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.

Ansymo
Antwerp Systems and Software Modelling

# 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, 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:** A (Harel) statechart, also known as a state machine diagram, is a behavioral diagram in the field of software engineering and systems design. It represents the various states that an object can be in throughout its lifetime and the transitions between those states in response to events.

9. **SCXML:** SCXML is a control flow language based on Harel State Charts. It offers powerful, application-independent control constructs, along with a plug-in capability that allows platforms to customize the language for specific domains.

# CHAPTER 2

# Background

## 2.1 Statecharts and Class Diagrams (SCCD)

### 2.1.1 The Statechart Formalism

Statecharts, originally introduced by D. Harel (??), are finite state machines extended with hierarchy and orthogonality (parallelism), allowing a complex system to be expressed in a more compact and elegant way.

In order to understand the basics of statecharts, it is useful to first look at higraphs. Higraphs, while normally used for formal mathematical specification, are also used as a basis for statecharts.

Statecharts can be represented as visual objects. While this thesis does not use a visual notation for the statecharts, it is usefull to understand the basics of statecharts. To understand the basics of statecharts, it is useful to first look at higraphs. Higraphs, while normally used for formal mathematical specification, are also used as a basis for statecharts.

#### Higraph

A higraph combines notions from several topovisual formalisms.

Topovisual formalisms are a class of graphical representations to model and visualize the topological structure of systems. They primarily focus on depicting the spatial relationships and connectivity between various components or entities within a system. The shapes of the visual objects is not that important in these formalisms.

One of these topovisual formalisms is the formalism of graphs, and the second is the notion of Euler circles, which later evolved into Venn diagrams.

A graph, in its most basic form, is simply a set of points, or nodes, connected by edges or arcs. Its role is to represent a (single) set of elements S and some binary relation R on them. The precise meaning of the relation R is part of the application and has little to do with the mathematical properties of the graph itself. Certain restrictions on the relation R yield special classes of graphs that are of particular interest, such as ones that are connected, directed, acyclic, planar, or bipartite.

A somewhat less widely used extension of graphs is the formalism of hypergraphs. A hypergraph is a graph in which the relation being specified is not necessarily binary. Formally, an edge no longer connects a pair of nodes, but rather a subset thereof.

TODO: Euler circles or Venn diagram

These previous formalisms can be used to create the Highgraph construct. Venn diagrams are used to account for set inclusion and exclusion. Each set is called a blob and is denoted using a rectangle with rounded edges. The nesting of curves denotes set inclusion.

## Statecharts

Statecharts are a higraph-based extension of standard state-transition diagrams, where the blobs represent states and arrows represent transitions. However, there are different kind of blobs and thus different kind of states. The different kind of states can be seen in Figure **??**. The first blob, Figure **??**, is called a BASIC-blob which is a state such that it has no sub-OR-blobs, sub-AND-blobs nor any sub-BASIC-blobs. OR-blobs are blobs that have one or more sub-OR-blobs. AND-blobs, Figure **??**, have orthogonal components. Orthogonal components of a blob are normally drawn with a dashed line and denote concurrent operations. An example combining all these different kind of blobs can be seen in Figure **??**. These blobs only introduce hierarchy in statecharts, the order to specify system dynamics in the diagrams are not yet specified. This is what transitions are used for. A transition is simply an arrow which indicates the system can change its current state from the source state of the transition to the destination state of the transition. Transitions are labeled as follows:

$$e[c]/a \tag{2.1}$$

This syntax denotes that the transition fires when event e occurs and condition c is true. The $/a$ usually means one of two different things, depending on the variant of statechart. In some definitions, $/a$ denotes that upon firing the transition, the a event is broadcast throughout the entire statechart, which may trigger more transitions. In the UML definition, $/a$ usually denotes a series of actions (such as computer code) to execute, upon firing the transition. The UML suggests the potential use of a function $GEN(e)$ , which can be one the many steps in a. This function is used to broadcast the event e. Most statechart variants combine these two aspects.

There are different kinds of transitions. First, there is the initial transition. It is the default transitions that fires upon entering any type of blob or the root. They force the diagram to be deterministic by denoting which is the current state upon entering an OR-blob. They are usually drawn as a small filled-in circle with the connected transition pointing towards a blob and may only be labeled with an action (no trigger event $e$).

### 2.1.2 State Chart XML (SCXML)

State Chart XML (SCXML) is a general-purpose event-based state machine language that combines concepts from CCXML and Harel State Tables.

#### Core constructs

**<scxml>** The top-level wrapper element, which carries version information. It can consist of the following attributes:

### 2.1.3 Statecharts and Class Diagrams (SCCD)

In this section the SCCD (a Statecharts and Class Diagrams hybrid) formalism and its SCCDXML representation, an extension of SCXML, is explained. SCCD facilitates the specification of complex timed, reactive, interactive discrete-event systems (e.g., complex user interfaces).

#### The SCCD language

The SCCD languages extends the SCXML, explained in subsection (TODO), by adding new features to its concrete syntax.

**Top-level Elements** The top-level element is a diagram. It has an input/output interface to communicate with its environment, it can optionally import library classes, and it holds a number of class definitions. One of these classes is the default, and is instantiated when the application is launched.

Listing (TODO) shows the top-level diagram of an application. It imports a library class that is used to draw the graphical elements on the screen, one input port called "input" which receives events when the user interacts with the UI (for example, pressing a key), and four classes, explained in the following subsections.

**Classes** Classes are the main addition of the SCCD language. They model both structure and behaviour—structure in the form of attributes and relations with other classes, behaviour in the form of methods, which access and change the values of attributes of the class, and an SCXML model, which constitutes the "modal" part of the system, modelling the control flow of the class's behaviour. At runtime, a class can be instantiated, which creates an object. Objects are initialized according to the class's constructor, and can be deleted, invoking the class's destructor. The relationships modelled between classes are instantiated at runtime in the form of links. They serve as communication channels, over which objects can send and receive events.

Listing 2 shows the definition of the "Ball" class. It defines a number of relations (discussed in the next paragraph), a constructor and destructor, a method that moves the ball to a new position, and an SCXML model that consists of four states. It can optionally also define private input ports and output ports. In this case, the ball defines a private input port, that allows the environment to send events that are only meant for a particular ball. For example, when the user left-clicks on a ball to select it, that event should only be sent to that specific instance.

**Relationships** Classes can have relationships with other classes. There are two types of relationships: associations and inheritance.

An association is defined between a source class and a target class, and has a name. It allows instances of the source class to send events to instances of the target class by referencing the association name. An association has a multiplicity, defined as a minimal cardinality $c_{min} \in N$ and a maximal cardinality $c_{max} \in N_{>0} \cup \{inf\}$. They control how many instances of the target class have to be minimally associated to each instance of the source class, and how many instances of the target class can be maximally associated to each instance of the source class, respectively. Each time an association is created, it results in a link between the source and target object. This link gets a unique identifier, allowing the source object to reference the target, for example to send events.

An inheritance relation results in the source of the relation to inherit all attributes and methods from the target of the relation. Specialisation of modal behaviour (i.e., (parts of) the SCXML model of the superclass) is currently not supported.

Listing (TODO) shows the relationships of the "Window" class. It has an association to its parent, the main application. Exactly one instance of that link has to exist between each "Window" instance and the main application. It is additionally associated to a number of buttons and balls, and inherits from the library class "UIWidget", allowing it to be drawn on screen.

**Events** Events in SCCD are strings. They are accompanied by a number of parameter values: the sender is obliged to send the correct number of values, and the receiver declares the parameters when catching the event. Each parameter has a name, that can be used as a local variable in the action associated with the transition that catches the event.

With the addition of a public input/output interface using ports, as well as classes and associations, comes the need for scoping events. In traditional SCXML models, an event is sensed by the Statecharts model that generated it. SCCD adds the ability to transmit events to class instances and to output ports. In particular, the raise tag was extended with a scope attribute, that can take on the following values:

- **local:** The event will only be visible for the sending instance.
- **broad:** The event is broadcast to all instances.
- **output:** The event is sent to an output port and is only valid in combination with the output attribute, which specifies the name of the output port.
- **narrow:** The event is narrow-cast to specific instances only, and is only valid in combination with the target attribute, which specifies the instance to send the event to. For example, an instance of the ”Window” class can narrow-cast an event by sending the event to a specific instance of the ”Ball” class, identified by a unique link identifier.
- **cd:** The event is processed by the object manager. See the next section for more details.

Listing (TODO) presents a transition modelled on the ”Button” class. It reacts to the user left-clicking the button (represented by an event sent on the button input port). The button reacts by notifying its parent that it was clicked.

### The Object Manager

At runtime, a central entity called the object manager is responsible for creating, deleting, and starting class instances, as well as managing links (instances of associations) between class instances. It also checks whether no cardinalities are violated: when the user creates an association, it checks that the maximal cardinality is not violated, and when the user deletes an association, it check whether the minimal cardinality is not violated. As mentioned previously, instances can send events to the object manager using the ”cd” scope. The object manager can thus be seen as an ever-present, globally accessible object instance, although it is implicitly defined in the runtime, instead of as a SCCD class.

When the application is started, the object manager creates an instance of the default class and starts its associated Statecharts model. From then on, instances can send several events to the object manager to control the set of currently executing objects. The object manager accepts four events. We list them below, including the parameters that have to be sent as part of the event:

- **create_instance(association name, class name, args*):**
- **delete_instance(link_ref):**
- **start_instance(link_ref):**
- **associate_instance(source_ref, association_name, target_ref):**

## The SCCD compiler

The semantics of an SCCD model are loosely based on the agent model, where each instance of a class can be seen as an agent that communicates with other agents through its input/output interface, and its autonomous behaviour controlled by its Statecharts model. The compiler generates appropriate code that continuously executes the system by allowing each agent to execute a step, which optionally generates output that can be sensed by the other agents.

The compiler supports two programming languages, Javascript and Python, and options for the statecharts semantics. Supporting multiple languages is a major advantage, as one can imagine developing an application in SCCD and generating code for multiple languages from the same model. The generated code would exhibit identical behaviour for each implementation language, such as a web-based application (implemented in HTML/Javascript) and a desktop application (implemented in, for example, Python).

SCCD has one semantic definition. There are, however, many platforms on which the code generated from an SCCD model can be run. The runtime platform provides essential functions used by the runtime kernel, such as the queueing of events and the scheduling of (timed) events. Three runtime platforms are supported. A platform holds a list (or queue) of events, and they differ in the way they handle events generated during execution. The kernel attempts to run the SCCD model in real-time, meaning that the delay on timed transitions is intepreted as an amount of seconds. Raising of events and untimed transitions are executed as fast as possible. Figure (TODO) presents an overview of the three platforms, and how they handle events.

The most basic platform, available in most programming languages, is based on threads. Currently, the platform runs one thread, which manipulates a global event queue, made thread-safe by locks. Input from the environment is handled by obtaining this lock, which the kernel releases after every step of the execution algorithm. This allows external input to be interleaved with internally raised events. Running an application on this platform can interfere with other scheduling mechanisms, such as a UI module, however.

To overcome this interference problem, the event loop platform reuses the event queue managed by an existing UI platform, such as Tkinter. The UI platform provides functions for managing time-outs (for timed events), as well as pushing and popping items from the queue. This results in a seamless integration of both Statecharts events and external UI events, such as user clicks: the UI platform is now responsible for the correct interleaving.

The game loop platform facilitates integration with game engines (such as the open-source Unity engine), where objects are updated only at predefined points in time. In the "update" function, the kernel is responsible for checking the current time (as some time has passed since the last call to the "up-date" function), and process all the events generated by objects. This means that events generated in between two of these points are not processed immediately, but queued and their processing delayed until the next processing time.

## Semantics

The Statecharts language has been around for a long time. In that time, its basic structures have almost not changed. In its original definition [(TODO)], Harel left many of the semantic choices undefined. Since then, many semantics have been defined, such as the one used in Statemate [(TODO)]. More recently, Esmaeilsabzali et al. [(TODO)] have performed a study of big-step modelling languages, such as Statecharts, and defined a set of semantic variation points, with which the different Statecharts execution semantics can be classified. Central to their discussion is the notion of a "big step". The execution of a Statecharts model is a sequence of big steps. A big step is a unit of interaction between a model and its environment. A big step takes input from the environment (at the beginning of the big step), and produces output to the environment (after the big step has taken place). Input cannot change during the big step. A big step consists of 0 or more small steps. A small step is an unordered set of 1 or more transition executions, but in our case, a small step always consists of exactly one transition execution. Small steps are grouped in so-called combo steps. A combo step is a maximal sequence of small steps, such that it only contains transitions that are orthogonal to each other.

The SCCD compiler allows to choose which semantics to use based on a number of semantic variation points. This gives modellers more control to fine-tune the application to their needs. The semantic variation points are:

- **Big Step Maximality** specifies when a big step ends: either after one combo step executed (Take One), or when no more combo steps can be executed (Take Many).

- **Internal Event Lifeline** specifies when an internally raised event becomes available: either in the next small step (immediately), in the next combo step (which only makes sense in combination with the Take Many option), or the event is queued and treated as an external event (making it available in the next big step).

- **Input Event Lifeline** specifies when an input event is available during a big step: either throughout the first small step, the first combo step, or throughout the whole big step.
- **Priority** specifies what to do when two transitions are enabled at the same time, where the source state of one of the transitions is the ancestor of the source state of the other transition. Either the transition of the ancestor gets priority (Source-Parent), or the transition of the (indirect) child gets priority (Source-Child).

## 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 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. 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:

$$atomicDEVS \equiv \langle S, ta, \delta_{int}, X, \delta_{ext}, Y, \lambda \rangle \tag{2.2}$$

The time base $T$ is continuous and is not mentioned explicitly

$$T = R \tag{2.3}$$

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)

$$S = \times_{i=1}^{n} S_i. \tag{2.4}$$

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

$$ta : S \rightarrow R^{+}_{0,+\infty}. \tag{2.5}$$

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

$$\delta_{int} : S \rightarrow S \tag{2.6}$$

models the transition from one state to the next sequential state. $\delta_{int}$ 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)

$$Y = \times_{i=1}^{l} Y_i. \tag{2.7}$$

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

$$\lambda : S \rightarrow Y \cup \{\phi\} \tag{2.8}$$

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 $\lambda$. At all other times, the non-event $\phi$ 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

$$Q = \{(s, e) | s \in S, , 0 \leq e \leq ta(s)\} \tag{2.9}$$

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 $\sigma$ in a state is used:

$$\sigma = ta(s) - e \tag{2.10}$$

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)

$$X = \times_{i=1}^{m} X_i \tag{2.11}$$

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 $\Omega$ contains all admissible input segments $\omega$

$$\omega : T \rightarrow X \cup \{\phi\} \tag{2.12}$$

In discrete-event system models, an input segment generates an input event different from the non-event $\phi$ 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

$$\delta_{ext} : Q \times X \rightarrow S \tag{2.13}$$

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, $\delta_{ext}$ 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 $\delta_{ext}$ 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.

$$coupledDEVS \equiv \langle X_{self}, Y_{self}, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\}, select \rangle \tag{2.14}$$

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

$$\{M_i | i \in D\}. \tag{2.15}$$

Each of the components must be an atomic DEVS

$$M_i = \langle S_i, ta_i, \delta_{int,i}, X_i, \delta_{ext,i}, Y_i, \lambda_i \rangle, \forall i \in D. \tag{2.16}$$

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

$$\{I_i | i \in D \cup \{self\}\}. \tag{2.17}$$

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 $self$:

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

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:

$$\forall i \in D \cup \{self\} : i \notin I_i. \tag{2.19}$$

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:

$$\{Z_{i,j} | i \in D \cup \{self\}, j \in I_i\}, Z_{self,j} : X_{self} \rightarrow X_j, \forall j \in D, Z_{i,self} : Y_i \rightarrow Y_{self}, \forall i \in D, Z_{i,j} : Y_i - \tag{2.20}$$

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:

$$select : 2^D \rightarrow D \tag{2.21}$$

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

$$select(E) \in E. \tag{2.22}$$

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

$$\langle X_{self}, Y_{self}, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\}, select \rangle \tag{2.23}$$

with all components $M_i$ atomic DEVS models

$$M_i = \langle S_i, ta_i, \delta_{int,i}, X_i, \delta_{ext,i}, Y_i, \lambda_i \rangle, \forall i \in D \tag{2.24}$$

the atomic DEVS

$$\langle S, ta, \delta_{int}, X, \delta_{ext}, Y, \lambda \rangle \tag{2.25}$$

is constructed.

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

$$S = \times_{i \in D} Q_i \tag{2.26}$$

where

$$Q_i = \{(s_i, e_i) | s \in S_i, 0 \leq e_i \leq ta_i(s_i)\}, \forall i \in D. \tag{2.27}$$

The time advance function $ta$

$$ta : S \to R^+_{0,+\infty} \tag{2.28}$$

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

$$ta(s) = \min\{\sigma_i = ta_i(s_i) - e_i | i \in D\}. \tag{2.29}$$

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

$$IMM(s) = \{i \in D | \sigma_i = ta(s)\}. \tag{2.30}$$

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

$$select : 2^D \to DIMM(s) \to i^* \tag{2.31}$$

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

$$\lambda(s) = Z_{i^*,self}(\lambda_{i^*}(s_{i^*})), if self \in I_{i^*}. \tag{2.32}$$

If the output of $i^*$ is not connected to the output of the coupled model, the non-event $\phi$ can be generated as output of the coupled model. As $\phi$ 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:

$$\delta_{int}(s) = (..., (s'_j, e'_j), ...), where(s'_j, e'_j) = (\delta_{int,j}(s_j), 0), forj = i^*, = (\delta_{ext,j}(s_j, e_j + ta(s), Z_{i^*,j})) \tag{2.33}$$

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:

$$\delta_{ext}(s, e, x) = (..., (s'_i, e'_i), ...) \tag{2.34}$$

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 $t_N$s. 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 models 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 $\delta_{ext}$ (extTransition), $\delta_{int}$ (intTransition), $ta$ (timeAdvance) and $\lambda$ (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.

Consider the example of the situation illustrated in Figure (TODO). There exists a descriptive class SomeDEVS for a DEVS (either atomic or coupled) with an input and output port locally known as IN and OUT. We want to connect the input port to the output port of the SimpleGenerator atomic- DEVS described above: both DEVS must of course be children of the same coupled-DEVS for the coupling to be performed. The coupled-DEVS descriptive class is defined below.

Note that the parameter to the constructor is used to parameterize the

SimpleGenerator atomic- DEVS. As for atomic-DEVS, the constructor first calls the parent class' constructor. Note also that the coupled-DEVS itself has an output port. The first coupling is an internal coupling, while the second is an external output coupling. This is a complete definition for a coupled-DEVS descriptive class. The default select function is used.

Once all the descriptive classes in the hierarchical DEVS model have been defined, the whole model can be build by instantiating the root DEVS. This is possible since the hierarchical representation of the model is built by composition rather than aggregation.

As a final warning, note that recursive definitions are illegal, since they are incompatible with a tree structure. As a trivial example, a coupled-DEVS' descriptive class SomeCoupledDEVS cannot call in its constructor the addSubModel method with an instance of SomeCoupledDEVS. This is mentionned since such recursive constructs will not be detected.

### Simulation Engine

CHAPTER 3

# Methodology

The methodology employed in this thesis outlines the systematic approach taken to translate SCCD into PyDEVS. This section provides an overview of the strategies, tools, and processes utilized to achieve the translation objectives. The translation 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.

We also present the overarching methodology for translating SCCD to PyDEVS, 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.

## 3.1 Translation Tools

The translation of SCCD to 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.

### 3.1.1 Parsing line by line

Initially, in the endeavor to translate SCCD to DEVS, the SCXML file was parsed meticulously, line by line. However, as we delved deeper into the process, we encountered challenges that made this approach increasingly cumbersome. The complexity of the SCXML structure made it difficult to extract the necessary information efficiently, leading to inefficiencies and potential errors in the translation process. Recognizing the need for a more robust and structured approach, we sought alternative solutions to streamline our workflow.

### 3.1.2 Visitor Pattern

Upon further investigation, it was discovered that the SCCD compiler already integrates a robust visitor pattern for translating SCXML files into Python and/or Java representations. This discovery marks a significant breakthrough in the translation process, as it not only facilitates the traversal of intricate data structures without altering the structure itself but also enables seamless integration of new languages or platforms.

The compiler generates an Abstract Syntax Tree (AST), which serves as the foundation for the visitor to traverse each node systematically. This thesis elucidates the profound utility of this approach. Firstly, the visitor parses the tree from the root to the leaf nodes, thereby enabling a chronological description of the output. This sequential traversal allows for the orderly declaration of imports and their subsequent usage, streamlining the code generation process. Furthermore, by compartmentalizing each node, the visitor pattern facilitates the generation of code independent of other nodes, enhancing the modularity and efficiency of the SCCD framework.

Moreover, the visitor pattern's versatility extends to accommodating various platforms, including Thread, UI Event Loop, and Game Loop, for both Python and Java. This inherent adaptability underscores the feasibility of integrating the DEVS formalism seamlessly into the SCCD framework. By developing a visitor tailored for DEVS and integrating it as a platform, extending the SCCD compiler's capabilities becomes a straightforward endeavor.

## 3.2 Considered Translation Approaches

### 3.2.1 First approach: Implementing everything into one AtomicDEVS

The first approach is to set the entire SCCD code, which includes the controller, ObjectManager and every class, to one AtomicDEVS model. While this could be possible, it was chosen not to do this for several reasons.

### 3.2.2 Second approach: Setting every component to AtomicDEVS

Another approach is 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.

### 3.2.3 Chosen approach: TODO

The approach that is chosen is to have a Controller component as a CoupledDEVS object and the ObjectManager and class type as an AtomicDEVS object. What is different with the approach explained in **??** is that the class type is an AtomicDEVS an not every instance of that type.

#### Controller component

This component originated from the SCCD runner. While the controller in the SCCD runner is created to control the runtime of the objects, the controller created in PyDEVS serves merely as a structural object. In the controller the submodels and their couplings are described. This is perfect because the classes in SCCD need to be coupled to each other and the ObjectManager. To couple them, the SCCD to PyDEVS compiler will generate a controller and link every AtomicDEVS to the AtomicDEVS of the ObjectManager.

#### ObjectManager component

The ObjectManager has the same use cases as the ObjectManager in SCCD. The only difference is that objects can not explicitely be created in this component but rather the Event to do an action gets now forwarded do the specific class component an instance should be created, deleted from. This way the ObjectManager itself still decides if the instance can/may be made or not. Instances can still send events to the ObjectManager with the c̈dšcope.

When the application is started, the default class

the object manager now thus not create an instance of the default class and starts its associated Statecharts model. But it sends a s̈tart_instanceË vent to the default class allowing the default class to start the instance that is already created in the

From then on, instances can send several events to the object manager to control the set of currently executing objects. The object manager accepts four events. We list them below, including the parameters that have to be sent as part of the event:

### Class component

### Class Instance component

# 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. @InbookBarnett2017, author="Barnett, Jim", editor="Dahl, Deborah A.", title="Introduction to SCXML", bookTitle="Multimodal Interaction with W3C Standards: Toward Natural User Interfaces to Everything", year="2017", publisher="Springer International Publishing", address="Cham", pages="81–107", abstract="SCXML is a control flow language based on Harel State Charts. It offers powerful, application-independent control constructs, along with a plug-in capability that allows platforms to customize the language for specific domains. This paper offers an overview of the language along with examples of its use.", isbn="978-3-319-42816-1", doi="10.1007/978-3-319-42816-$1_5$", $url =$ "$https : //doi.org/$10.1007/978 − 3 − 319 − 42816 − $1_5$" @articleHAREL1987231, title = Statecharts: a visual formalism for complex systems, journal = Science of Computer Programming, volume = 8, number = 3, pages = 231-274, year = 1987, issn = 0167-6423, doi = https://doi.org/10.1016/0167-6423(87)90035-9, url = https://www.sciencedirect.com/science/article/pii/0167642387900359, author = David Harel, abstract = We present a broad extension of the conventional formalism of state machines and state diagrams, that is relevant to the specification and design of complex discrete-event systems, such as multi-computer real-time systems, communication protocols and digital control units. Our diagrams, which we call statecharts, extend conventional state-transition diagrams with essentially three elements, dealing, respectively, with the notions of hierarchy, concurrency and communication. These transform the language of state diagrams into a highly structured and economical description language. Statecharts are thus compact and expressive—small

diagrams can express complex behavior—as well as compositional and modular. When coupled with the capabilities of computerized graphics, statecharts enable viewing the description at different levels of detail, and make even very large specifications manageable and comprehensible. In fact, we intend to demonstrate here that statecharts counter many of the objections raised against conventional state diagrams, and thus appear to render specification by diagrams an attractive and plausible approach. Statecharts can be used either as a stand-alone behavioral description or as part of a more general design methodology that deals also with the system's other aspects, such as functional decomposition and data-flow specification. We also discuss some practical experience that was gained over the last three years in applying the statechart formalism to the specification of a particularly complex system.

# Appendices

# APPENDIX A

# An appendix

An Appendix is just like another chapter.