Universiteit Antwerpen

Translating statecharts and class diagrams to DEVS

Sam Pieters

Principal Adviser: Hans Vangheluwe
Assistant Adviser: Joeri Exelmans

Dissertation Submitted in May 2024 to the
Department of Mathematics and Computer Science
of the Faculty of Sciences, University of Antwerp,
in Partial Fulfillment of the Requirements
for the Degree of Master of Science.

Contents

List of Figures

List of Tables

Nederlandstalige Samenvatting

Nederlandse samenvatting komt hier.

Acknowledgements

I would like to express my deepest gratitude to Professor Hans Vangheluwe for his invaluable guidance, unwavering support, and for entrusting me with the project that laid the foundation for this thesis. His insightful perspectives and encouragement have been instrumental in shaping my academic journey.

I am also indebted to Joeri Excelmans and Randy Paredis for their consistent assistance and unwavering commitment throughout the duration of this project. Their expertise, encouragement, and willingness to lend a helping hand have been truly invaluable.

Abstract

This master's thesis explores the translation of Statecharts and class diagrams (SCCD) models into the PyDEVS (Python-based Discrete Event System Specification) framework, investigating its implications and applications. SCCD, a language that combines the Statecharts language with Class Diagrams. It allows users to model complex, timed, autonomous, reactive, dynamic-structure systems. PyDEVS, on the other hand, provides a versatile platform for discrete event simulation. The primary objective of this research is to bridge the gap between these two simulation paradigms, facilitating the integration of SCCD models into the PyDEVS ecosystem. This translation enables the utilization of SCCD models within PyDEVS for simulation and analysis, thereby extending the reach of SCCD-based approaches to a broader audience of researchers and practitioners. The thesis first presents an in-depth analysis of SCCD and PyDEVS, highlighting their respective features, strengths, and weaknesses. Subsequently, it proposes a systematic methodology for translating SCCD models into PyDEVS specifications, addressing challenges such as semantic differences, modeling paradigms, and computational efficiency. To demonstrate the utility of this translation approach, the thesis showcases several case studies where SCCD models are successfully translated into PyDEVS and simulated using the PyDEVS framework. Furthermore, the thesis discusses the practical implications and benefits of integrating SCCD into PyDEVS, such as enhanced scalability, interoperability with existing simulation tools, and the ability to leverage PyDEVS's rich ecosystem of libraries and tools for simulation analysis. Overall, this research contributes to advancing the field of discrete event simulation by facilitating the seamless integration of SCCD models into the PyDEVS framework, opening up new avenues for research and applications in diverse domains requiring concurrent system modeling and analysis.

CHAPTER 1

Introduction

In today's complex and interconnected world, the modeling and simulation of concurrent systems play a pivotal role in understanding and analyzing various phenomena, ranging from distributed algorithms to communication protocols. Synchronous Concurrent Constraint Datalog (SCCD) has emerged as a powerful formalism for expressing and reasoning about concurrent systems, offering expressive capabilities to capture intricate interactions among components. On the other hand, the PyDEVS (Python-based Discrete Event System Specification) framework provides a versatile platform for discrete event simulation, facilitating the modeling and analysis of dynamic systems.

However, despite their individual strengths, SCCD and PyDEVS represent distinct simulation paradigms with differences in modeling semantics, computational approaches, and tooling ecosystems. This dichotomy poses challenges for researchers and practitioners seeking to leverage the advantages of both SCCD and PyDEVS for modeling and simulating concurrent systems.

This master's thesis aims to bridge this gap by exploring the translation of SCCD models into the PyDEVS framework, thereby facilitating the integration of SCCD-based approaches into the PyDEVS ecosystem. By enabling the simulation of SCCD models using PyDEVS, this research seeks to extend the reach of SCCD-based methodologies to a broader audience while harnessing the capabilities of PyDEVS for efficient and scalable simulation.

In this introduction, we provide an overview of SCCD and PyDEVS, highlighting their respective features, strengths, and applications. We also outline the motivation behind this research, discussing the potential benefits of integrating SCCD into PyDEVS and the challenges inherent in reconciling the differences between these simulation paradigms. Furthermore, we present an

outline of the thesis, detailing the structure and organization of subsequent chapters, which include an analysis of SCCD and PyDEVS, a methodology for SCCD-to-PyDEVS translation, case studies demonstrating the translation approach, and discussions on the implications and future directions of this research.

Overall, this thesis aims to contribute to the advancement of discrete event simulation by bridging simulation paradigms and facilitating the seamless integration of SCCD models into the PyDEVS framework, thereby fostering innovation and exploration in concurrent system modeling and analysis.

1.1 Definitions

1. **Statecharts and class diagrams (SCCD):** A language that combines the Statecharts language with Class Diagrams. It allows users to model complex, timed, autonomous, reactive, dynamic-structure systems.

2. **PyDEVS:** Python-based Discrete Event System Specification, a framework for discrete event simulation that provides a platform for modeling and analyzing dynamic systems using the DEVS (Discrete Event System Specification) formalism.

3. **Discrete Event Simulation:** A simulation technique used to model systems where events occur at distinct points in time, with changes in system state triggered by these events.

4. **Translation:** The process of converting models or specifications from one formalism or representation to another while preserving essential characteristics and semantics.

5. **Modeling Paradigm:** A set of principles, concepts, and methodologies used for constructing models to represent real-world systems or phenomena.

6. **Semantic Differences:** Variations in meaning or interpretation between different formalisms or modeling approaches, which may require adjustments or transformations during the translation process.

7. **Computational Efficiency:** The ability of a simulation framework or algorithm to execute simulations with minimal computational resources, such as time and memory.

8. **Statechart:**

9. **SCXML:**

CHAPTER 2

Background

2.1 SCCD

2.1.1 Statechart

2.1.2 SCXML

2.2 DEVS Formalism

The Discrete Event System Specification (DEVS) formalism was first introduced by Zeigler (TODO) in 1976 to provide a rigourous common basis for discrete-event modelling and simulation. For the class of formalisms denoted as discrete-event (TODO), system models are described at an abstraction level where the time base is continuous (TODO), but during a bounded time-span, only a finite number of relevant events occur. These events can cause the state of the system to change. In between events, the state of the system does not change. This is unlike continuous models in which the state of the system may change continuously over time. As an extension of Finite State Automata, the DEVS (Discrete Event Systems) formalism captures concepts from Discrete Event simulation. As such it is a sound basis for meaningful model exchange in the Discrete Event realm. (TODO: citation)

The DEVS formalism fits the general structure of deterministic, causal sys-

tems 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. It is shown in [Zei84a] how the DEVS formalism is closed under coupling: for each coupled DEVS, a resultant atomic DEVS can be constructed. 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. In the following, the different aspects of the DEVS formalism are explained in more detail.

2.2.1 The atomic DEVS Formalism

The atomic DEVS formalism is a structure describing the different aspects of the discrete-event behaviour of a system:

$$atomicDEVS \equiv \langle S, ta, \delta_{int}, X, \delta_{ext}, Y, \lambda \rangle \tag{2.1}$$

The time base T is continuous and is not mentioned explicitly

$$T = R \tag{2.2}$$

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

$$S = \times_{i=1}^{n} S_i. \tag{2.3}$$

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

$$ta : S \rightarrow R^{+}_{0,+\infty}. \tag{2.4}$$

As time in the real world always advances, the image of ta must be non-negative numbers. $ta = 0$ 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 $ta(s) = +\infty$.

The internal transition function

$$\delta_{int} : S \to S \tag{2.5}$$

models the transition from one state to the next sequential state. δ_{int} describes the behaviour of a Finite State Automaton; ta 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)

$$Y = \times_{i=1}^{l} Y_i. \tag{2.6}$$

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

The output function

$$\lambda : S \to Y \cup \{\phi\} \tag{2.7}$$

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 ϕ 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

$$Q = \{(s, e) | s \in S, , 0 \leq e \leq ta(s)\} \tag{2.8}$$

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

$$\sigma = ta(s) - e \tag{2.9}$$

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)

$$X = \times_{i=1}^{m} X_i \tag{2.10}$$

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

The set Ω contains all admissible input segments ω

$$\omega : T \to X \cup \{\phi\} \tag{2.11}$$

In discrete-event system models, an input segment generates an input event different from the non-event ϕ 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

$$\delta_{ext} : Q \times X \to S \tag{2.12}$$

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.

2.2.2 The coupled DEVS Formalism

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

$$coupledDEVS \equiv \langle X_{self}, Y_{self}, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\}, select \rangle \tag{2.13}$$

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

$$\{M_i | i \in D\}. \tag{2.14}$$

Each of the components must be an atomic DEVS

$$M_i = \langle S_i, ta_i, \delta_{int,i}, X_i, \delta_{ext,i}, Y_i, \lambda_i \rangle, \forall i \in D. \tag{2.15}$$

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

$$\{I_i | i \in D \cup \{self\}\}. \tag{2.16}$$

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 $self$:

$$\forall i \in D \cup \{self\} : I_i \subseteq D \cup \{self\}. \tag{2.17}$$

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:

$$\forall i \in D \cup \{self\} : i \notin I_i. \tag{2.18}$$

Note how one can always 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 $Z_{i,j}$ are defined:

$$\{Z_{i,j} | i \in D \cup \{self\}, j \in I_i\}, Z_{self,j} : X_{self} \to X_j, \forall j \in D, Z_{i,self} : Y_i \to Y_{self}, \forall i \in D, Z_{i,j} : Y_i - \tag{2.19}$$

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 (TODO). 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:

$$select : 2^D \to D \tag{2.20}$$

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

$$select(E) \in E. \tag{2.21}$$

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

2.2.3 Closure of DEVS under coupling

As mentioned before, it is possible to construct a resultant atomic DEVS model for each coupled DEVS. This closure under coupling of atomic DEVS models makes any coupled DEVS equivalent to an atomic DEVS. By induction, any hierarchically coupled DEVS can thus be flattened to an atomic DEVS. As a result, the requirement that each of the components of a coupled DEVS be an

atomic DEVS can be relaxed to be atomic or coupled as the latter can always be replaced by an equivalent atomic DEVS.

The core of the closure procedure is the selection of the most imminent (i.e., soonest to occur) event from all the components' scheduled events (TODO). In case of simultaneous events, the select function is used. In the sequel, the resultant construction is described.

From the coupled DEVS

$$\langle X_{self}, Y_{self}, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\}, select\rangle \tag{2.22}$$

with all components M_i atomic DEVS models

$$M_i = \langle S_i, ta_i, \delta_{int,i}, X_i, \delta_{ext,i}, Y_i, \lambda_i\rangle, \forall i \in D \tag{2.23}$$

the atomic DEVS

$$\langle S, ta, \delta_{int}, X, \delta_{ext}, Y, \lambda\rangle \tag{2.24}$$

is constructed.

The resultant set of sequential states is the product of the total state sets of all the components

$$S = \times_{i\in D} Q_i \tag{2.25}$$

where

$$Q_i = \{(s_i, e_i)|s \in S_i, 0 \leq e_i \leq ta_i(s_i)\}, \forall i \in D. \tag{2.26}$$

The time advance function ta

$$ta : S \to R^+_{0,+\infty} \tag{2.27}$$

is constructed by selecting the most imminent event time, of all components. This means, finding the smallest time remaining until internal transition, of all the components

$$ta(s) = \min\{\sigma_i = ta_i(s_i) - e_i|i \in D\}. \tag{2.28}$$

A number of imminent components may be scheduled for a simultaneous internal transition. These components are collected in a set

$$IMM(s) = \{i \in D|\sigma_i = ta(s)\}. \tag{2.29}$$

From IMM, a set of elements of D, one component i^* is chosen by means of the select tie-breaking function of the coupled model

$$select : 2^D \to DIMM(s) \to i^* \tag{2.30}$$

Output of the selected component is generated before it makes its internal transition. Note also how, as in a Moore machine, input does not directly influence output. In DEVS models, only an internal transition produces output.

An input can only influence/generate output via an internal transition similar to the presence of memory in the form of integrating elements in continuous models. Allowing an external transition to produce output could lead to infinite instantaneous loops. This is equivalent to algebraic loops in continuous systems. The output of the component is translated into coupled model output by means of the coupling information

$$\lambda(s) = Z_{i^*,self}(\lambda_{i^*}(s_{i^*})), if self \in I_{i^*}. \tag{2.31}$$

If the output of i^* is not connected to the output of the coupled model, the non-event ϕ can be generated as output of the coupled model. As ϕ literally stands for no event, the output can also be ignored without changing the meaning (but increasing performance of simulator implementations).

The internal transition function transforms the different parts of the total state as follows:

$$\delta_{int}(s) = (..., (s'_j, e'_j), ...), where (s'_j, e'_j) = (\delta_{int,j}(s_j), 0), for j = i^*, = (\delta_{ext,j}(s_j, e_j + ta(s), Z_{i^*,j})) \tag{2.32}$$

The selected imminent component i^* makes an internal transition to sequential state $\delta_{int,i^*}(s_i^*)$. Its elpased time is reset to 0. All the influencees of i^* change their state due to an external transition prompted by an input which is the output-to-input translated output of i^*, with an elapsed time adjusted for the time advance $ta(s)$. The influencees' elapsed time is reset to 0. Note how i^* is not allowed to be an influencee of i^* in DEVS. The state of all other components is not affected and their elapsed time is merely adjusted for the time advance $ta(s)$.

The external transition function transforms the different parts of the total state as follows:

$$\delta_{ext}(s, e, x) = (..., (s'_i, e'_i), ...) \tag{2.33}$$

An incoming external event is routed, with an adjustment for elapsed time, to each of the components connected to the coupled model input (after the appropriate input-to-input translation). For all those components, the elapsed time is reset to 0. All other components are not affected and only the elapsed time is adjusted.

Some limitations of DEVS are that

- a conflict due to simultaneous internal and external events is resolved by ignoring the internal event. It should be possible to explicitly specify behaviour in case of conflicts;
- there is limited potential for parallel implementation;
- the select function is an artificial legacy of the semantics of traditional sequential simulators based on an event list;

- it is not possible to describe variable structure.

Some of these are compensated for in parallel DEVS (TODO).

2.2.4 Implementation of a DEVS Solver

The algorithm in Figure (TODO) is based on the closure under coupling construction and can be used as a specification of a (possibly parallel) implementation of a DEVS solver or "abstract simulator" (TODO). In an atomic DEVS solver, the last event time t_L as well as the local state s are kept. In a coordinator, only the last event time t_L is kept. The next-event-time t_N is sent as output of either solver. It is possible to also keep t_N in the solvers. This requires consistent (recursive) initialization of the t_Ns. If kept, the t_N allows one to check whether the solvers are appropriately synchronized. The operation of an abstract simulator involves handling four types of messages. The $(x, from, t)$ message carries external input information. The $(y, from, t)$ message carries external output information. The $(*, from, t)$ and $(done, from, t_N)$ messages are used for scheduling (synchronizing) the abstract simulators. In these messages, t is the simulation time and t_N is the next-event-time. The $(*, from, t)$ message indicates an internal event * is due.When a coordinator receives a $(*, from, t)$ message, it selects an imminent component i^* by means of the tie-breaking function select specified for the coupled model and routes the message to i^*. Selection is necessary as there may be more than one imminent component (with minimum next remaining time).

When an atomic simulator receives a $(*, from, t)$ message, it generates an output message $(y, from, t)$ based on the old state s. It then computes the new state by means of the internal transition function. Note how in DEVS, output messages are only produced while executing internal events. When a simulator outputs a $(y, from, t)$ message, it is sent to its parent coordinator. The coordinator sends the output, after appropriate output-to-input translation, to each of the influencees of i^* (if any). If the coupled model itself is an influencee of i^*, the output, after appropriate output-to-output translation, is sent to the the coupled model's parent coordinator.

When a coordinator receives an $(x, from, t)$ message from its parent coordinator, it routes the message, after appropriate input-to-input translation, to each of the affected components.

When an atomic simulator receives an $(x, from, t)$ message, it executes the external transition function of its associated atomic model.

After processing an $(x, from, t)$ or $(y, from, t)$ message, a simulator sends a $(done, from, t_N)$ message to its parent coordinator to prepare a new schedule. When a coordinator has received $(done, from, t_N)$ messages from all its components, it sets its next-event-time t_N to the minimum t_N of all its components and sends a $(done, from, t_N)$ message to its parent coordinator. This

process is recursively applied until the top-level coordinator or root coordinator receives a $(done, from, t_N)$ message.

As the simulation procedure is synchronous, it does not support asynchronously arriving (real-time) external input. Rather, the environment or Experimental Frame should also be modelled as a DEVS component.

To run a simulation experiment, the initial conditions t_L and s must first be set in all simulators of the hierarchy. If t_N s kept in the simulators, it must be recursively set too. Once the initial conditions are set, the main loop described in Figure (TODO) is executed.

2.3 The pythonDEVS (pyDEVS) simulator

In this chapter we first present an implementation of the classical formalism. An early prototype of a DEVS Modeling and Simulation Package is then introduced. The chapter then moves on to discuss a real-time execution framework. The motivation behind real-time DEVS execution is discussed as well as the implications of 0-time semantics.

2.3.1 Design and implementation

This version of the DEVS Modeling and Simulation Package has been implemented using Python, an interpreted, very high-level, object-oriented programming language. The package consists of two files, the first of which (DEVS.py) provides a class architecture that allows hierarchical DEVS models to be easily defined. The simulation engine (SE) itself is implemented in the second file (Simulator.py). Based on the DEVS simulator described in (TODO), it uses the same message- passing mechanism. A detailed description of both the model architecture and the SE follows.

Model Architecture

The model architecture implemented in DEVS.py is a canvas from which hierarchical DEVS mod- els can be easily described. It consists of a number of classes arranged to capture the essence of hierarchical DEVS. A model is described in a dedicated file by deriving coupled and/or atomic- DEVS descriptive classes from this architecture. These atomic models are then arranged hierarchically through composition. Methods and attributes form the standard interface that allows an SE, such as the one described in the next sub-section, to interact with the instantiated DEVS model. Our main concern with the architecture are twofold: remain as consistent as possible with the original hierarchical DEVS definition, and maintain a flexible approach to DEVS so as to encourage model reusability through parameterization.

The class architecture is represented in Figure (TODO): BaseDEVS is the root class which provides basic functionalities common to both atomic and coupled models.

Two classes are inherited from BaseDEVS to deal with the specifics of atomic and coupled DEVS formalisms (Figure (TODO)). These three classes are all abstract in that they cannot be directly instantiated. Rather, a model is described by deriving descriptive classes from either the AtomicDEVS or the CoupledDEVS class. This provides them with a suitable constructor and overrides the default interface methods. Note that the constructors at every level of the class hierarchy have an active role. Hence, a descriptive class' constructor should always start by calling the parent class' constructor.

The constructor of the AtomicDEVS class merely initializes the myID attribute and provides a default initial value for the DEVS' total state, through the state and elapsed attributes. The remaining class definitions consist of default method declarations for the interface functions δ_{ext} (extTransition), δ_{int} (intTransition), ta (timeAdvance) and λ (outputFnc). These methods expect no parameter, and it is up to the modeler to be consistent with the corresponding functions' domain when overriding the methods. Except for outputFnc (which uses the poke method as described below), all the methods shall return a value compatible with the corresponding function range.

Since default values are provided for both attributes and methods, the minimal atomic-DEVS descriptive class is empty:

This atomic-DEVS is passive. It remains in its default state forever. A more interesting example is a generator, which sends a message (the integer 1 in this case) through its unique output port at a constant time interval:

As mentioned above, the outputFnc returns no value; instead it relies on the poke method to send message (second parameter) through the OUT output port (first parameter). The companion method, peek, returns the message on the input port that is given as a unique parameter, and is used exclusively in the extTransition function. Both poke and peek methods are defined in the AtomicDEVS class, and should not be overridden.

The CoupledDEVS class only has one method to override, the tie-breaking select function. This takes a list of sub-models which are in conflict. The select function should return the sub-model instance from this list who's transition is to fire next. All a CoupledDEVS sub-models should be included by passing their instance variables to the addSubModel function.

Coupling of ports is performed through the connectPorts method. Its first parameter is the source port and the second parameter is the destination port. A coupling is rejected and an error message issued if the coupling is invalid.

The source and destination ports are instances of the class Port. This class defines channels where events may pass between DEVS models. These channels are defined in the Port's inLine and outLine attributes.

CHAPTER 3

Methodology

The methodology employed in this thesis outlines the systematic approach taken to translate System of Communicating Concurrent Deterministic Finite-State Machines (SCCD) into Discrete Event System Specification (DEVS). This section provides an overview of the strategies, tools, and processes utilized to achieve the translation objectives.

The translation of SCCD to DEVS involves converting models represented in one formalism into another while preserving their essential behavioral characteristics. Such translation is critical for interoperability, model verification, and simulation interoperability between systems designed using different modeling paradigms.

In this chapter, we present the overarching methodology for translating SCCD to DEVS, detailing the steps involved, challenges encountered, and the rationale behind the chosen approaches. Additionally, we discuss alternative approaches considered during the development process, providing insights into why certain strategies were pursued while others were discarded.

The methodology encompasses various stages, including initial approach exploration, tool selection, implementation workflow, validation, and verification. Each stage plays a crucial role in ensuring the accuracy and effectiveness of the translation process.

Furthermore, the methodology emphasizes the importance of evaluating the translated DEVS models against predefined criteria to assess their fidelity to the original SCCD specifications. By adhering to a systematic methodology, this research aims to contribute to the advancement of model translation techniques and facilitate seamless integration between different modeling formalisms.

This chapter serves as a guide to the methodology adopted in this thesis, providing readers with a comprehensive understanding of the processes involved in translating SCCD to DEVS.

3.1 Translation Tools

The translation of System of Communicating Concurrent Deterministic Finite-State Machines (SCCD) to Discrete Event System Specification (DEVS) involves the use of specialized tools and techniques to facilitate the conversion process. In this section, we discuss the evolution of the translation tools employed in this research, from initial parsing methods to the adoption of a visitor pattern for seamless translation.

Initially, the translation process began with a straightforward approach of parsing the SCCD models represented in XML format line by line. While this method provided basic functionality, it quickly became apparent that it was not efficient or robust enough to handle the complexities of SCCD models and their translation to DEVS.

Upon further investigation, it was discovered that the SCCD compiler (TODO: reference) already implemented a visitor pattern for translating XML files into Python and Java representations. Leveraging this existing pattern proved to be a significant breakthrough in the translation process. By utilizing the visitor pattern, which allows for the traversal of complex data structures without modifying the structure itself, we were able to streamline the translation process and improve its efficiency and accuracy.

The visitor pattern provided a structured and modular approach to translating SCCD models to DEVS, enabling clear separation between the translation logic and the underlying model representation. This decoupling of concerns allowed for easier maintenance, extensibility, and reuse of the translation code.

An additional advantage of adopting the visitor pattern for XML to DEVS translation was its compatibility with the existing compiler infrastructure of SCCD. The modular nature of the visitor pattern facilitated seamless integration of the DEVS translation functionality into the existing compiler framework of SCCD. This integration allowed for the incorporation of DEVS translation as a standard feature within the SCCD compiler, providing users with a comprehensive toolset for model development and analysis.

By adopting the visitor pattern for XML to DEVS translation, we were able to overcome many of the limitations encountered with the previous parsing methods. The visitor pattern facilitated a more systematic and reliable approach to translating SCCD models to DEVS, ultimately enhancing the

quality and fidelity of the translated models.

In summary, the evolution of translation tools from basic XML parsing to the adoption of the visitor pattern reflects the iterative nature of the research process, wherein insights gained from experimentation and exploration lead to the refinement and improvement of translation techniques.

3.2 Considered Translation Approaches

3.2.1 First approach: Implementing everything into one AtomicDEVS

3.2.2 Second approach: Setting every component to AtomicDEVS

The first approach was to set every possible component (Controller, ObjectManager and classes) to an AtomicDEVS model. This proved to be impossible with the standard atomicDEVS because in SCCD, statecharts can be created at runtime. The classic AtomicDEVS does not allow to create AtomicDEVS objects at runtime and thus mapping statecharts to a corresponding AtomicDEVS is impossible.

CHAPTER 4

Implementation

Blablabla

4.1 Definitions

Some introductory section

CHAPTER 5

Examples

Blablabla

5.1 Definitions

Some introductory section

CHAPTER 6

Conclusions

Example of todo: TODO: Write conclusions here.

Bibliography

Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Square Bracket Associates, 2008.

Appendices

APPENDIX A

An appendix

An Appendix is just like another chapter.