorithm
        state.queueSize := qSize;
    end initialization;
    function extTransition
    algorithm
        if ( state.seqState == SeqStates.P_IDLE ) then
            state.currentJob:=peek(p_in);
            state.timeElapsed:=0;
            state.seqState := SeqStates.P_BUSY;
        elseif ( state.seqState == SeqStates.P_BUSY ) then
            state.timeElapsed:=state.timeElapsed+elapsed;
            if ( len(state.queue)<state.queueSize ) then
                state.queue.append(peek(p_in));
                state.seqState := SeqStates.P_BUSY;
            elseif ( len(state.queue)==state.queueSize ) then
                state.seqState := SeqStates.P_DISCARD;
            end if;
        end if;
    end extTransition;

    function intTransition
```

continue ...

```
algorithm
    if ( state.seqState == SeqStates.P_BUSY ) then
        if ( len(state.queue)==0 ) then
            state.timeElapsed:=0;
            state.currentJob:=null;
            state.seqState := SeqStates.P_IDLE;
        elseif ( len(state.queue)>0 ) then
            state.timeElapsed:=0;
            state.currentJob:=state.queue.pop(0);
            state.seqState := SeqStates.P_BUSY;
        end if;
    elseif ( state.seqState == SeqStates.P_DISCARD ) then
        state.seqState := SeqStates.P_BUSY;
    end if;
end intTransition;
function outputFnc
    DevsEvent evt = null;
algorithm
    if ( state.seqState == SeqStates.P_BUSY) then
        poke(p_out, state.currentJob);
        state.currentJob:=null;
    elseif ( state.seqState == SeqStates.P_DISCARD) then
        poke(p_discard, peek(p_in));
    end if;
end outputFnc;
function timeAdvance
    output Integer timespan;
algorithm
    if ( state.seqState == SeqStates.P_IDLE) then
        timespan:=INFINITY;
    end if;
    if ( state.seqState == SeqStates.P_BUSY) then
        timespan := state.currentJob.size - state.timeElapsed;
    elseif ( state.seqState == SeqStates.P_DISCARD) then
        timespan:=0;
    end if;
end timeAdvance;

end Processor;
```

6.4.3 Chained Processors - Root

The chained processors model extends the `CoupledDEVS` class. The coupled model constructs subcomponents and connect them.

```
class Root
    extends CoupledDEVS;
    parameter Integer ria;
    parameter Integer rib;
    parameter Integer rszl;
    parameter Integer rszh;
    parameter Integer rsize;
    parameter String name;
    output DevsPort r_out;
    output DevsPort r_discard;

continue ...
    Generator g1(ia=ria, ib=rib, szl=rszl, szh=rszh);
    Processor p1(qSize=rsize);
    Processor p2(qSize=rsize);
    Processor p3(qSize=rsize);
equation
    connect( g1.g_out, p1.p_in);
    connect( p1.p_out, r_out);
    connect( p1.p_discard, p2.p_in);
    connect( p2.p_out, r_out);
    connect( p2.p_discard, p3.p_in);
    connect( p3.p_out, r_out);
    connect( p3.p_discard, r_discard);
end Root;
```

6.4.4 The Experiment Model

Once the models have been constructed, the next step is to do experiments with the models. The experiment class extends the `DevsExperiment` class. It instanstiates the simulation model with given parameters and constructs a simulator with the model instance. When it is translated into language-specific code, the simulator will be executed and the model is simulated.

```
class RootExperiment
    extends DevsExperiment;
    Root rootModel(ria=2, rib=5, rszl=3, rszh=10, rsize=2, name="RootExperiment");
    Simulator sim(simModel=rootModel);
end RootExperiment;
```

6.5 Python DEVS Representation

In order to simulate the DEVS models, the Modelica represented models must be translated into programming languge-specific models. Theoretically, the Modelica representation can be translated into any language-specific DEVS models suitable for simulation by specific DEVS simulators. Below is the Python DEVS code generated by the DEVS Modelica compiler introduced in chapter 4 for the Modelica models discussed above.

6.5.1 Job Event

The Python DEVS representation of the `Job` event is also a class. In order to keep the uniqueness of the `id` property, a class variable `idNumber` is generated. Each time a `Job` is created, the `idNumber` is incremented. Subsequentially, the newly created `Job's id` is assigned with the value of the `idNumber`. Also, for trace study and analysis purposes, a utility `__str__` function and a `toXML` function are generated. The former function is used to translate a `Job` into a string representation, and the latter retruns the XML representation of a `Job`.

```
class Job( object ):
    idNumber = 0

    def __init__(self, szl, szh):
        self.szl = szl
        self.szh = szh
        self.id = Job.idNumber + 1
        Job.idNumber = Job.idNumber + 1
        self.size = randint(self.szl, self.szh)
    def __str__(self):
        strRep = ''
        strRep = strRep + "\nid: " + str(self.id)
        strRep = strRep + "\nsize: " + str(self.size)
        return strRep
    def toXML(self):
        strRep = ''
        strRep = strRep + "\n<attribute category=\"P\">"
        strRep = strRep + "\n\t<name>id</name>"
        strRep = strRep + "\n\t<type>Integer</type>"
        strRep = strRep + "\n\t<value>"+str(self.id)+"</value>"
        strRep = strRep + "\n</attribute>"
        strRep = strRep + "\n<attribute category=\"P\">"
        strRep = strRep + "\n\t<name>size</name>"
        strRep = strRep + "\n\t<type>Integer</type>"
        strRep = strRep + "\n\t<value>"+str(self.size)+"</value>"
        strRep = strRep + "\n</attribute>"

        return strRep
```

6.5.2 Atomic Processor

The Modelica representation of `ProcessorState` and `Processor` model are translated into Python classes `ProcessorState` and `Processor`.

Processor State

```
class ProcessorState:

    def __init__(self):
        self.currentJob = None
        self.queue = []
        self.queueSize = 0
        self.timeElapsed = 0.0
        self.seqState = Processor.P_IDLE
    def __str__(self):
        strRep = ''
        strRep = strRep + "\ncurrentJob: " + str(self.currentJob)
        strRep = strRep + "\nqueue: "
        for item in self.queue:
            strRep = strRep + "\n\t" + str(item)
        strRep = strRep + "\nqueueSize: " + str(self.queueSize)
        strRep = strRep + "\ntimeElapsed: " + str(self.timeElapsed)
        strRep = strRep + "\nseqState: " + str(self.seqState)
        return strRep
    def toXML(self):
        strRep = ''
        strRep = strRep + "\n<attribute category=\"C\">"
        strRep = strRep + "\n\t<name>currentJob</name>"
        strRep = strRep + "\n\t<type>Job</type>"
        if (self.currentJob!=None):
            strRep = strRep + "\n\t<value>"+self.currentJob.toXML()+"</value>"
        else:
            strRep = strRep + "\n\t<value>None</value>"
        strRep = strRep + "\n</attribute>"
        strRep = strRep + "\n<attribute category=\"P\">"
        strRep = strRep + "\n\t<name>queue</name>"
        strRep = strRep + "\n\t<type>list</type>"
        strRep = strRep + "\n\t<value>"
        for item in self.queue:
            strRep = strRep + "\n\t" + item.toXML()
        strRep = strRep + "\n\t</value>"
        strRep = strRep + "\n</attribute>"
        strRep = strRep + "\n<attribute category=\"P\">"
```

```
continue ......
        strRep = strRep + "\n\t<name>queueSize</name>"
        strRep = strRep + "\n\t<type>Integer</type>"
        strRep = strRep + "\n\t<value>"+str(self.queueSize)+"</value>"
        strRep = strRep + "\n</attribute>"
        strRep = strRep + "\n<attribute category=\"P\">"
        strRep = strRep + "\n\t<name>timeElapsed</name>"
        strRep = strRep + "\n\t<type>Real</type>"
        strRep = strRep + "\n\t<value>"+str(self.timeElapsed)+"</value>"
        strRep = strRep + "\n</attribute>"
        strRep = strRep + "\n<attribute category=\"P\">"
        strRep = strRep + "\n\t<name>seqState</name>"
        strRep = strRep + "\n\t<type>Processor.SeqStates</type>"
        strRep = strRep + "\n\t<value>"+str(self.seqState)+"</value>"
        strRep = strRep + "\n</attribute>"

return strRep
```

Processor Model

```
class Processor( AtomicDEVS ):
        P_IDLE = 'P_IDLE'
        P_BUSY = 'P_BUSY'
        P_DISCARD = 'P_DISCARD'

        def __init__(self, qSize, name):
                AtomicDEVS.__init__(self, name)
                self.qSize = qSize
                self.name = name
                self.p_in = self.addInPort("p_in")
                self.p_out = self.addOutPort("p_out")
                self.p_discard = self.addOutPort("p_discard")
                self.state = ProcessorState()
                self.initialization()
        def initialization( self ):
                self.state.queueSize = self.qSize
        def outputFnc( self ):
                evt = None
                if (self.state.seqState == Processor.P_BUSY):
                        self.poke(self.p_out, self.state.currentJob)
                        self.state.currentJob = None
                elif (self.state.seqState == Processor.P_DISCARD):
                        self.poke(self.p_discard, self.peek(self.p_in))
```

continue ......

```
def extTransition( self ):
    if (self.state.seqState == Processor.P_IDLE):
        self.state.currentJob = self.peek(self.p_in)
        self.state.timeElapsed = 0
        self.state.seqState = Processor.P_BUSY
    elif (self.state.seqState == Processor.P_BUSY):
        self.state.timeElapsed = self.state.timeElapsed + self.elapsed
        if (len(self.state.queue) < self.state.queueSize):
            self.state.queue.append(self.peek(self.p_in))
            self.state.seqState = Processor.P_BUSY
        elif (len(self.state.queue) == self.state.queueSize):
            self.state.seqState = Processor.P_DISCARD
    return self.state
def intTransition( self ):
    if (self.state.seqState == Processor.P_BUSY):
        if (len(self.state.queue) == 0):
            self.state.timeElapsed = 0
            self.state.currentJob = None
            self.state.seqState = Processor.P_IDLE
        elif (len(self.state.queue) > 0):
            self.state.timeElapsed = 0
            self.state.currentJob = self.state.queue.pop(0)
            self.state.seqState = Processor.P_BUSY
    elif (self.state.seqState == Processor.P_DISCARD):
        self.state.seqState = Processor.P_BUSY
    return self.state
def timeAdvance( self ):
    if (self.state.seqState == Processor.P_IDLE):
        timespan = INFINITY
    elif (self.state.seqState == Processor.P_BUSY):
        timespan = self.state.currentJob.size - self.state.timeElapsed
    elif (self.state.seqState == Processor.P_DISCARD):
        timespan = 0
    return timespan
```

6.5.3 Coupled Multi-Processors

For Python DEVS, each model requires a name. In the generated coupled Python DEVS models, each sub-model is given a name using the textual representation of the sub-model's instance variable name. For instance, the `Generator` instance `g1` is given the name `''g1''`.

```
class Root( CoupledDEVS ):

    def __init__(self, ria, rib, rszl, rszh, rsize, name):
        CoupledDEVS.__init__(self, name)
        self.ria = ria
        self.rib = rib
        self.rszl = rszl
        self.rszh = rszh
        self.rsize = rsize
        self.name = name
        self.r_out = self.addOutPort("r_out")
        self.r_discard = self.addOutPort("r_discard")
        self.g1 = Generator(ia = self.ria, ib = self.rib, szl = self.rszl, szh = self.rszh,
        name='g1')
        self.addSubModel( self.g1 )
        self.p1 = Processor(qSize = self.rsize, name='p1')
        self.addSubModel( self.p1 )
        self.p2 = Processor(qSize = self.rsize, name='p2')
        self.addSubModel( self.p2 )
        self.p3 = Processor(qSize = self.rsize, name='p3')
        self.addSubModel( self.p3 )
        self.connectPorts( self.g1.g_out, self.p1.p_in)
        self.connectPorts( self.p1.p_out, self.r_out)
        self.connectPorts( self.p1.p_discard, self.p2.p_in)
        self.connectPorts( self.p2.p_out, self.r_out)
        self.connectPorts( self.p2.p_discard, self.p3.p_in)
        self.connectPorts( self.p3.p_out, self.r_out)
        self.connectPorts( self.p3.p_discard, self.r_discard)
```

6.5.4 The Experiment Model

For Python DEVS, besides the experiment class, an experiment termination condition function and a `__main__` block of Python code are generated. Thus, the generated Python DEVS models can be executed directly by the Python interpreter. Users can modify this piece of code for their own convenience.

```
class RootExperiment( object ):

    def __init__(self):
        self.rootModel = Root(ria = 2, rib = 5, rszl = 3, rszh = 10, rsize = 2, name = "RootEx-
        periment")
        self.sim = Simulator(model = self.rootModel)
```

```
continue ......

def terminate_whenEndTimeReached(model, clock, end_time=1000):

    if clock >= end_time:
        return True
    else:
        return False

if __name__ == '__main__':
    experiment = RootExperiment()
    experiment.sim.simulate(termination_condition=terminate_whenEndTimeReached, verbose=True,
    trace=True)
```

6.6 Simulation Trace

Once Modelica DEVS representations are compiled into Python DEVS models, the Python models can be simulated by the Python DEVS simulator. Based on the automatically generated `__str__` and `toXML` functions of the Python DEVS models, an XML representation of DEVS simulation trace file is generated. Below is a small part of the simulation trace for the experiment `RootExperiment` mentioned above.

```
<trace>
    ......

    <event>
        <model>RootExperiment.p1</model>
        <time>0.0</time>
        <kind>EX</kind>
        <state>
            <attribute category="C">
                <name>currentJob</name>
                <type>Job</type>
                <value>None</value>
            </attribute>
            <attribute category="P">
                <name>queue</name>
                <type>DevsList</type>
                <value></value>
            </attribute>
            <attribute category="P">
                <name>queueSize</name>
                <type>Integer</type>
                <value>2</value>
            </attribute>
            <attribute category="P">
                <name>timeElapsed</name>
                <type>Real</type>
                <value>0.0</value>
            </attribute>
            <attribute category="P">
                <name>seqState</name>
                <type>Processor.SeqStates</type>
                <value>P_IDLE</value>
            </attribute>
        </state>
    </event>

    ......

</trace>
```

This part of the simulation trace includes the details of one event for the model `RootExperiment.p1`. This event is the initialization event which occurs at simulation time 0. Because the intialization event is sent by the coordinator, the value of the `kind` element of this event is set as `‘‘EX’’`, which means an external event to `RootExperiment.p1`. The initial values of the state variables of this model are reflected by the values of corresponding `attribute` elements of the event.

6.7 Trace Plotting Using Visual Trace Plotter

Now we have the XML representation of the simualtion trace. We can use the visual trace plotter (see chapter 3) to plot the trace file.

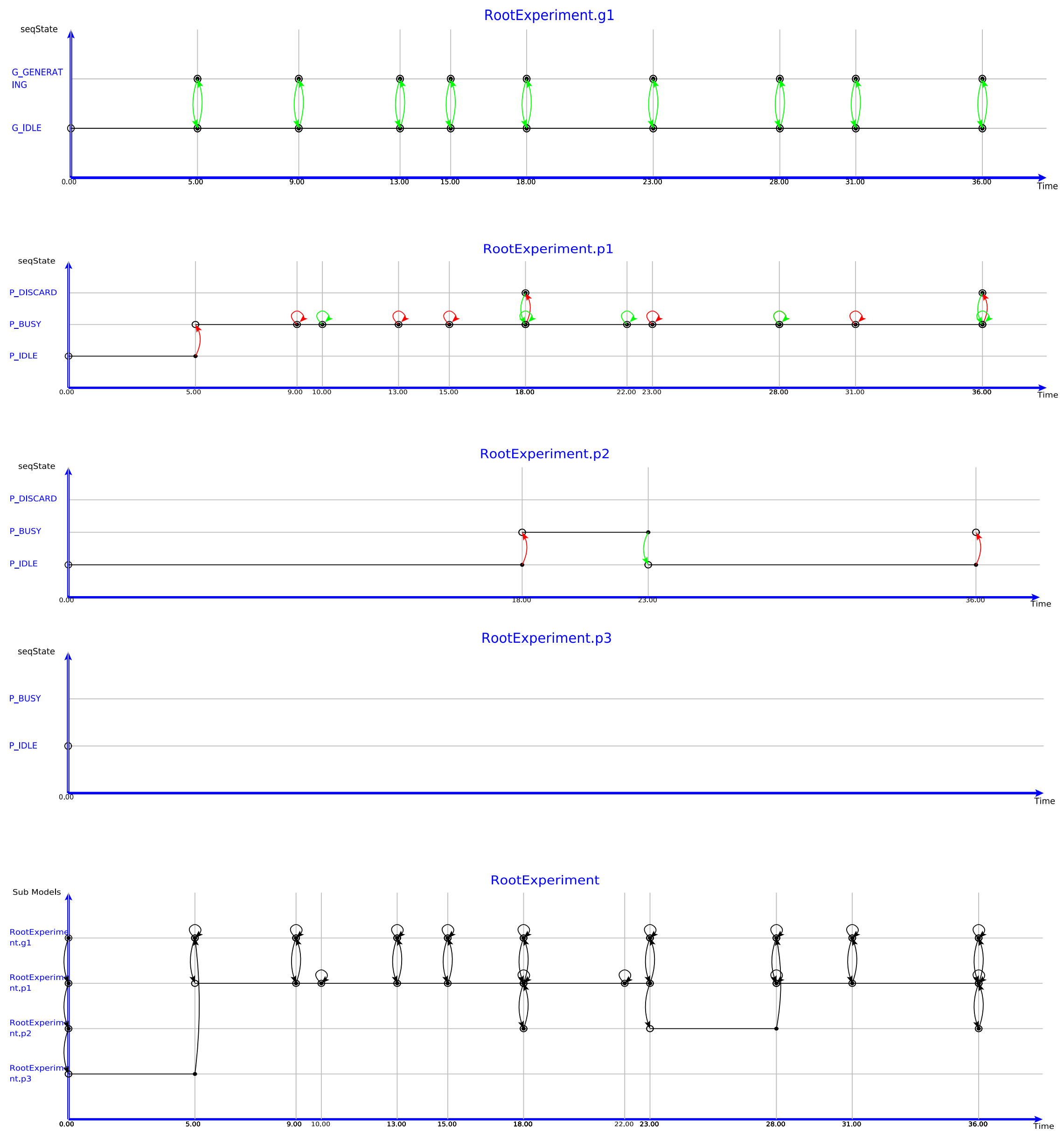


Figure 6.6: The Beginning Events of a Simulation

Figure 6.6 is a simulation trace for all models beginning from simulation time 0. On the top of the figure is the trace of the `Generator g1`, from which we can see that after a random time interval, a `Job` is generated. Next to `g1` is the `Processor p1`. Because `p1's` input port `p_in` is connected with `g1's` output port `g_out`, every time a job is generated in `g1`, an external event occurs in `Processor p1`. As `p1's p_discard` port is connected with `p2's p_in` port, once a job is discarded from `p1`, a external event happens at `p2`. `P2` and `p3` have the same relation as that of `p1` and `p2`. Because during the simulation time shown in Figure 6.6, `p2` has never discarded a job, there is no external event in `p3`, and so it is always in the `P_IDLE` state.

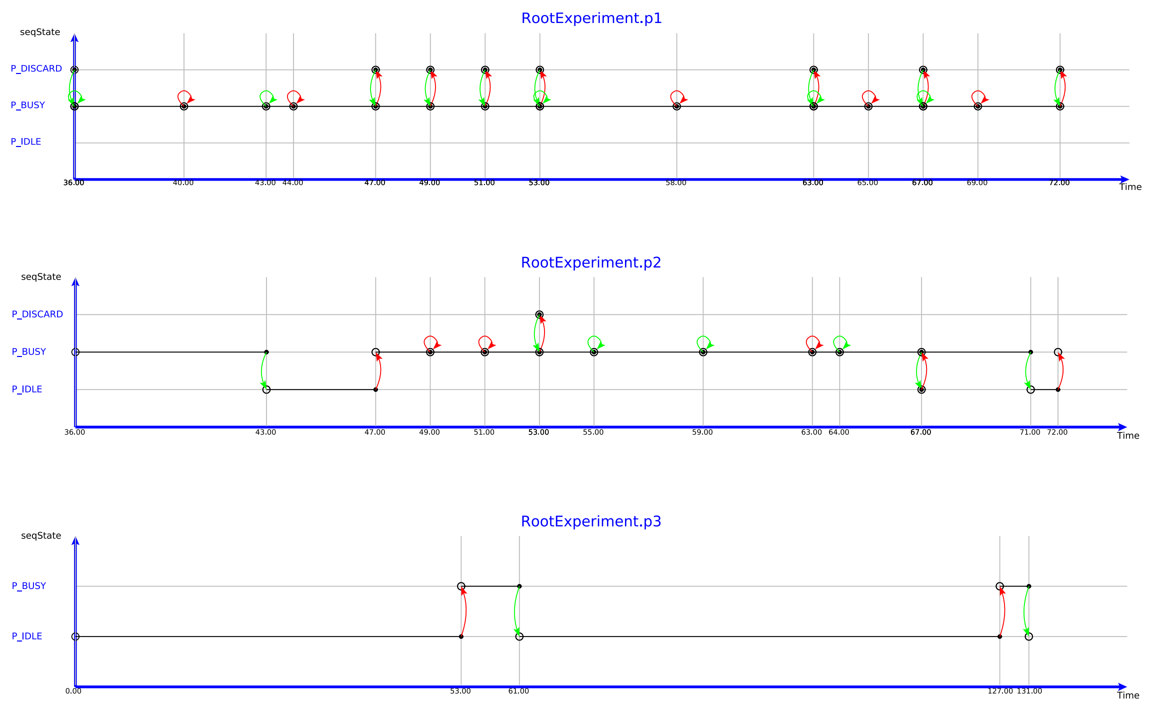


Figure 6.7: Simulation Trace That All Processors Having Jobs

Figure 6.7 presents the simulation trace during a period time that all the three processors have jobs. Combining Figure 6.6 and Figure 6.7 , we can see that, at the beginning of the simulation, `p1` is very busy, `p2` can get jobs occasionally, and `p3` is free; after a period of time, `p2` becomes very busy, and discards jobs occasionally to `p3`. From the two figures we also see that `p3's seqState` coordinate has only two states `P_IDLE` and `P_BUSY`, while `p1` and `p2` have three states `P_IDLE`, `P_BUSY`, and `P_DISCARD`. This means that `p3` only transits its sequential states between `P_IDLE` and `P_BUSY`, and it has never discarded a job.

7 Conclusions and Future Work

We have finished the discussion of the Infrastructure for DEVS Modelling and Experiment system. Let us give a summary to the thesis, draw conclusions for the research, and present issues for future work.

In this thesis, we mainly discuss the following issues. In the first chapter we discussed the background of DEVS and introduced some related research in building DEVS modelling and simulation environments. In the second chapter, we presented some concepts used in this thesis and introduced the overall architecture of the Infrastructure. In the third chapter, we discussed model validation and trace plotting, and introduced the design and implemention of the DEVS visual trace plotter. In the fourth chapter, we looked into neutral DEVS model representation, in which we introduced the object-oriented model description language Modelica, and discussed representing DEVS models using Modelica and challenges of translating neutral Modelica represented DEVS models into programming language represented models. In the fifth chapter, we discussed meta-modelling and building visual modelling environment. We introduced multi-formalism modelling and meta-modelling environment AToM^3, designed and implemented the DEVS visual modelling environment in AToM^3. In the sixth chapter, we put everything introduced in the previous chapters together and gave a case study using all the parts of the Infrastructure to build DEVS models for the chained processors system.

In the research and development of the Infrastructure, we made the following contributions to DEVS modelling and simulation.

1. We defined an XML DTD to standardize the DEVS simulation trace. The benefits of using standardized simulation trace are obvious. The standardized simulation trace makes a clear interface between DEVS simulators and DEVS simulation trace plotters. One trace plotter can plot traces generated by different simulators follow the same XML DTD. Traces generated by one simulator can be plotted by different plotters that understand the same XML DTD.
2. We designed and implemented a visual DEVS trace plotter that can visualize a standardized XML represented DEVS simulation trace. The visual trace plotter can display traces of different models. This makes trace analysis and model debugging much easier than reading simulation trace in pure textual format.
3. We proposed and demonstrated the ideas of representing DEVS models in high-level modelling language Modelica, and built a Modelica model compiler that can translate Modelica DEVS model representations into Python DEVS model representations. Representing DEVS models at a high level has many benefits. Firstly, it release the burden of the modeller to learn programming languages. Secondly, it provides opportunities for model compilers to verify models' syntax automatically. And finally, high-level model represen-

tations in modelling languages are easier to be standardized than models represented in low-level programming languages.

4. We built a visual modelling environment for DEVS. There are two major features of the visual modelling environment. First, the environment is modelled and is generated automatically based on its models. Second, the visual DEVS models built in the environment can be translated into Modelica DEVS model representations.
5. We integrated the meta-modelling, visual modelling, neutral model representation, and trace visualization technologies together to demonstrate the viability and feasibility of applying them to DEVS modelling and exmperiment.

The following aspects need more efforts for the future research.

1. For the visual modelling environment, the visual modelling GUI interface needs to be enhanced. Due to the fact that the visual modelling environment is automatically generated, the graphical user interface of the environment is not very user friendly. More work needs to be done in the future to make it more user friendly.
2. The Modelica compiler checked Modelica DEVS models' syntax and generated Python DEVS model representations. One more issue for the compiler needs to be considered in the future. Which parts of the DEVS formalism should be verified and enforced by the compiler, and how to do that?
3. The Modelica DEVS model compiler can only generate output for Python DEVS now. Compilers for generating output for other DEVS simulators such as DEVSJava, and ADEVS should be considered in the future.
4. We presented a case study using the Infrastructure in this thesis. This partially tested the functionalities of the system. More case studies need to be done to further test the system.

Bibliography

[AB94] Chow A.C.H. and Zeigler B.P. Parallel DEVS: a parallel, hierarchical, modular modeling formalism. In *Winter Simulation Conference Proceedings, 11-14 Dec.*, pages 716–722, 1994.

[ABK94] Chow A.C., Zeigler B.P., and Doo Hwan Kim. Abstract simulator for the parallel DEVS formalism. In *AI, Simulation, and Planning in High Autonomy Systems, 1994 ('Distributed Interactive Simulation Environments'), Proceedings of the Fifth Annual Conference on 7-9 Dec. 1994*, pages 157–163, 1994.

[AEA$^+$02] Muzy A., Innocenti E., Aiello A., Santucci J.F., and Wainer G. Cell-DEVS quantization techniques in a fire spreading application. In *Proceedings of the Winter Simulation Conference, 2002. Volume 1*, pages 542–549, 2002.

[AHW05] Knupfer A., Brunst H., and Nagel W.E. High performance event trace visualization. In *Parallel, Distributed and Network-Based Processing, 2005. PDP 2005. 13th Euromicro Conference on 9-11 Feb. 2005*, pages 258–263, 2005.

[AT02] Ferscha Alois and Satish K. Tripathi. Parallel and distributed simulation of discrete event systems. Technical report, 2002.

[Bar97] Fernando J. Barros. Modeling formalisms for dynamic structure systems. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 7(4):501–515, 1997.

[BV02] Jean Sebastien Bolduc and Hans Vangheluwe. A modeling and simulation package for classic hierarchical DEVS. Technical report, 2002.

[Dub06] Denis Dube. Graph layout for domain specific modelling. Master's thesis, McGill University, 2006.

[EHS97] Jan Ellsberger, Dieter Hogrefe, and Amardeo Sarma. *SDL : formal object-oriented language for communicating systems.* Prentice Hall, London, New York, 1997.

[Fri04] Peter Fritzson. *Principles of object-oriented modeling and simulation with Modelica 2.1.* IEEE Press, New York, 2004.

[HK06] Ki Jung Hong and Tag Gon Kim. DEVSpecl: DEVS specification language for modeling, simulation and analysis of discrete event systems. *Information and Software Technology*, 48:221–234, 2006.

[HR00] D. Harel and B. Rumpe. Modeling languages: Syntax, semantics and all that stuff, part i: The basic stuff. Technical report, Jerusalem, Israel, 2000.

[Nut05] James J. Nutaro. ADEVS: A discrete event system simulator. Technical report, 2005.

[PBV03] Ernesto Posse, Jean-S´ebastien Bolduc, and Hans Vangheluwe. Generation of DEVS modelling and simulation environments. Technical report, http://www.cs.mcgill.ca/ hv/publications/03.SCSC.DEVScodegen.pdf, 2003.

[PP93] Herbert Praehofer and Dietmar Pree. Visual modeling of DEVS-based multiformalism systems based on higraphs. In *Proceedings of the 25th conference Winter simulation*, pages 595–603, Los Angeles, California, United States, 1993.

[SK94] Hae Sang Song and Tag Gon Kim. The DEVS framework for discrete event systems control. In *AI, Simulation, and Planning in High Autonomy Systems, 1994 ('Distributed Interactive Simulation Environments'), Proceedings of the Fifth Annual Conference on 7-9 Dec. 1994*, pages 228–234, 1994.

[ST00] Takahashi S. and Ishioka T. Trace visualization and analysis tool for supervisory control systems. In *Systems, Man, and Cybernetics, 2000 IEEE International Conference on Volume 2, 8-11 Oct. 2000*, pages 1198–1203, 2000.

[Uhr01] A. M. Uhrmacher. Dynamic structures in modeling and simulation: a reflective approach. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 11(2):206–232, 2001.

[Van04] Hans. Vangheluwe. The discrete event system specification (DEVS) formalism. Technical report, 2004.

[VdL03] Hans Vangheluwe and Juan de Lara. *Computer Automated Multi-paradigm Modelling: Meta-modelling and Graph Transformation.* IEEE, New Orleans, Louisiana, December 2003.

[VdL05] Hans Vangheluwe and Juan de Lara. *Model-Based Development: Meta-Modelling, Transformation and Verification*, pages 289–312. The Idea Group Inc., October 2005.

[WG01a] G. Wainer and N. Giambiasi. Timed cell-DEVS: modeling and simulation of cell spaces. In *Discrete Event Modeling and Simulation: Enabling Future Technologies*, Springer, 2001.

[WG01b] Gabriel A. Wainer and Norbert Giambiasi. Application of the cell-DEVS paradigm for cell spaces modeling and simulation. *Simulation*, 76(1):22–39, 2001.

[Xu05] Weigao Xu. The design and implementation of the μmodelica compiler. Master's thesis, McGill University, 2005.

[Y.C89] Ho Y.C. Editor's introduction to special issue on deds. *Proceedings of the IEEE*, 77:3–6, 1989.

[Zei84] B. P. Zeigler. *Multifacetted Modeling and Discrete Event Simulation.* Academic Press, London, 1984.

[Zei03] B.P. Zeigler. DEVS today: recent advances in discrete event-based information technology. In *Modeling, Analysis and Simulation of Computer Telecommunications Systems MASCOTS 2003 (11th IEEE/ACM International Symposium on 12-15 Oct. 2003)*, pages 148–161, 2003.

[Zei05] Bernard P. Zeigler. Introduction to DEVS modeling and simulation with JAVA: Developing component-based simulation models. Technical report, 2005.

[ZKB99] Bernard P. Zeigler, Doohwan Kim, and Stephen J. Buckley. Distributed supply chain simulation in a DEVS/CORBA execution environment. In *December 1999 Proceedings of the 31st conference on Winter simulation: Simulation—a bridge to the future - Volume 2*, 1999.

[ZPK00] B. P. Zeigler, H. Praehofer, and T. G. Kim. *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems, 2nd Edition.* Academic Press, New York, 2000.

[ZV93] Bernard P. Zeigler and Sankait Vahie. DEVS formalism and methodology: Unity of conception / diversity of application. In *Proceedings of the Winter Simulation Conference*, pages 573–579, 1993.