
A Modelled Visual Multi-Paradigm Modelling and Enactment Environment for Workflow Modelling

Author:
Addis GEBREMICHAEL

Promotor:
Prof. Dr. Hans VANGHELUWE

MASTER'S THESIS

*A thesis submitted in fulfillment of the requirements
for the degree of Master of Science: Computer Science
in the
Modelling, Simulation and Design Lab
Department of Mathematics and Computer Science*

August 25, 2017

Abstract

In multi-paradigm modelling, workflow (process) modelling is used to precisely describe the order of execution of model operations applied on a set of formalisms. The UML 2.0 Activity Diagrams, used in this thesis, inherently possess design patterns for modelling concurrency and synchronization, among others, at a high level of abstraction. In addition, activities in a process model (PM) can orchestrate with external services. Such behaviours of workflow modelling require that they should be explicitly modelled. This thesis, thereby, realizes the semantics of a PM using Statecharts Class-diagram (SCCD) formalism. SCCD facilitates the specification of dynamic-structure systems that are timed, autonomous and reactive. Moreover, SCCD is a low-level language where its semantic domain is already defined and simulated. Denotationally mapping PM onto SCCD, thus, provided explicit modelling of its behaviour such as concurrency, and essentially enabled the enactment of the PM itself. Furthermore, expressing the behaviour of PM activities in an explicitly modelled SCCD allowed interactivity of a PM with an external service and/or a user. A visual modelling environment that integrates the graphical concrete syntax of both languages (PM and SCCD) is developed using Tkinter on top of the Modelverse, which is a multi-paradigm modelling and simulation environment. The user interface behaviour of the visual editor is also explicitly modelled in SCCD.

Acknowledgments

I would like give many thanks to Prof. Vangheluwe for introducing me to the state of the art in the field of software engineering. I am also very grateful for the MSDL associates for constantly assisting me throughout this research. Lastly, i would like to thank God, my mother, close family and friends, without whom i would not have managed to come any far.

Contents

1	Introduction	7
1.1	Context	7
1.2	Problem statement	7
1.3	Expected contributions	8
1.4	Outline	8
2	Background	9
2.1	Domain Specific Modelling Languages (DSMLs)	9
2.2	Model Operations	10
2.3	The FTG+PM Language	11
2.4	Statecharts Class-diagram (SCCD)	13
2.5	Petri-nets	18
3	Mapping PM onto SCCD	20
3.1	Mapping Approaches	20
3.2	Data Management	26
3.3	Implementation	26
4	Visual Editor	34
4.1	Design Choices and Application Architecture	34
4.2	Process Model Editor	36
4.3	SCCD Editor	37
5	Example	40
5.1	Case Study	41
5.2	External Service Interaction	43
6	Conclusion	46
6.1	Future Works	46

List of Figures

2.1	Graphical concrete syntax of a textual model represented by "a+b=c"	9
2.2	An example of a Petri-net model transformation rule	10
2.3	Graphical concrete syntax of <i>ruleBlocks</i> in MoTifs language	11
2.4	FTG+PM for the development and verification of a power window case study, adopted from [23]	12
2.5	A small portion of the UML 2.0 Activity Diagram meta-model	13
2.6	A subset of UML 2.0 Activity Diagram modelling constructs	13
2.7	Two basic states connected by a transition	14
2.8	Composite state B encapsulating two basic states C and D.	15
2.9	Semantics of Statecharts in (a) by "flattening".	15
2.10	Depth/hierarchy in Statecharts.	15
2.11	Parallel state Y with two orthogonal regions A and B	15
2.12	When the transition to the history state is triggered, the state of A will be restored to its last recorded state.	16
2.13	This figure illustrates the relation between classes in a Class-diagram and the Statecharts that describe their behaviour.	17
2.14	An enabled transition.	18
2.15	A disabled transition.	18
2.16	An example of a Petri-net model before (a) and after (b) firing the transition.	18
2.17	A WF-net is a Petri net with a source and a sink place. The goal is that a process initiated via place <i>source</i> successfully completes by putting a token in place <i>sink</i> [17].	19
3.1	Workflow modelling at different levels of abstraction	20
3.2	A simple PM instance expressed using a subset of UML 2.0 ADs constructs	21
3.3	Sequential mapping of PM in figure 3.2 onto the Orchestrator SCCD	22
3.4	Interleaving branches in parallel regions	23
3.5	A branch exiting parallel region without synchronizing	23
3.6	Process model instances with complex behaviour	23
3.7	The Orchestrator SCCD using generic mapping	24
3.8	Orthogonal components of the Orchestrator parallel state in figure 3.7, mapping an Executable, Fork and a Join node	25
3.9	Orthogonal components of the Orchestrator parallel state in figure 3.7, mapping a Start, Decision and a Finish nodes	25
3.10	Data-flow representation at design-time using UML 2.0 Activity Diagram constructs	26
3.11	Process Model abstract syntax expressed in Class-diagram meta-language	27
3.12	Process Model graphical concrete syntax adopted by the developed editor	27
3.13	Statecharts abstract syntax expressed in Class-diagram meta-language.	28
3.14	SCCD graphical concrete syntax adopted by the developed editor	28
3.15	Model transformation which creates the Orchestrator SCCD enclosed in an <i>Atomic</i> rule-block	30
3.16	'ModifyForkToFork' rule	30
3.17	'RemoveForkToFork' rule	30

3.18	Model transformation rule in (a) enclosed in <i>Atomic</i> rule-block and (b) in <i>ForAll</i> rule-block	30
3.19	Model transformation rule enclosed in a <i>ForAll</i> rule-block for mapping an <i>Executable</i> activity to an orthogonal component in the Orchestrator SCCD parallel state.	31
3.20	Model transformation rule enclosed in <i>ForAll</i> rule-block used for mapping <i>Fork</i> to an orthogonal component in the Orchestrator SCCD parallel state	32
3.21	Model transformation rule enclosed in <i>ForAll</i> rule-block used for mapping <i>Join</i> node to an orthogonal component in the Orchestrator SCCD parallel state	33
4.1	An overview of the implemented application architecture.	34
4.2	The developed visual editor for PM.	36
4.3	Prompting user for selecting association types defined in the abstract syntax of PM	37
4.4	SCCD visual editor for Activity_A in figure 4.2	38
4.5	State attributes editor in SCCD	39
4.6	Multiple angle points of an edge used for editing its position	39
5.1	A process model developed in the visual editor for the power window case study.	40
5.2	The executable SCCD model generated after mapping PM to the Orchestrator SCCD	42
5.3	The behaviour of Combine activity in figure 5.1 modelled in SCCD	43
5.4	EBNF grammars of LoLA's file format.	44
5.5	SCCD model in the developed visual editor which depicts the behaviour of an external service interaction.	45

Listings

3.1	Schedule of model transformation rules used for mapping PM to SCCD	29
3.2	Model transformation rule in the Modelverse	32
4.1	SCCD instance of a Button type used for developing the visual editor expressed in SCCDXML notation	35

1

Introduction

This chapter presents an introduction to the context of this thesis and the problems it addresses.

1.1 Context

Modeling takes a prominent role in the development phase of complex and software-intensive systems. In Model-driven Engineering (MDE), Multi-Paradigm Modelling (MPM) advocates to use the most appropriate formalism(s) to explicitly model all relevant aspects of a system including the process, at the right level of abstraction [5]. Hence, inherent to the nature of MPM is the use of a wide range of appropriate formalisms, i.e. Domain Specific Modelling Languages (DSMLs), and associated operations used to manipulate the formalisms. These two aspects can be ideally integrated in the Formalism Transformation Graph and Process Model (FTG+PM) framework [9]. FTG lays down the relationships among the multitude of DSMLs and model operations used for the development of a particular system(s) in a given domain. A Process Model is used in conjunction with the FTG to model the order of execution of model operations applied on a set of formalisms, and ultimately for a particular motive. In that regard, a process model captures and exhaustively describes the control-flow and data-flow of the process MPM is being applied to, which can be used not merely for documentation purposes but also for its enactment, analysis and optimization. The PM used in this thesis is a subset of the Unified Modelling Language (UML) 2.0 Activity Diagrams, where the UML is a standard in the MDE community [11]. The Modelverse is a prototype tool with the support for all aspects of MPM including the integration and enactment of FTG+PM [23].

1.2 Problem statement

Process models have design patterns for modelling concurrency and synchronization behaviour, among others, at a high-level of abstraction. In the enactment of FTG+PM in the Modelverse, such behaviours of PM are not explicitly modelled. Although control-flow behaves according to the language constructs, activities in a parallel region, for instance, have sequential execution which deviates from its actual semantics.

PM activities can also coordinate with external services. While requesting for a given service, the PM shall autonomously handle its behaviour, such as lack of service response and/or connection timeouts. This requires explicit modelling of its behaviour with the support for timed events, triggering of events and handling them autonomously, and reacting to external events.

Hence, this thesis aims to assess how it can be possible to model workflows (processes) at different levels of abstraction, such that it enables to explicitly model its control-flow behaviour and essentially enact the PM itself. In addition, this thesis explores how we can explicitly model the behaviour of PM activities that interact with an external service and/or a user. In support of modelling at multiple levels of abstraction (of PM with a low-level language), an integrated visual modelling environment is desired. To deal with the inherent complexities of user interface interaction behaviour, the visual editor is to be explicitly modelled with an appropriate formalism.

1.3 Expected contributions

The principal objective of this thesis is to assess and demonstrate how it can be possible to define the semantics of PM in a low-level language, such that control-flow and activities of PM can be explicitly modelled and essentially enacted. In that regard, this thesis uses Statecharts Class-diagram (SCCD) as the appropriate low-level language where its semantic domain is already defined and simulated. SCCD is a hybrid formalism which combines the object-oriented expressiveness of Class-diagrams with the behavioural discrete-event characteristics of Statecharts. This, in turn, facilitates the specification of dynamic-structure systems that are timed, autonomous and reactive.

Upon successful completion of this thesis, we will discuss how we can denotationally map PM onto SCCD, and demonstrate the enactment of PM in which its semantics is given in SCCD. Moreover, by encapsulating the behaviour of PM activities in an explicitly modelled SCCD, we will show how the PM autonomously behaves while interacting with an external service and/or a user. The graphical concrete syntax of both languages can be used to develop an integrated visual modelling environment where it can be possible to model workflows at different levels of abstractions. The visual editor, to be developed in Tkinter, is suppose to interact with the Modelverse as a model repository, and in support of a multi-paradigm modelling and simulation environment. The user interface of the editor is to be explicitly modelled in SCCD, as it entails a behaviour that is timed, autonomous, reactive and has a dynamic-structure.

1.4 Outline

The remainder of this thesis is structured in the following manner. The next chapter gives a background introduction on the three main aspects of MPM, i.e. language engineering (DSMLs), model operations, and process modelling. In addition, it also gives a concise introduction to the SCCD formalism and Petri-nets. Chapter 3 has an extended discussion on the approaches explored and the implementation of mapping PM onto SCCD. Chapter 4 explains the design choices made and the implementation of the developed integrated visual editor. Chapter 5 has a case study as an example for demonstrating the semantics of PM given in SCCD, and elaborates the behaviour of an external service and/or a user interaction of PM activity, which is explicitly modelled in SCCD. The external service is an optimal reachability analysis of a Petri-net model available outside of the FTG. Finally, chapter 6 concludes this thesis by summarizing the research and reflecting on our findings, followed by future works that can further be incorporated as part of this research.

2

Background

This chapter gives an introduction to model-driven engineering concepts and techniques in which the research in this thesis finds relevant. It starts by explaining the core concepts behind MPM, that includes domain-specific modelling languages, model operations and process modelling. It also highlights the support of these three aspects of MPM by a prototype tool used in this research, i.e. the Modelverse [23]. Following is a concise introduction to the SCCD formalism and Petri-nets. While discussing Petri-nets, this chapter also introduces to the idea of workflow-nets. *Workflow-nets* are a particular class of Petri-nets that have become one of the standard ways to model and analyze workflows [17].

2.1 Domain Specific Modelling Languages (DSMLs)

Model-driven engineering (MDE) is currently the mainstream approach to the development of complex and software-intensive systems. Its underlying philosophy starts by developing *domain-specific* structural and behavioral models of the system under development [9]. *Domain-specific* means that initially models of the system should be described in a language close to the domain being addressed. This essentially reduces the accidental complexity of a software engineering process by shifting the specification from computing concepts to conceptual models or abstractions within the problem domain. Hence, DSMLs are more user intuitive, easy to learn, and maximally constrain the user, allowing to reduce errors. Models created in DSMLs are used for documentation, analysis, simulation and code generation for multiple platforms [20].

A DSML can be fully defined by [8]:

- Its *abstract syntax*, defining the DSML constructs and their allowed combinations. This information is typically captured in a meta-model, where the DSML being developed conforms to.
- Its *concrete syntax*, specifying the visual representation of the different constructs. This representation can be either graphical (using icons), or textual.
- Its *semantics*, defining the meaning of models created in the domain. This encompasses both the semantic domain (what is its meaning) and the semantic mapping (how to give it meaning).

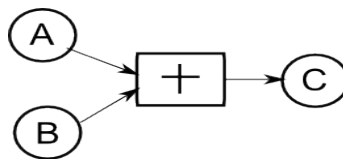


Figure 2.1: Graphical concrete syntax of a textual model represented by "a+b=c"

This can be more elaborated with a trivial, and simple to understand, example as follows. Figure 2.1 and the string of characters "a+b=c" can both be seen as an equivalent visual representation of the same model, graphically and textually respectively. This visual representation of the model is for the abstract syntax concept of an operation (addition) that takes in signal 'a' and signal 'b', followed by another operation (assignment) which produces signal 'c'. The abstract syntax is a representation in memory

with some kind of data structure for the variables and operations used in this model. Nonetheless, the semantics or "meaning" of the model is yet to be defined in the semantic domain, which, if assumed to be, is the set of real numbers (i.e. what the variables evaluates to), with the semantic mapping being the computation (algorithmic simulation) of the operations (i.e. how the variables are evaluated).

The Modelverse allows for language engineering through meta-modelling, where languages are themselves instances of the meta-language and can thus be seen as models themselves [23]. This offers users a multitude of possibilities to create new DSMLs without modifying the tool. The Modelverse is also a model repository in a sense that it runs as a service and enables users to share models and languages.

2.2 Model Operations

Another aspect of MPM is model operations. Model operations are used to manipulate models for some use. It can be for defining the semantics of a modelling language or can simply be for model management operations such as model merge and split, which is irrespective of their semantics. Model operations define the semantics of a model instance denotatively (by mapping onto another language where its semantic is already defined) or operationally (by directly executing the model producing execution trace or simulation) [23]. The core of this thesis research is to define the semantic domain of a process model denotatively, by mapping it onto SCCD, where SCCD has an existing semantics and is executable.

Model operations are commonly expressed using either action language code or model transformations. This thesis research uses model transformations, which are based on the work in the graph transformation community [12]. In this context, transformation rules consists of the smallest units in which a change from source model to target model can be defined. Rules use patterns and graph rewriting algorithms to ultimately define CRUD operations on a model. Patterns can be of two types, pre-condition and post-condition. Pre-condition patterns have a read-only operation and are used to determine when and which elements of the source model are to be transformed. While a post-condition has a create, update and delete operation and determines how the source model can be transformed. Model transformations are models in themselves and can be visualized as in figure 2.2. This

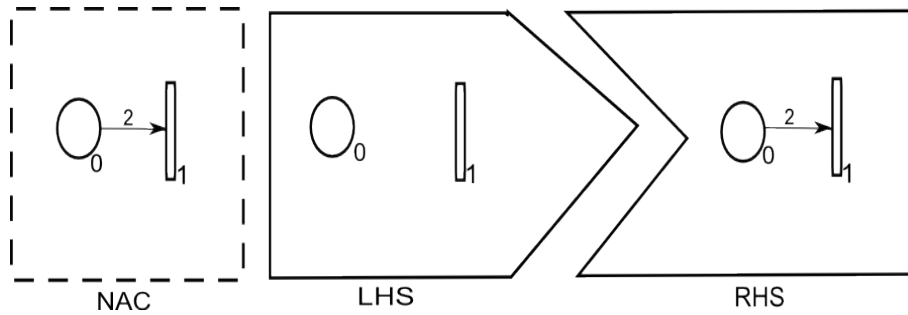


Figure 2.2: An example of a Petri-net model transformation rule

representation depicts three patterns out of which the first two are pre-condition patterns and the last a post-condition. The pattern in the left hand side (LHS) contains all the positive reads while the pattern in the negative application condition (NAC) holds the negative reads of this rule. Both patterns help in determining the pattern to be found. When found, the pattern is replaced by the right hand side (RHS), which represents the result after the rewriting phase. The elements to be deleted are elements that have a positive read, and thus appear in the LHS, but do not appear in the RHS. Likewise, elements that are added appear in the RHS, but not in the LHS. In the example on figure 2.2, a petri-net model with a place (labelled 0) and a transition (labelled 1) pattern is searched for in the LHS. If the pattern is found given the NAC is not satisfied, i.e. an arc is not created for that pattern, it updates the model in the RHS by adding an arc (labelled 2).

After having defined the rules, the next step is scheduling of the defined rules for execution. To illustrate how the scheduling of rules works, the *MoTifs* [15], a model transformation language, is used. Before having to discuss further on how the scheduling works, *MoTifs* uses the concept of *RuleBlocks* to encapsulate the rules. The reason for encapsulating rules in these *RuleBlocks* and the scheduling of them is mainly because a rule in itself does not specify whether CRUD operations should be applied on how many of the matches found in the model and in what order multiple rules shall execute. Figure 2.3, depicts the concrete syntax of the *MoTifs* language available constructs, and their description is discussed as follows.

- *ARule*: Atomic rule executes the rule for one match found. If no matches are found, it fails.
- *QRule*: A query rule succeeds if the LHS matches and the NACs do not match. The RHS of the rule is ignored.
- *FRule*: For-all rule executes the rule for each match in the match set. It fails if no matches can be found.
- *SRule*: Sequence rule executes the rule until no more matches can be found. It fails if no matches can be found.

- *BRule*: Allows for other rule-blocks to be nested and executes ,non-deterministically, one of the succeeding child rule-blocks.
- *CRule*: Allows nested transformation. The referenced transformation schedule is executed once.
- *BSRule*: Also allows for defining hierarchy by nesting other rule-blocks and executes (non-deterministically) one of the succeeding child rule-blocks until none of them succeeds.
- *PRule*: Allows for different ruleBlocks to be executed in parallel.
- *LRule*: Defines loops and its condition using an atomic rule-block. The body of the loop has a CRule rule-Block. As long as the atomic rule-block succeeds, a CRule will be executed.

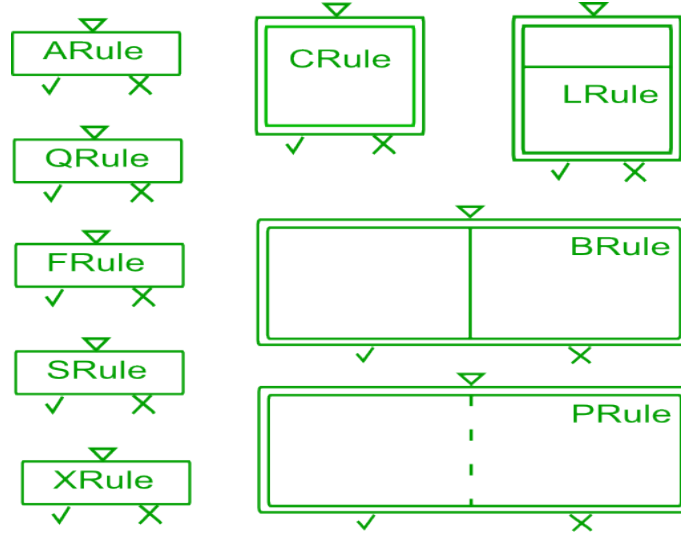


Figure 2.3: Graphical concrete syntax of *ruleBlocks* in MoTifs language

Each *ruleBlock* has a public interface with an input port, represented by a triangle, and two output ports, where ✓ represents success and ✗ for failure.

Similar to creating models in the Modelverse, this tool also enables the creation of model operations as models themselves, allowing several language instances to be executed. Thus, adding or modifying model operations does not alter the tool itself. There are some tools that use imperative languages, such as *metadept* [2] which uses EOL, others like *AToMPM* [16] use model transformations. The Modelverse has support for both, model transformations through RAMification and explicitly modelled action language. Model transformation through RAMification is often used with MPM [23]. RAMification is the process of modifying the meta-model of a language in such a way that it can be used in the patterns of transformation rules. With this approach, the model operations including source and target languages are all explicitly modelled.

2.3 The FTG+PM Language

The last aspect of MPM is process modelling. A Process Model (PM) is used in combination with the Formalism Transformation Graph (FTG), constituting the two sub-languages of the FTG+PM framework. With a wide range of formalisms and model operations utilized in the application of MPM, it can get quite confusing and difficult to execute model operations in a reproducible manner. Hence, the FTG+PM language provides a complete approach by offering an overview of all formalisms and their relations, i.e. the operations between them. In addition, it manages the various modelling artifacts created and the process itself. As this section gives a brief introduction to FTG+PM, further detail of this framework can be found in [9].

The FTG presents all used formalisms (DSMLs) and their relations (model operations) to be used in a process. It is represented by a hypergraph with languages as nodes and transformations as edges. It formulates the relationships among the various domain-specific languages and transformations used for the development of a particular system(s) in a certain domain. As depicted in figure 2.4, languages in the FTG are denoted by labelled rectangles, and transformation operations denoted by labelled circles on the edges. The incoming edges towards an operation show the source languages of the transformation while the outgoing edges from an operation point to the target languages. It can also be the case where the FTG model may include self directed loops, given the source and target languages remain the same.

A PM manages the used formalisms and their relations for a particular intent and can be used for documentation, enactment (i.e. automatic execution and chaining of operations), analysis and optimization. A PM generally consists of two types of nodes, namely operations and data. Operations are commonly model transformations, which is an automatic procedure, but can also be

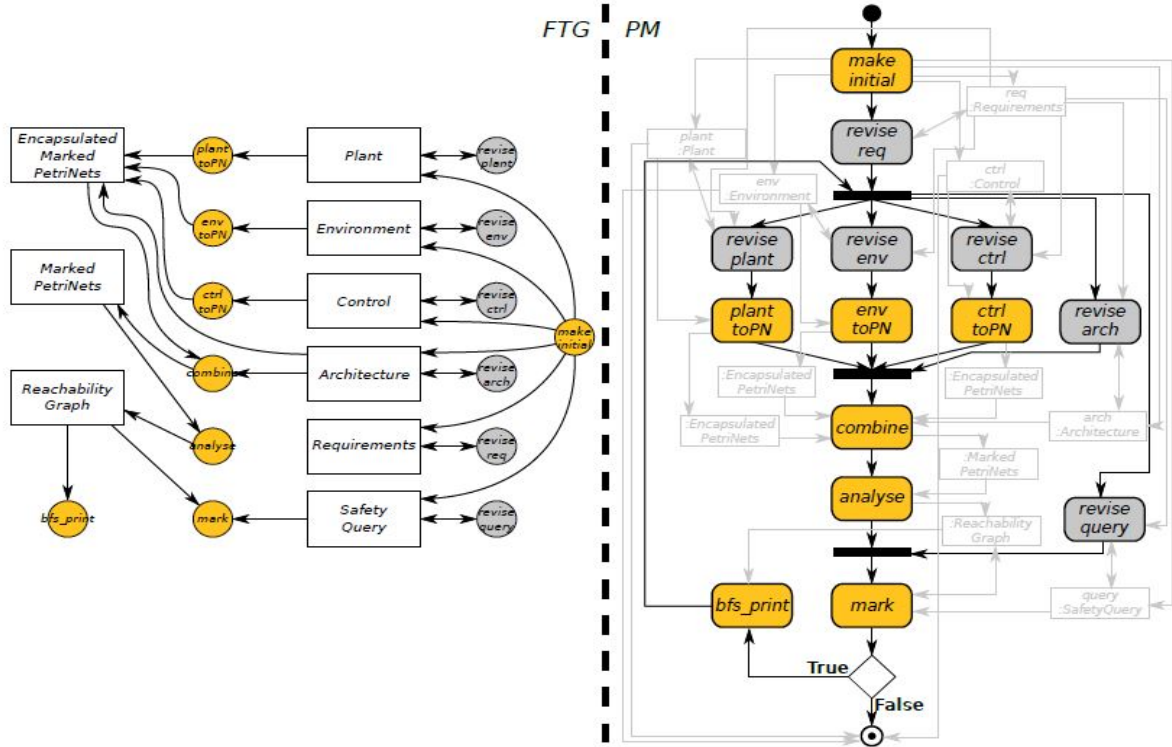


Figure 2.4: FTG+PM for the development and verification of a power window case study, adopted from [23]

manual operation where domain experts are required to handle. This can be seen on figure 2.4 where operations/activities are in grey colour, whereas in yellow are automatic model transformations. Data are the models these operations consume and produce. Thus, operation and data nodes form the two flows, i.e. control-flow and data-flow respectively. Control-flow begins from an origin point (the Start node) and is followed throughout until the process ends (the Finish node is reached). When control-flow encounters an operation, it executes it. During execution, an operation may consume data. After execution, the operation may also produce data and control-flow passes onto the next operation. Thus, data-flow specifies which model(s) are to be used by what operation and manages the modelling artifacts created during the process. While control-flow mandates which operations must execute in what order and is completely ignorant of what model(s) an operation acts on.

The PM in this thesis (taken as a subset of the UML 2.0 Activity Diagrams [14]) may include other operations which specify the structural (i.e. compositional hierarchy) and behavioural aspects of a process, such as a Fork, Join, and a Decision node. A Fork node represents concurrent execution of activities and influences control-flow in such a way that when control-flow encounters it, all subsequent operations automatically start execution. A Join node synchronizes activities executing in parallel. That means when control-flow encounters a join node, all previously executing activities have finished. A Decision node also influences the control-flow of a process by evaluating its input data with conditional data in its outgoing edges and decides in which of the outgoing edges control-flow proceeds. A subset of the UML 2.0 Activity Diagrams modelling constructs are shown in figure 2.6 and a portion of its meta-model expressed in Class-diagrams in figure 2.5.

Figure 2.4, shows an overview of a FTG+PM model for an automotive domain which describes all the modelling artifacts used, and the process for the intention of building a software system that controls the power window of an automobile. In the context of MPM, this case study is often used as it is simple to understand and addresses many of the challenges faced [3][9]. At the left hand side, the FTG is shown which presents the different DSMLs used, such as Control, Environment, Encapsulated Petri-net etc., and all operations among them, such as Control2EPN, Combine, Analyze. In the PM shown on the right hand side of figure 2.4, we can see that the PM refers to the languages and operations defined in the FTG and presents the order in which these operations can be defined. It also captures the specific modelling artifacts (DSMLs) that are propagated among different operations, shown in semi-transparent rectangular boxes. The details of the DSMLs and model operations used in this case study are further discussed in the literature [23][3][9].

The PM has a flow that begins with the input of requirements from different domains. Although different requirement can be used, we will focus on the verification aspect of the power window, i.e. we want to substantiate that the power window will never go up when an object is inserted through. In that regard, Control model is responsible for translating the key-presses (i.e. the Environment model which can be in Up, Down, or Neutral mode) into commands to the engine responsible for window

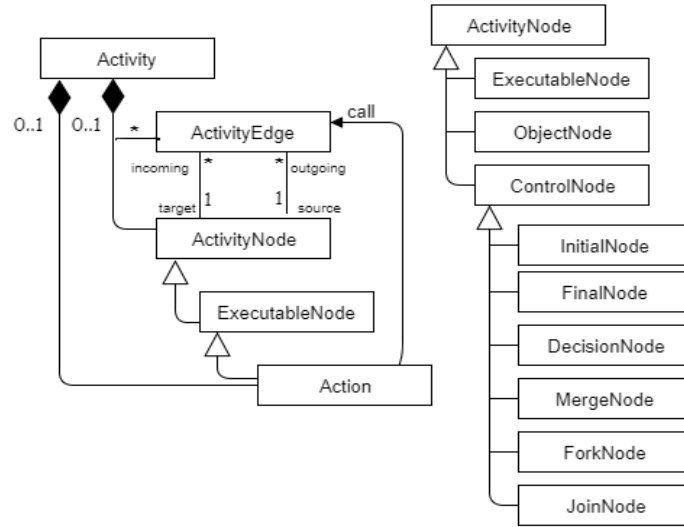


Figure 2.5: A small portion of the UML 2.0 Activity Diagram meta-model

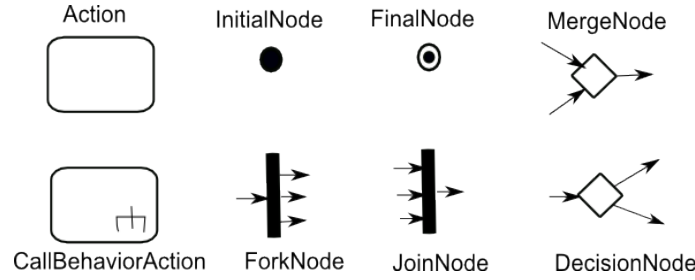


Figure 2.6: A subset of UML 2.0 Activity Diagram modelling constructs

movement. Similarly, domain experts create models for the Plant, Architecture and a safety query in a domain-specific language. The Control2EPN operation is, for instance, a model transformation which takes in, as data input, an instance of a Control model and produce an Encapsulated Petri-net (EPN). Similarly, Plan, Environment, Architecture are transformed to an EPN model and are combined by the Combine operation which yields a marked Petri-net. This Petri-net is then used to conduct a reachability analysis and, thus, produces a reachability graph. Note the use of a safety-query, also given as a model, which specifies a place p with a single token in it. Ultimately, if the query is found in the reachability graph by the Mark operation, it is deemed unsafe and control-flow goes back to a point where all domain experts revise their models, given as a manual operation. Otherwise, the system is deemed safe, and the process terminates.

The Modelverse has full support for the use of process models including their enactment, i.e. chaining of automatic operations. With the repository nature of the Modelverse, coupled with the processes themselves being models, it is possible for processes to be shared among different users with permission access. Sharing the FTG+PM, thereby, allows users for reproducibility by only enacting the same PM. Combined with the previously discussed aspects of MPM, the Modelverse offers a fully-fledged support of the FTG+PM framework, including the enactment of the FTG+PM in this power window case study. This case study is later used in this thesis to demonstrate the semantic of the PM given in SCCD, and it is further discussed in chapter 5.

2.4 Statecharts Class-diagram (SCCD)

Statecharts, first introduced by David Harel [6], is a visual modeling language that is a higraph-based extension of standard state-transition diagrams that is used to aid the specification of complex, reactive, timed, autonomous, interactive discrete-event systems. It is appropriate for describing *large* and *reactive* systems as it naturally adds the notion of depth, orthogonality and modularity, to 'normal' Finite State Automata (FSAs) [7]. However it lacks the facilities for specifying the structure of a system in addition to creating, deleting and communicating multiple Statecharts instances at run-time.

Class-diagram, in the Unified Modeling Language (UML) [11], is a visual modeling language that describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among classes. SCCD

[19] is a hybrid formalism which combines the structural object-oriented expressiveness of Class-diagrams with the behavioural discrete-event characteristics of Statecharts. This, in turn, facilitates the specification of dynamic-structure systems that are timed, autonomous and reactive. Classes 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 a Statecharts model, which describes the modal behaviour of the class, modelling its control-flow. At run-time, a class can be instantiated, which creates an object. Objects are initialized according to the classs constructor, and can be deleted, invoking the classs destructor. After initialization, an object is controlled by its Statecharts through changes in the object's state caused by triggers like events and operation invocation. The relationships modelled between classes are instantiated at run-time in the form of links. These links serve as communication channels, over which objects can send and receive events.

In this chapter the Statecharts and Class-diagram (SCCD) formalism is discussed. Many parts of this section are taken from where the formalism is first defined [19] and a documentation for the SCCDXML [18] notation. This notation is a concrete textual syntax representation, used as a serialization format for an existing SCCD compiler that generates code for multiple platforms. The various available constructs of Statecharts and their semantics are discussed first using a neutral visual representation, followed by a discussion of Class-diagram constructs supported by an example illustrating a combination of both Statecharts and Class-diagram in a single SCCD model. Finally, a brief introduction to how events can be used for communication among instances in SCCD models is discussed followed by a highlight of the object manager, which is in charge of the management of objects at run-time.

2.4.1 Constructs

In this section the semantics of the different constructs that make up the formalism are discussed. For readability purposes, the constructs are categorized as either Statecharts constructs or a concept that encapsulate the Statecharts into a class, and ultimately into a Class-diagram.

The constructs of Statecharts formalism discussed in this section are based on a neutral visual representation as proposed in Harel's definition published in 1987 [6]. Statecharts is a visual modelling language that extends finite state automata with added hierarchy, parallelism, history and broadcast communication. Harel created the formalism to be able to describe large and reactive systems, as there was no such method available at the time.

Basic State

The basic state acts as the main building block of a Statecharts and is represented by a rounded rectangle as in figure 2.7, where two basic states are depicted. A basic state, like that of composite and parallel states, represents a mode the system can be in. A state can be entered (which executes an optional block of executable content) and exited (which executes an optional block of executable content) using transitions. A Statecharts consisting solely of basic states has to have exactly one default/initial state, this is represented by an incoming edge with a black-dot as a source, as shown in figure 2.7.



Figure 2.7: Two basic states connected by a transition

Transition

In figure 2.7, the two states are connected by a transition originating from the initial state on the left, with a label of the form event[guard]/action. This means that, if the current state is A, upon reception of the trigger event e , the Statecharts will transition from state A to state B if and only if the guard condition c is satisfied. This guard condition can reference parameter values received by the transition which catches the event. Upon firing the transition, an action will be executed which in this case is raising an event r and an executable content a . Events in SCCD are strings that 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. Thus, transitions are triggered by an event or a timeout, or can be spontaneous. They can optionally specify a condition, an action and raising of an event.

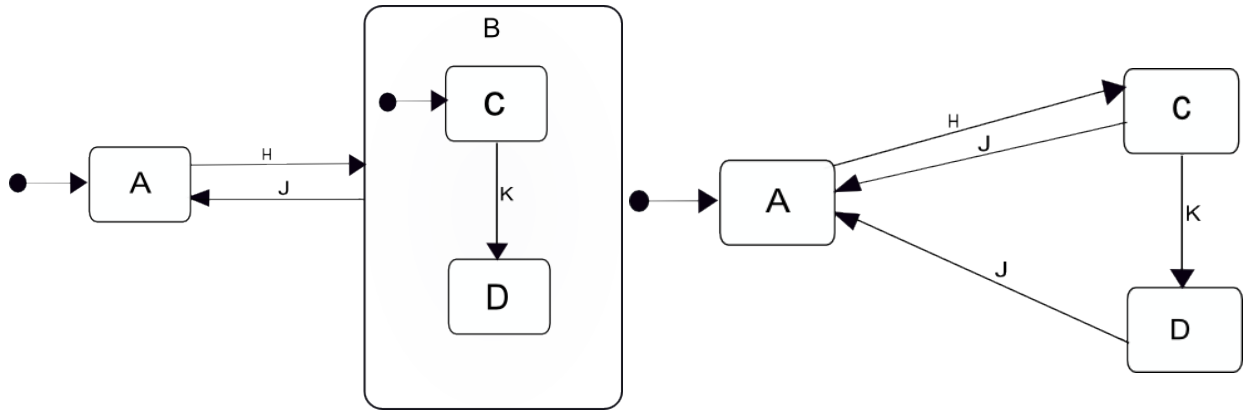


Figure 2.9: Semantics of Statecharts in (a) by "flattening".

Figure 2.8: Composite state B encapsulating two basic states C and D.

Figure 2.10: Depth/hierarchy in Statecharts.

Composite State

Composite states add a notion of hierarchy to Statecharts. The composite state is also called the XOR state because when such a state is active, exactly one of its substates must be active. Each composite state should have exactly one initial substate and transitions can occur at and between every level of the state hierarchy.

To illustrate this we look at the example in Figure 2.8. At initialization time, state A is active. Upon reception of the event *H*, the composite state B will be entered and consequently its initial substate C as well. At this point an event *K* can bring the Statecharts in the state configuration where B and its substate D are active, while an event *J* would bring the Statecharts back to its initial configuration where state A is active. Figure 2.9 shows the semantics of the Statecharts in figure 2.8 by "flattening" it. We can see that both substates C and D have an outgoing transition with event *J*. Thus, either one of the active substates, i.e. C or D can change the Statecharts configuration back to state A. Nonetheless, in case of non-determinism, i.e. if both the parent state and one of its substates have a transition leaving it on the same event, Statecharts will handle this by either enabling the transition associated with the inner substate or the transition associated to the parent state.

As mentioned before, with Statecharts it is possible to define actions and output events, that should be raised on either entering or exiting a specific state. When multiple layers of hierarchy are traversed on firing a transition, these actions are raised in an intuitive way. The exit actions are raised first, from the deepest level up to, but excluding, the first shared parent between the source and target states. This is then followed by executing the enter actions in the opposite direction down to the target states.

Parallel State

Besides the XOR composition achieved by a composite state, also AND composition is available in the Statecharts formalism. These are better known as parallel states or orthogonal components and allow for parallelism to be modelled. Upon entering a parallel state, each of the orthogonal regions (substates) will become active.

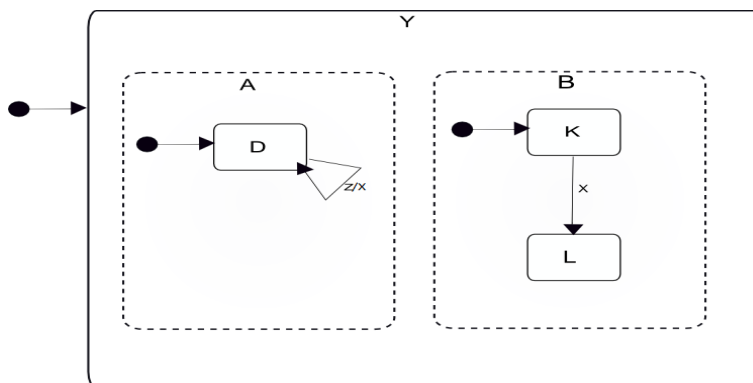


Figure 2.11: Parallel state Y with two orthogonal regions A and B

A parallel state is represented the same way as a composite state, however its substates are depicted by dashed rectangles expressing that they are active at the same time. We see such a state in Figure 2.11 labelled Y. Since this is the default state at the top level, this will directly be entered upon initialization. Consequently both substates A and B will be entered, which ultimately results in both inner states D and K being active at the same time. When the transition of K to L is triggered by the event X, state D will still remain active. In addition, an action appearing along a transition in Statecharts is not merely sent to the outside world as an output. Rather, it can affect the behavior of the Statecharts itself in orthogonal components and it is known as *broadcasting*. This can be achieved by a simple broadcast mechanism, as in the Statecharts shown in figure 2.11, where if an external event Z occurs, a transition labelled Z/X in orthogonal component A is taken, the action X of the transition is immediately activated and regarded as a new event, possibly causing further transitions in other components, in this case transition labelled X in orthogonal component B.

History State

A history state, which is depicted by a circle with the label H, adds memory to a component. A history state keeps track of the current configuration when its parent state is exited. If a transition has the history state as a target, the configuration that was saved is restored. If no configuration was saved yet, the default state is entered instead.

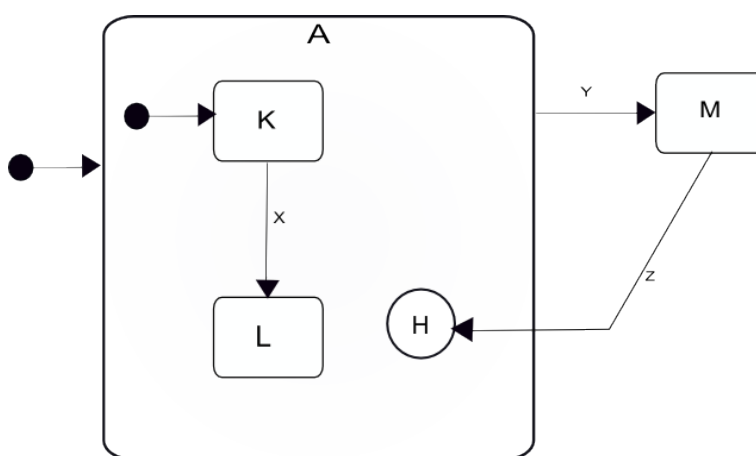


Figure 2.12: When the transition to the history state is triggered, the state of A will be restored to its last recorded state.

This can be illustrated further as in Figure 2.12 where there is a composite state A that has a history state and two sub-states of which K is the default one. If an event X is received after initialization, the composite state will reside in sub-state L (i.e., L is now active). Upon triggering the transition to state M, which is enabled by the event Y, the current sub-state of A is recorded first. When this is followed by an event Z, the transition to the history state is taken resulting in A being reactivated and thus having the saved state restored, where sub-state L is active again. Statecharts offers two types of history. The default shallow type only saves one layer of state in a component while the deep type saves all descendants of the component. The latter is represented by adding an 'asterisk' (*) to the state label resulting in H*.

Classes

The top level of a SCCD model resembles a UML Class-diagram with classes and edges connecting them. 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 a Statecharts model, which describes the modal behaviour of the class modelling its control-flow. At run-time, a class can be instantiated, which creates an object. Objects are initialized according to the classes constructor, and can be deleted, invoking the classes destructor. After initialization, an object is controlled by its Statecharts through changes in the objects state caused by triggers such as events and operation invocations. An SCCD model can also have an *external* class, denoted by a dotted rectangular box, which can be referenced from outside and thus can not be linked to a Statecharts for its behavioural specification. The relationships modelled between classes are instantiated at run-time in the form of links. They serve as communication channels, over which objects can send and receive events.

Class 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, which defines the minimal and maximal cardinality. 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. Inheritance edges have a priority attribute which allows to specify in which order classes need to be inherited (in case of multiple inheritance). Inheritance relations with higher priority are inherited from first.

In Figure 2.13 we can see the Class-diagram from the perspective of a single class, i.e. ClassD. This class is related with classes ClassE and ClassF by an inheritance edge and a named unidirectional association edge labelled 'association_g', respectively. Finally, a dashed edge with label <<behaviour>> is used to link the class to its corresponding Statecharts.

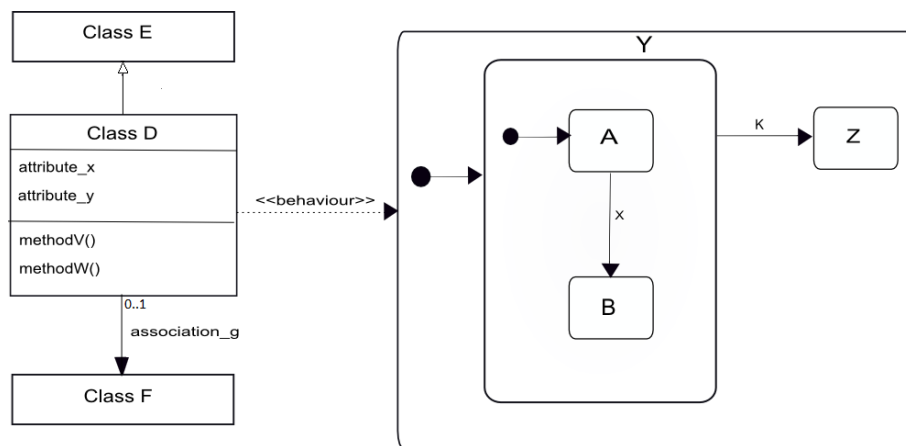


Figure 2.13: This figure illustrates the relation between classes in a Class-diagram and the Statecharts that describe their behaviour.

2.4.2 Events in SCCD

In the traditional Statecharts formalism, when casting an event, it was obvious that the scope of an event was local to the Statecharts that generates it. Now SCCD adds the ability to transmit events to class instances and to output ports with the addition of a public input/output interface using ports, as well as classes and associations. Thus, different levels of scope are added as described below by the different scope names used in the action associated with a transition that catches an event.

- *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 *port* 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 by referencing a link.
- *cd*: The event is processed by the object manager. See the next section for more details.

2.4.3 Object Manager

At run-time, 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 run-time, instead of as a SCCD class.

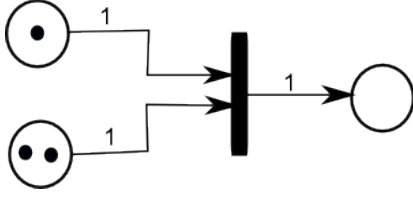


Figure 2.14: An enabled transition.

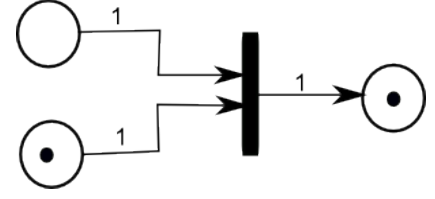


Figure 2.15: A disabled transition.

Figure 2.16: An example of a Petri-net model before (a) and after (b) firing the transition.

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 such as *associate_instance* and *delete_instance* to the object manager to control the set of currently executing objects.

2.5 Petri-nets

Petri-nets are mathematical models used for describing discrete event dynamic systems [1]. A classical Petri-net is a directed bipartite graph with two node types, namely *Places* and *Transitions*. The nodes are connected via directed arcs, of which connection between two nodes of the same type is not allowed. Places are represented by circles and transitions by rectangles, as in figure 2.14 which depicts a simple Petri-net model.

Definition 2.5.1. A Petri-net is a 5-tuple (P, T, A, ω, M_0) [10], where

- $P = \{p_1, p_2, \dots, p_n\}$ is set of finite Places
- $T = \{t_1, t_2, \dots, t_n\}$ is set of finite Transitions
- $A \subseteq (P \times T) \cup (T \times P)$ set of arcs (flow relation)
- $\omega : A \rightarrow \mathbb{N}$ a weight function
- $M_0 : P \rightarrow \mathbb{N}$ an initial marking

At any time a place in a Petri-net model can contain zero or more tokens, represented by small black dots. The state (marking) of the Petri-net is the distribution of tokens over its places. Given a place (as source) connected to a transition (as target) using an arc, the place is referred to as an *input* place of the transition. Likewise, when a transition (as source) is connected to a place (as target) with an arc, this place is called an *output* place of the transition. A transition without input place(s) is called a source transition whereas a sink transition has no output place(s).

The semantic mapping of a Petri-net model can be simulated in its reachability or coverability graph by using the standard *firing rule*. A transition t is said to be *enabled* with respect to some marking M if and only if each input place p of t contains equal to or greater than the *weights* placed on the arcs connecting p to t . Note that a source transition is always enabled. An enabled transition t may fire, and if it fires, t *consumes* as many token(s) as the *weights* placed on each arc from p to t . In addition, transition t also *produces* for each p , as many token(s) as the *weights* placed on each arcs from t to output place p . In the example Petri-net model on figure 2.14, we can see two input places for the transition and that it is enabled. After firing, as depicted on figure 2.15, it no longer becomes enabled given that there is not at-least one token in place $p1$.

Petri-nets are very intuitive due to their simple graphical concrete syntax. They are commonly used for analysis of complex systems with shared resources such as distributed systems, parallel programs, control systems, shared memory and networks [1].

2.5.1 Workflow-nets and Soundness Properties

Workflow nets (WF-nets) are a particular class of Petri-nets, used to model and analyze workflow procedures. A workflow procedure specifies the set of partially ordered tasks required to process workflow cases successfully. Petri-nets which model workflow procedures have two typical properties. First, they always have two special places i and o , which correspond to the beginning and termination of a workflow procedure respectively. Place i is a *source* place and o is a *sink* place. Second, for each transition t and place p , there should be a directed path from place i to o via t and p . A Petri net which satisfies these two requirements is called a Workflow net [17]. Figure 2.17 depicts a workflow-net with a *source* and a *sink* place. The "cloud" represents a process which can be initiated by putting token(s) on the input place *source* and the goal is that the process successfully completes by putting a token in place *sink*.

Having (PN, M) to denote a Petri-net PN with some initial state marking M , let us see some properties of a Petri-net that are used to derive the soundness properties of a workflow-net.



Figure 2.17: A WF-net is a Petri net with a source and a sink place. The goal is that a process initiated via place *source* successfully completes by putting a token in place *sink* [17].

Definition 2.5.2. (Live) A Petri net (PN, M) is *live* if for every reachable state M' and every transition t , there is a state M'' reachable from M' which enables t .

Definition 2.5.3. (Bounded) A Petri net (PN, M) is *bounded* if for every reachable state and every place p , the amount of token in p is bounded.

A workflow-net such as the one depicted in Figure 2.17, is sound if and only if the following three requirements are satisfied: [17]

- *Option to complete*- For every state M reachable from the initial state i , there exists a firing sequence leading from state M to a state which just marks the place *sink*.
- *Proper completion*- When the state which marks the place *sink* is reached, the state configuration of all other places should be empty.
- *No dead transitions* - it should be possible to execute an arbitrary activity by following the control-flow of a workflow-net.

Hence, the soundness properties stated above can be translated to the liveness and boundedness problem, i.e. a Workflow-net is sound if and only if it is live and bounded.

3

Mapping PM onto SCCD

As the underlying aspect of this thesis research is to define the semantics of PM in SCCD, this chapter gives an elaborated discussion on the approaches explored and an implementation, in the Modelverse, of denotational mapping PM onto SCCD.

3.1 Mapping Approaches

Apparent to both languages (PM and SCCD) is the use of common design patterns such as hierarchical composition, sequential-execution, concurrency and synchronization, multiple instantiation (of PM activities or SCCD class instances) through sequential iteration, and the support for data exchange among the modelling constructs. A PM has two nodes, i.e. activity and data, and an edge associating activities. Activities can be of an executable type or a control node such as Fork and Join which influence the control-flow of a process. An executable activity is a model operation or an external service which consumes and/or produces data as models. The behaviour of such an activity is to be expressed in an explicitly modelled SCCD.

While encapsulating SCCD in PM executable activity, the executable corresponds to a single SCCD instance, as depicted on

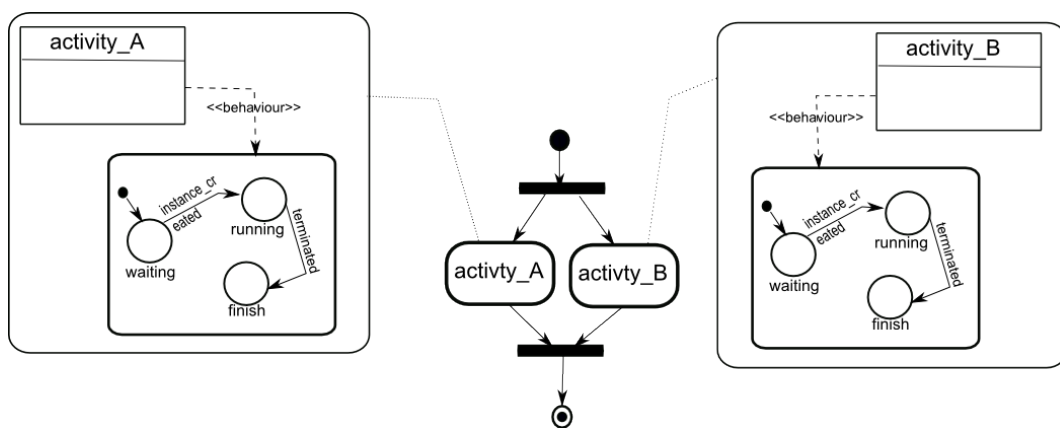


Figure 3.1: Workflow modelling at different levels of abstraction

figure 3.1. This figure shows a PM with multiple levels of abstraction where the behaviour of PM activities is expressed in a low-level language (SCCD). Hence, model operations and/or external services can be explicitly modelled in the Statecharts belonging to the Class-diagram instance. Nonetheless, the PM is still the one in charge of managing the creation and execution order of an executable (expressed in SCCD instance), in addition to controlling the data artifacts produced and consumed. The PM, thus, needs to be mapped onto a SCCD instance(s), so that ultimately the PM can be realized as one SCCD model, which can then be executed.

From here on, we will refer to the mapped SCCD instance as the *Orchestrator* and it entails the two main responsibilities of PM, i.e. managing the *control-flow* and *data-flow* of a process. Managing the control-flow means that the Orchestrator (as the

default SCCD) is created first and bears the responsibility of controlling the order of spawning SCCD instances according to the semantics of PM constructs influencing its control-flow. In addition, the Orchestrator is in charge of organizing the data artifacts in such a way that, while simulated, it should be well aware of which data is consumed and/or produced by which SCCD instance that is related to an executable.

There are two approaches explored in this research that would make the mapping of PM onto SCCD Orchestrator possible. For simplicity, we can leave the data-flow aspect to be discussed in the subsequent section. The two approaches with their related benefits and limitations are discussed next.

3.1.1 Sequential Mapping

In sequential mapping, PM constructs are mapped onto SCCD constructs in a more intuitive way by fully exploiting the common design patterns inherent to both languages. This can be illustrated in the following example. Figure 3.2, depicts a simple PM instance consisting of a subset of the UML 2.0 ADs constructs. A denotational mapping of the PM in figure 3.2 is shown in the corresponding Orchestrator Statecharts in figure 3.3.

At initialization time of the Orchestrator SCCD instance, the initial state is entered, i.e. the parallel state with orthogonal

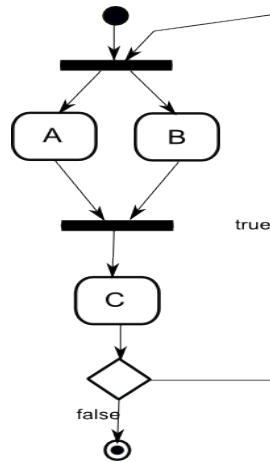


Figure 3.2: A simple PM instance expressed using a subset of UML 2.0 ADs constructs

components. This is related to the PM where at the start, control-flow encounters a Fork node, denoting concurrent execution of activities A and B. Similarly, when the parallel state is entered in the Orchestrator Statecharts, it will have all its orthogonal components active. The behaviour of SCCD instances related to an executable activity can be described with three basic states as children of a hierarchical state, such as the orthogonal component 'activity_A' or the composite state 'activity_C' in figure 3.3. When each orthogonal component in the parallel state is entered, the initial states become active, and raise a 'create_instance' event to the object manager of SCCD, responsible for creating the corresponding SCCD instance. Upon receiving 'instance_created' event back from the object manager (by the event attached to the transition originating from the initial state), the orthogonal component transitions to a running state and waits until it receives a 'terminated' event from its corresponding SCCD instance. When that happens, the orthogonal component transitions to a finish state, where it raises a local event 'finished'.

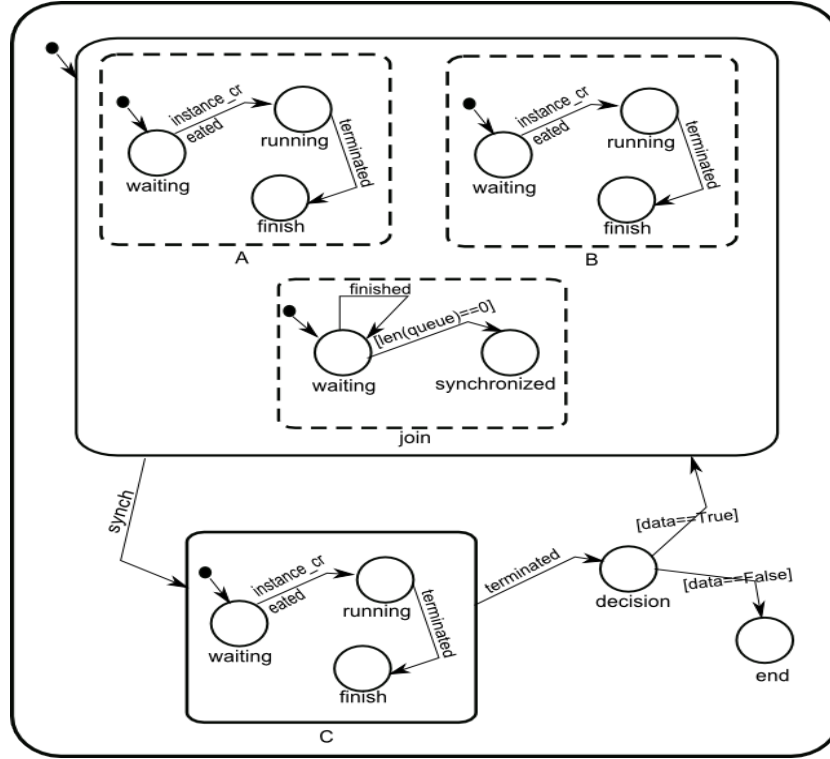


Figure 3.3: Sequential mapping of PM in figure 3.2 onto the Orchestrator SCCD

The third orthogonal component is a mapping of the Join construct in the PM. This orthogonal component's initial waiting state also becomes active when the parallel state is entered. Attached to this state is a transition that listens to the 'finished' event raised by the rest of the orthogonal components. Every time this transition triggers, it takes an action which removes the element who raised the 'finished' event from its collection. While mapping, this collection can be initialized with names of activity nodes the Join synchronizes from the PM. The other transition, targeting the synchronized state, will be triggered when the guard condition becomes valid, i.e. the collection becomes empty. At this point, all orthogonal components have reached their 'finish' state, thereby synchronized. The synchronized state of the Join orthogonal component is also entered and raises a 'synch' event which triggers the transition exiting the parallel state. This denotes the synchronization of activities A and B by a Join node in the corresponding PM in figure 3.2.

In a similar manner, the state transitioning (flow of control) continues until the 'decision' state, mapped to Decision node in the PM, is reached. This time, the conditions placed upon the state will be evaluated to direct the flow of control. The evaluation compares the data consumed by the Decision at run-time, with data values placed on the outgoing edges of the Decision node in the corresponding PM. This data from the PM can be supplied to the mapped 'decision' state in the Orchestrator during the mapping (model transformation) process. Hence, in this model, if the evaluation results to be False, the process will reach an 'end' state. Otherwise, it will transition back to the parallel state, where all orthogonal components enter their initial states. This, thereby, enables for sequential multiple instantiation of the SCCD instances.

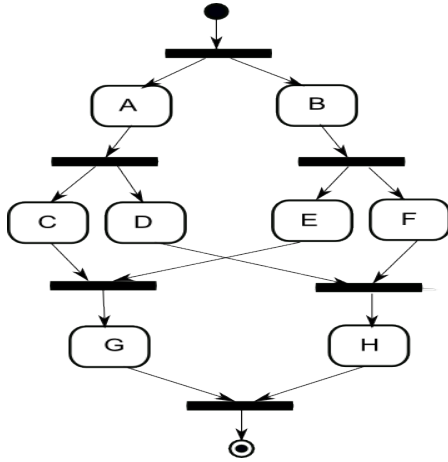


Figure 3.4: Interleaving branches in parallel regions

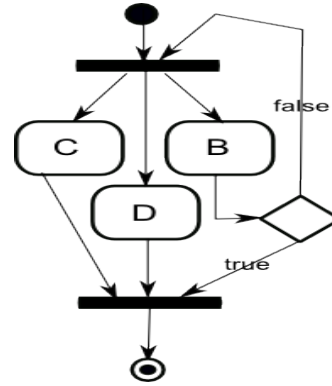


Figure 3.5: A branch exiting parallel region without synchronizing

Figure 3.6: Process model instances with complex behaviour

Despite the intuitive way of mapping PM onto SCCD, which makes it also easy to debug, sequential mapping can have a major setback. Process models can possibly get too complex in the definition of their control-flow, especially when it involves the interleaving of Fork branches in parallel regions, such as the one in figure 3.4. It is almost impossible to manage the mapping of such PM onto SCCD in such a sequential manner. In SCCD, a transition from a child state of an orthogonal component cannot exit the parallel region, as this breaks encapsulation and can interfere with the behavior of other orthogonal components of the parallel state. Transitions only from the parallel state can exit the parallel region, which will automatically exit all its children as well.

Another example of a PM where sequential mapping can be impossible is in the case where a branch originating from a Fork node exits the concurrent region before being synchronized. This is admissible, though with little practical value, as the meta-model of UML AD does not place any such constraints [14]. An example of a PM depicting such a behaviour is shown on figure 3.5. Both process models in figure 3.4 and 3.5 can be verified as conforming to the abstract syntax definition of PM but yet impossible or too complex to map to SCCD sequentially. Hence a more generic approach is needed as discussed in the following section.

3.1.2 Generic Mapping

As the name implies, this generic way of mapping can apply to any given PM which conforms to its meta-model. To manage control-flow, instead of sequentially mapping PM constructs to their SCCD constructs with common design pattern, such as Fork to Parallel state, this approach maps each PM activity type to an orthogonal component inside a single parallel state. These orthogonal components remain active as of the point the parallel state is entered. The orthogonal components have states which describe the behaviour of PM constructs being mapped to, and can make use of transitions to emulate the control-flow and data-flow of a process. Thereby, this approach employs a *token* based routing of control-flow and fully leverages *concurrency* in SCCD.

The underlying idea behind this approach is that an activity, of any type except a Join construct, becomes active when it acquires the shared *token*. A Join node becomes active when all previous nodes signal that they have *finished*. This approach can be summarized in the Orchestrator SCCD depicted on figure 3.7. The Orchestrator has, in its parallel state, orthogonal components which are mapped to each activity types in PM. In that sense, the orthogonal component models the behaviour of the corresponding SCCD instance that it is mapped to.

The default SCCD instance (i.e. the Orchestrator) is instantiated first and it initializes the state of each node in the flow of control. For instance, a Start node has a valid token entry during initialization. The token entry for all nodes of PM actually emulates the flow of control in the process. That is to say if a state in an orthogonal component mapped to a given node "consumes" the token, it signals control-flow is at the current node. This will imply that when the process terminates, only the Finish Orthogonal component in figure 3.7 will have a valid token. Similarly, all nodes with a control-flow edge targeting a Join node have a state of being "finished" or not. This state can be verified by a state in the Join orthogonal component to signal for synchronization of a parallel region.

To demonstrate how this approach works in SCCD we will have a look at each mapping of PM node to an orthogonal component in the parallel state of the Orchestrate Statecharts. After initialization completes, the parallel state is entered which has a corresponding orthogonal component for each node in the PM, as shown in figure 3.7. An *Executable* orthogonal component has two states in which it can be at, reflecting the behaviour of the SCCD instance it relates to. This can be seen in figure 3.8 which is a closer look at the Executable orthogonal component of the Orchestrator parallel state in figure 3.7. It has an initial

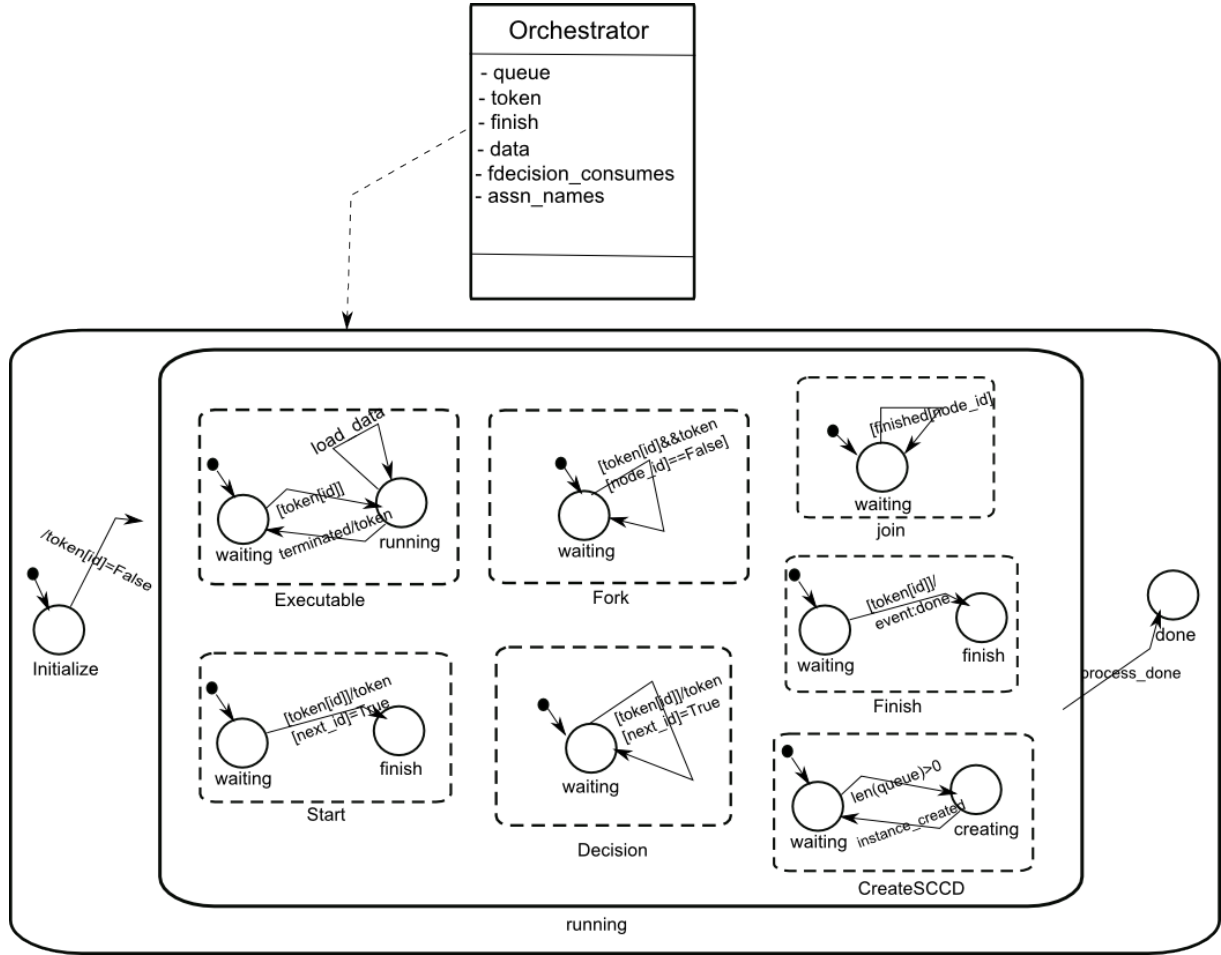


Figure 3.7: The Orchestrator SCCD using generic mapping

'waiting' state with a transition targeting a 'running' state. This transition is triggered when its guard condition becomes valid, i.e the executable "consumes" the token. At this time, the transition takes an action which places the name of the executable in a queue, where the Orchestrator uses to spawn SCCD instances. This queue is used in the CreateSCCD orthogonal component in figure 3.7, which has a waiting and creating state. The transition attached to the waiting state gets triggered so long as the queue is not empty. Upon triggering, it raises an event to the object manager of SCCD to create the SCCD instance with the specified executable name. Note that the SCCD instance has an identical name to its corresponding executable activity.

After the SCCD instance is created, the executable orthogonal component transitions to a 'running' state. This state has two transitions. The transition with an event 'load_data' is used to provide data upon the request of the SCCD instance and this is discussed in section 3.2. The second transition listens to the event 'terminated', which is raised from the running SCCD instance. When triggered, this transition takes an action to "release" the token from the executable activity and passes it to the successor node. However, if the successor node is of type Join, in addition to "releasing" the token, it signals that the executable has "finished". At this time, the state also transitions back to the 'waiting' state. This is because an activity can always have the possibility to instantiate itself again if the PM allows it. At any point in time during execution, control-flow can reverse back to a point already executed, which enables for multiple instantiation of activities sequentially. This can be seen in the example PM depicted on figure 3.2. SCCD Classes enable to model such a dynamic-structure behaviour where multiple instances of the same type can be created, both sequentially and simultaneously.

A *Join* orthogonal component has a single state with a transition targeting itself. This can be seen in figure 3.8, which is a closer look at the Join orthogonal component of the Orchestrator parallel state in figure 3.7. This transition hold a guard condition, when satisfied, implies the synchronization of parallel branches of a Fork node in the PM. Thus, this guard condition is defined by all nodes currently "consuming" the token (i.e. source elements attached to the incoming edges of a Join construct in the corresponding PM) signal that they have "finished". When the transition triggers, the Join orthogonal component relays the token to its successor and remains in the same 'waiting' state. This enables the flow of control to activate it again if the PM allows it. When a *Fork* orthogonal component is entered, it will be in the initial 'waiting' state, which has a transition targeting itself. This

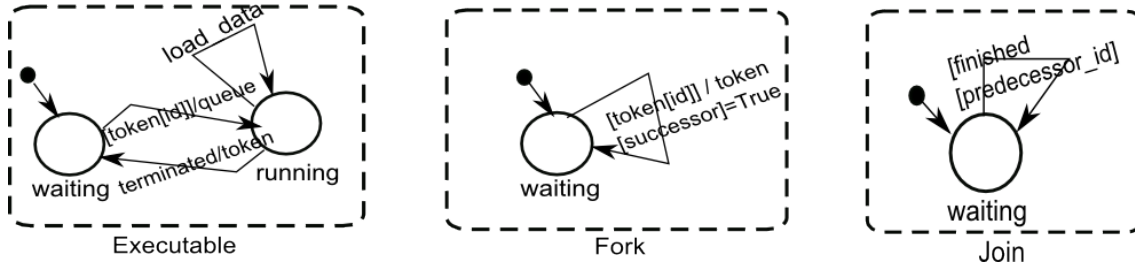


Figure 3.8: Orthogonal components of the Orchestrator parallel state in figure 3.7, mapping an Executable, Fork and a Join node

can be seen in figure 3.8 which is a closer look at the Fork orthogonal component of the Orchestrator parallel state in figure 3.7. This transition has a guard condition that becomes valid if the Fork "consumes" the token. After triggering, the transition takes an action which "releases" the token from the Fork node and distributes the token to its successor nodes. Nonetheless, there can be a PM instance where a Fork branch exits its parallel region before synchronizing, as shown previously in figure 3.5. It is worth mentioning that this kind of control-flow can lead to simultaneous multiple instantiation of an executable activity, which does not make sense and even more essentially, it makes the process model *unsound*. Here we can apply the concept of workflow-nets to verify the soundness property of workflow models discussed in chapter 2. A Workflow-net is sound if and only if it is live and bounded. Thus, the process model in figure 3.5 can become unbounded when the decision node non-deterministically chooses to exit the parallel region before synchronizing. This can lead to process execution anomalies such as *improper termination* of a process.

A *Decision* orthogonal component has a single state with a transition targeting itself, as in figure 3.9 which is a closer look at the Decision orthogonal component of the Orchestrator parallel state in figure 3.7. This transition is triggered when its guard condition becomes valid, i.e the Decision node "consumes" the token. At this time, the transition takes an action which "releases" the token from the Decision node and *decides* who to pass the token to. This decision requires an evaluation that takes as input the run-time value of data consumed by the Decision node and compares this input with data values placed on its outgoing edges. Ultimately the evaluation will find one of the conditions to be valid, which dictates the flow of control towards the valid edge by "releasing" the token to its associated node.

A *Start* orthogonal component in figure 3.9, has a 'waiting' state that it enters to when the parallel state becomes active. This

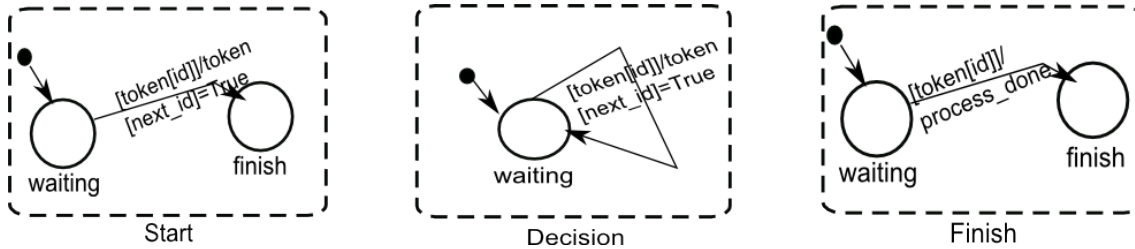


Figure 3.9: Orthogonal components of the Orchestrator parallel state in figure 3.7, mapping a Start, Decision and a Finish nodes

state has a transition targeting a 'finish' state and holds a guard condition which specifies that the node "consumes" the token. This takes place during initialization time of the Orchestrator at its initial state. After triggering, the transition takes an action which "releases" the token from the start node and passes it to the successor node. At this time a state transition occurs to the 'finish' state and remain in that state since control-flow does not encounter it again. Lastly, a *Finish* orthogonal component in 3.9 also has a 'waiting' state with a transition targeting a 'finish' state. This transition holds a guard condition which gets triggered when the finish node "consumes" the token. Once triggered, this transition raises an event 'process_done', which also triggers the transition exiting the Parallel state. The process will thus leave the parallel state and transitions to the 'End' state, where the process completes.

This way we can claim that the generic approach completes the major setback of sequential mapping. A join orthogonal component can synchronize branches of any parallel region without having to be associated with a particular parallel region as in the case for sequential mapping. It is also possible to map a PM with control-flow having a branch exiting a parallel region before synchronizing. Nonetheless, generic mapping does not provide easy debugging for constructs being mapped. However, we can use traceability links in the implementation, discussed after the following section.

3.2 Data Management

Data artifacts are propagated throughout the PM while enacting executable activities. These data artifacts represent model instances of Domain Specific Modelling Languages (DSMLs). The data-flow of a PM specifies which model(s) are used by which operations, while control-flow dictates which operation to execute and is clueless of which data an operation acts on. Hence, during the mapping of PM onto SCCD, the Orchestrator SCCD should provide a way to manage the data-flow, and decide which data is passed on to which executable activity (SCCD instance).

Mapping data-flow of a PM onto SCCD Orchestrator is relatively straightforward and consists of two steps. The first step is that during the transformation of PM to SCCD, the transformed Orchestrator SCCD keeps a record of which model operation, represented by an executable activity, produces and/or consumes what data, if any. These data artifacts are represented by a meta-data of models provided at design time of the PM instance, as shown in figure 3.10. Thus, each data has a 'tag' label which identifies the data a model operation consumes and/or produces (as an operation can consume/produce multiple data), a 'name' of the model instance, and a 'type' (metamodel) name. This meta-data can be provided to the Orchestrator SCCD while implementing model transformation of PM to SCCD.

The next step is to provide a communication protocol where the Orchestrator and the other SCCD instances exchange data.

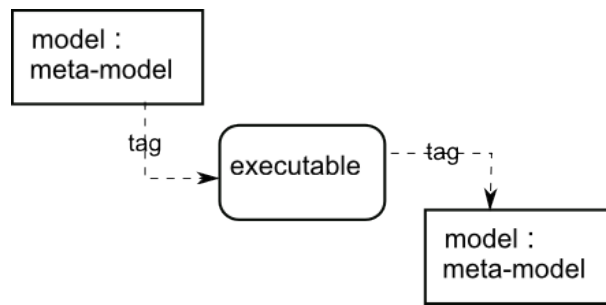


Figure 3.10: Data-flow representation at design-time using UML 2.0 Activity Diagram constructs

As discussed in the previous section, the transformed Orchestrator SCCD has an orthogonal component which is mapped to an executable activity of PM. In this orthogonal component there are two states, which depict the behaviour of an SCCD instance corresponding to the executable activity. When in the 'running' state, there is an attached transition which listens to the event 'load_data', as in the executable orthogonal component on figure 3.8. Since this state is enclosed by an orthogonal component inside a parallel state, and that these orthogonal components can be as many for all executable types of a given PM instance, the transition can listen to data requests which does not belong to the executable in which the orthogonal component is mapped to. Hence, this transition has a guard condition which evaluates who requested the data based on a 'name' parameter. The 'load_data' event is raised by the running SCCD instance and sends its name as a parameter. The Orchestrator SCCD then sends the appropriate data from its record collection, by raising an event 'set_data' to the requesting SCCD instance. Once receiving this data, the SCCD instance can access the actual model(s), operate on them, and store model(s) created during model operations, by using interfaces provided by the Modelverse tool.

3.3 Implementation

A model operation is used to define the semantics of Process Model (PM) denotationally by mapping it onto SCCD. Thus, the first step towards implementing this is to create the modelling languages themselves. Once these languages are created using their abstract and concrete syntax, it is possible to create an instance model of the languages. The implementation environment used is the Modelverse, a prototype tool for multi-paradigm modelling and simulation [21]. The Modelverse only has an interpreter for neutral action language code which this implementation uses. The semantics of this language is explicitly modelled, which makes it possible to automatically generate interpreters for any desired platform.

3.3.1 Abstract and Concrete Syntax of PM and SCCD

In this implementation, the abstract syntax, represented by a meta-language Class-diagram, is defined for SCCD, as depicted on figure 3.13. The SCCD meta-model is developed in reference to its documentation in [18]. The abstract syntax for the PM language (as a subset of UML 2.0 AD) is presented in the previous chapter. This implementation also refers to it and can be seen in figure 3.11. Note that edge associations, in both meta-models, can have inheritance relation and this is supported by the Modelverse, as associations can behave just like classes. They can also have attributes and associations of their own. An edge of such kind is represented on both meta-models as a Class type and this is shown in figure 3.11 and 3.13 by linking the Class

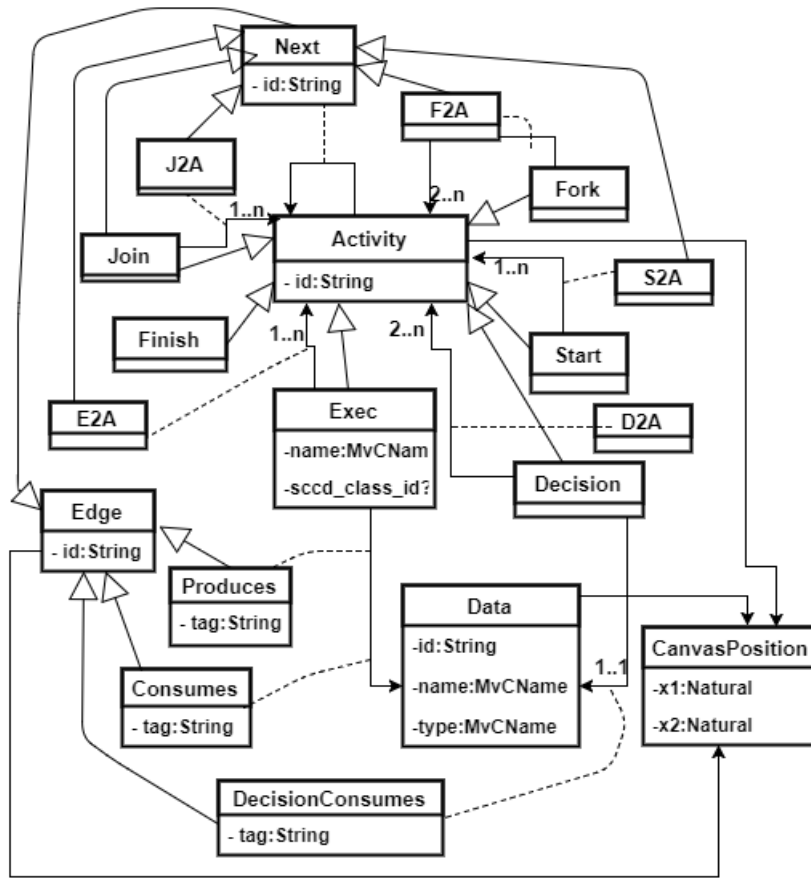


Figure 3.11: Process Model abstract syntax expressed in Class-diagram meta-language

with the actual association using a dashed-line. Associations have a zero-to-many cardinality except the ones explicitly labelled on both meta-model figures.

In the developed meta-model of PM language, there are few changes introduced, without changing the semantics of the lan-

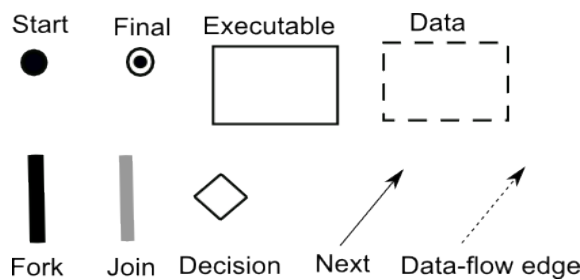


Figure 3.12: Process Model graphical concrete syntax adopted by the developed editor

guage. One of the slight changes is that higher constraints have been placed on the language constructs, as the intent is to execute instance models and requires conformance verification. For example, it does not make sense to express both control-flow and data-flow using a generic 'Activity' node and 'Activity' edge as it is the case in the meta-model of ADs shown in figure 2.5 of the previous chapter. All sub-classes, such as 'Fork', inherit from this association and yet they are not associated with data-flow in their semantics. Moreover, although control-flow is represented by the 'Next' association class, there are specialized association types introduced to place constraints on. For example, a 'Fork' node has a target lower cardinality of two, where an instance model can be verified for conformance. These specialized association types are used by the developed front-end, enabling to create only allowed edge types, as discussed in section 4. On the other hand, the generic types, such as 'Next' and 'Activity', are used during model transformation for matching any control-flow edge and activity type, respectively.

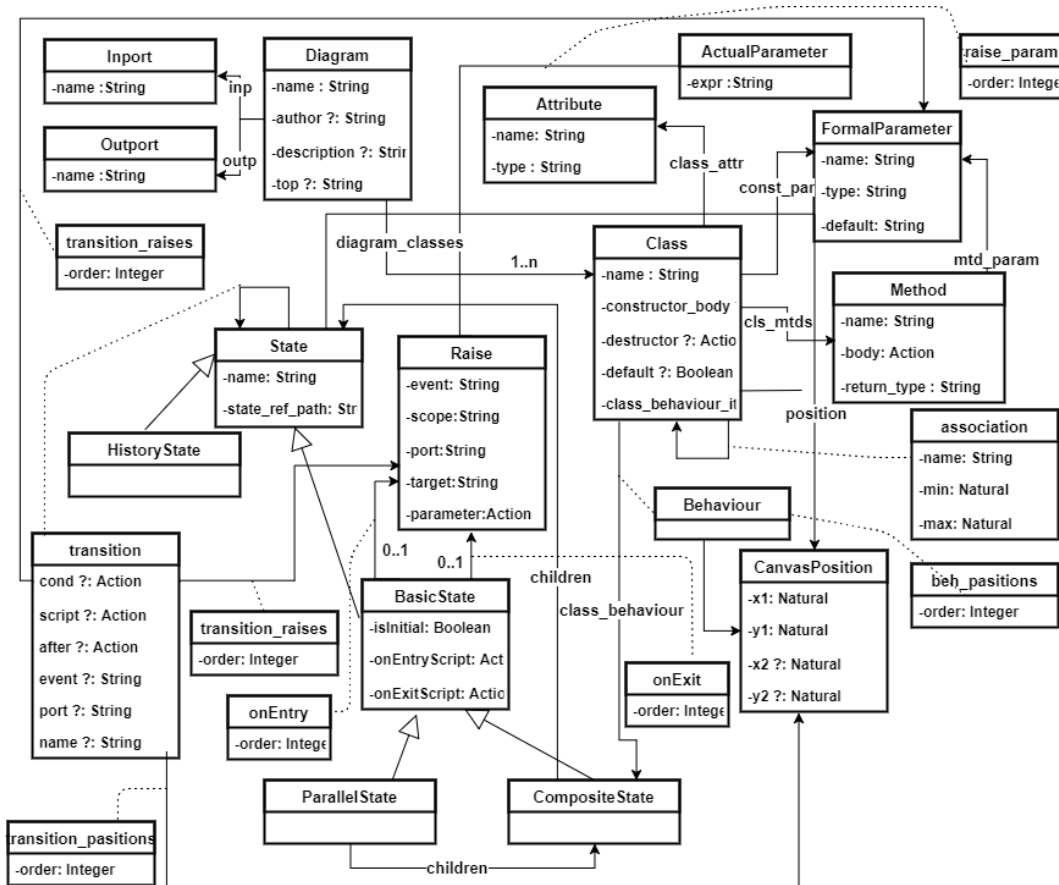


Figure 3.13: Statecharts abstract syntax expressed in Class-diagram meta-language.

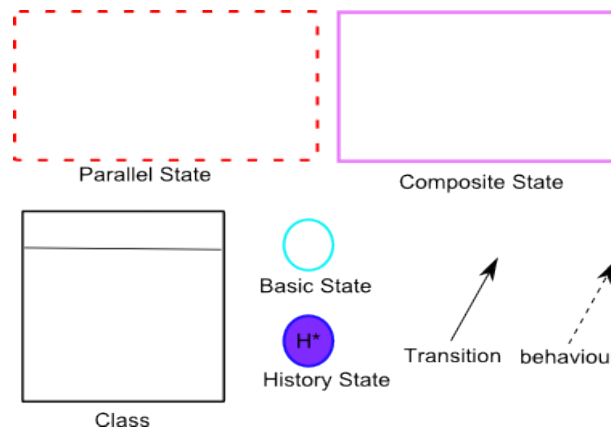


Figure 3.14: SCCD graphical concrete syntax adopted by the developed editor

Some of the introduced changes are also used to adopt to the developed Graphical User Interface (GUI) editor, as it is also the case for SCCD meta-model. These updates include additional entities, such as 'CanvasPosition', which saves the position of element on the editor's canvas and can be used for rendering models from the repository back to the editor. Note that the Modelverse is also a model repository and uses the abstract syntax of a language as data structure to save models. The graphical concrete syntax of Statecharts in SCCD, as depicted on figure 3.14, looks a bit different than Harel's first representation, shown in chapter 2. These changes took place as an alternative choice, and include highlighting of initial states in colour rather than

a small uni-directional edge targeting the state. In addition, hierarchical states are not rounded rectangles. A 'Parallel' state is represented by a dashed-line rectangle, with 'Composite' states as its orthogonal components. The developed concrete syntax of the PM language also adopts to UML 2.0 ADs with a slight change that enables a clear distinction between 'Join' and 'Fork' nodes as well as a different representation for data nodes, as shown in figure 3.12.

3.3.2 Model Transformation of PM to SCCD

The Modelverse enables the creation of model operations as models themselves, allowing several language instances to be executed. In this case, the model operation takes, as input, an instance of PM and SCCD, and produce one SCCD model instance that includes the Orchestrator SCCD. As discussed in the previous chapter, the Modelverse has support for both types of commonly used model operation approaches, model transformations through RAMification and explicitly modelled action language. While mapping PM constructs onto SCCD with the chosen approach, there is a tedious task of identifying order, i.e. the preceding and/or successor element of PM activities and data nodes being mapped. This implementation, thereby, uses model transformations through RAMification, as it can take advantage of the very intuitive (i.e. explicitly modelled) pattern searching method. In addition, note from the abstract syntax meta-model of the PM language in figure 3.11, an 'Activity' is a generic entity as all nodes, except 'Data', inherit from it. Similarly, the association 'Next' is a generic edge denoting control-flow edges of a PM. Hence, the implemented model transformation uses these generic types to search for pattern matches, which reduces the number of rules required for mapping the constructs.

The mapping of PM instance onto SCCD model consists of multiple transformation rules and enclosing of these rules in a chosen rule-block. The rule-blocks are then scheduled for execution, as depicted in the code fragment shown in figure 3.1. The rules and their scheduling are implemented using a textual concrete syntax, which the Modelverse only supports at this time. However, graphical illustrations of some of the rules discussed are provided here for readability. In addition, all executable code fragments are written in neutral action language code, which the Modelverse has an interpreter for.

```
Initial (schedule, create_orchestrator) {}

OnSuccess (create_orchestrator, create_association) {}
OnSuccess (create_association, ModifyFork2Fork) {}
OnSuccess (ModifyFork2Fork, ModifyForkToFork) {}
OnSuccess (RemoveForkToFork, RemoveJoinToJoin) {}
OnSuccess (RemoveJoinToJoin, initialize) {}
OnSuccess (initialize, data_produces) {}
OnSuccess (map_exec, map_decision) {}
OnSuccess (data_produces, data_consumes) {}
OnSuccess (data_consumes, compile_init_script) {}
OnSuccess (map_decision, map_fork) {}
OnSuccess (map_fork, map_join) {}
OnSuccess (map_join, map_finish) {}
OnSuccess (map_finish, map_start) {}
OnSuccess (map_start, success) {}

OnFailure (create_orchestrator, create_association) {}
OnFailure (create_association, ModifyFork2Fork) {}
OnFailure (ModifyFork2Fork, RemoveForkToFork) {}
OnFailure (RemoveForkToFork, RemoveJoinToJoin) {}
OnFailure (RemoveJoinToJoin, initialize) {}
OnFailure (initialize, data_produces) {}
OnFailure (map_exec, map_decision) {}
OnFailure (data_produces, data_consumes) {}
OnFailure (data_consumes, compile_init_script) {}
OnFailure (map_decision, map_fork) {}
OnFailure (map_fork, map_join) {}
OnFailure (map_join, map_finish) {}
OnFailure (map_finish, map_start) {}
OnFailure (map_start, success) {}
```

Listing 3.1: Schedule of model transformation rules used for mapping PM to SCCD

This schedule exposes a number of steps starting at the 'Initial' step, where it has the first rule as a second parameter, i.e. 'create_orchestrator'. Each step is connected to two other steps with an 'OnSuccess' or 'OnFailure' function. Hence, each rule in this case is an entry as a parameter on both steps where execution can follow. For example, the first rule ('create_orchestrator') is a parameter on both functions such that if execution finds a match or not the next rule to execute is 'create_association'.

The first rule is used to create the Orchestrator SCCD, and is enclosed in an *Atomic* rule-block, and it is depicted on figure 3.15. This rule searches for the SCCD diagram entity in the LHS and adds a new SCCD (the Orchestrator) to the diagram on the

RHS. On the RHS of this rule, the Orchestrator Class also sets itself as the default Class of the SCCD model and declares, in its constructor, variables used to manage control-flow and data-flow during execution of the process model it is 'mapped' to. These variables are of a *dictionary* collection type with a key entry of 'id' attribute of each node. The Orchestrator uses these variables as follows:

- *token*: each entry in this collection has a Boolean type value used to emulate the flow of control. If a node has a valid token, it implies control is currently at the node.
- *finish*: each entry in this collection has a Boolean type value used to indicate if an activity has currently finished executing or not. This is only used by nodes preceding a Join construct to signal they have finished for synchronization of a parallel region.
- *data*: each executable activity entry has a 'produces' and 'consumes' collection entry in this variable and it is used to record data-flow in both directions.
- *decision_consumes*: each decision node entry has an actual value of data consumed. This value is assigned during execution and is used to dictate the flow of control.

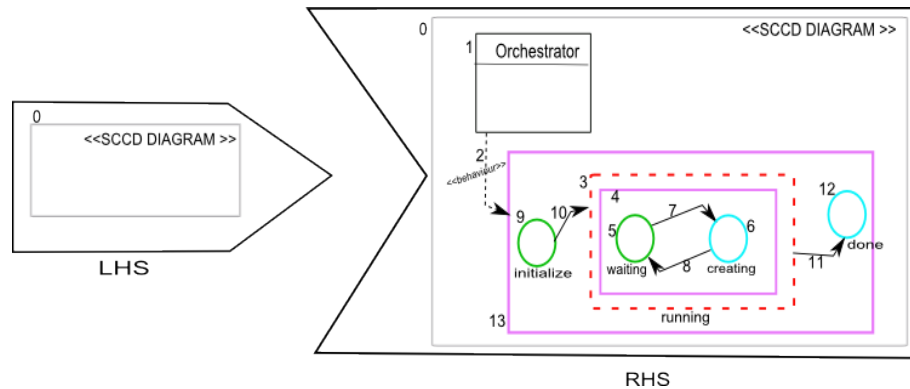


Figure 3.15: Model transformation which creates the Orchestrator SCCD enclosed in an *Atomic* rule-block

The Statecharts of the Orchestrator SCCD has three states as discussed in section 3.1.2. In figure 3.15, the parallel state has an orthogonal component with a 'waiting' state that transitions to 'creating' state, so long as the queue is not empty. When the transition attached triggers, it communicates with the object manager to spawn the SCCD instance with the first name from the queue.

The 'create_association' rule, enclosed with *ForAll* rule-block, is used to create an association among the Orchestrator SCCD instance and the rest of the SCCD instances related to executable activities. These associations can be used by the Orchestrator to send to the Object Manager of SCCD to create the SCCD instances.

The 'ModifyForkToFork' and 'RemoveForkToFork' rules, depicted on figure 3.16 and 3.17 respectively, are used together for the same purpose. When a Fork node in PM is connected to another Fork, it is semantically equivalent to remove the second Fork and simply create an association edge between the first Fork and the target nodes of the second Fork. This will make the mapping of Fork node simpler to handle by avoiding unnecessary pattern searching. Thus, the 'ModifyForkToFork' rule is enclosed in an *Atomic* rule-block to find a match (as in the LHS) and creates (in the RHS) an edge between the first fork and the generic Activity associated with the second fork. In the schedule, the 'ModifyForkToFork' rule loops until a match is not found, which by then the second fork node can be removed by the following rule that searches for all 'Fork-to-Fork' matches at once using *ForAll* rule-block.

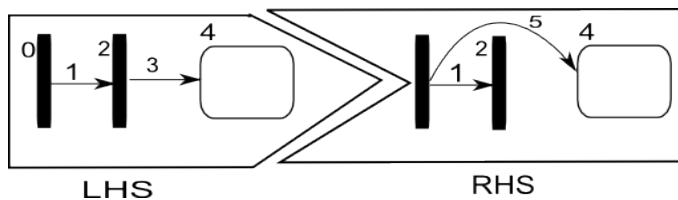


Figure 3.16: 'ModifyForkToFork' rule

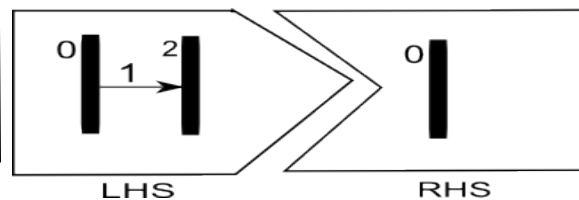


Figure 3.17: 'RemoveForkToFork' rule

Figure 3.18: Model transformation rule in (a) enclosed in *Atomic* rule-block and (b) in *ForAll* rule-block

Node Type	Variable	Initial Value
Start	token[id]	True
Decision	token[id]	False
	decision_consumes[id]	None
Join	finish[preceding_node_id]	False
Exec	token[id]	False
	data[id]['consumes']	Empty List
	data[id]['produces']	Empty List
Fork/Finish	token[id]	False

Table 3.1: Initial configuration of variable instances of the Orchestrator SCCD

Similarly, the 'RemoveJoinToJoin' rule searches for all matches where a Join node has a direct association with another Join, and that the second one is connected to a generic Activity. Again, it is semantically equivalent to remove the second join node and create a new association between the first Join node and the generic Activity.

The 'initialize' rule has, in the LHS, a basic state (i.e. the initial state of the Orchestrator SCCD) and a generic activity in PM. This rule is enclosed in a *ForAll* rule-block to find a match equivalent to the Cartesian product of the basic state and all nodes represented by the generic activity type. For every match found, based on its actual type, it initializes the variables of the Orchestrator in the RHS. The configuration of variables based on the activity type is shown in table 3.1. Note that node 'id' attribute is used as a key entry, and if the generic activity is of type Join, all its preceding nodes will have an entry in the 'finish' variable as well.

The 'map_exec' rule, as shown in the LHS of figure 3.19, searches for all matches with a pattern having an executable type

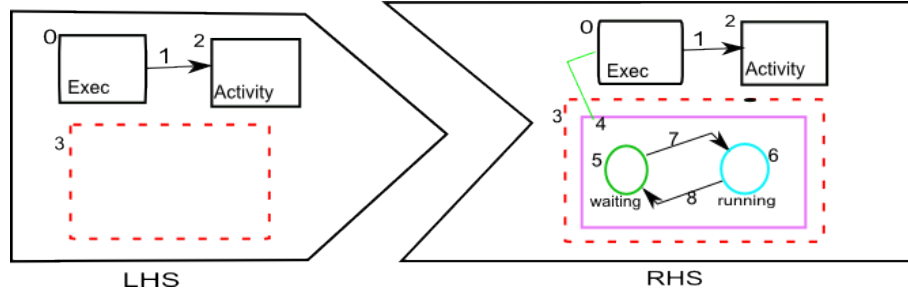


Figure 3.19: Model transformation rule enclosed in a *ForAll* rule-block for mapping an *Executable* activity to an orthogonal component in the Orchestrator SCCD parallel state.

connected to a generic activity node using a generic control-flow edge. The search also includes the parallel state in the Orchestrator SCCD. For each match found, it adds a new orthogonal component in the parallel state and configures the attributes of states and transition created. The transition's 'guard' attribute is set for a valid token and the 'script' attribute is defined as the successor activity (in the RHS) being assigned a valid token. The orthogonal component is mapped to the executable construct with a *traceability* link. This link can be used by other successor rules in the schedule for searching new patterns created during model transformations. In addition, they can be used in debugging operation by providing a *trace* to the mapping of constructs belonging to different model instances. The orthogonal component has two states describing the behaviour of the SCCD instance an executable activity corresponds to, and it is discussed in section 3.1.2.

The two rules related to data-flow, namely 'data_produces' and 'data_consumes', have a similar purpose. In the LHS, the rules search for all matches satisfying the pattern where an executable activity connected to a data node is found. This edge is an outgoing edge for 'data_produces' rule and an incoming edge for 'data_consumes'. There is also a search for the initial state of the Orchestrator SCCD, as shown in the code snippet in figure 3.2. Note that the constraint on the LHS is used to identify the initial basic state of the Orchestrator SCCD, as there can be multiple instances of basic states in the input model. For each match found, in the RHS, it uses the initial state of the Orchestrator to initialize, in its OnEntryScript attribute, the *data* variable instance used to record meta-data of models in both directions.

The 'map_decision' rule has in its LHS a pattern search for the parallel state of the Orchestrator SCCD and data node connected to a decision using the 'DecisionConsumes' association edge. This association has a constraint defined in the abstract syntax of the PM language, which makes it a binding requirement for a decision node to consume exactly one data. For each match found, in the RHS it adds a new orthogonal component with a child basic state inside the Orchestrator Parallel state, and configures the

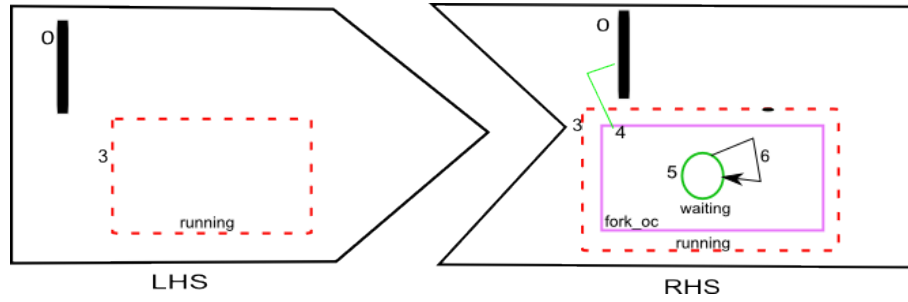


Figure 3.20: Model transformation rule enclosed in *ForAll* rule-block used for mapping *Fork* to an orthogonal component in the Orchestrator SCCD parallel state

decision_consumes variable instance. This variable is assigned a value based on the run-time result of data (model attribute) the decision node consumes. The basic state in the orthogonal component has a transition targeting itself. This transition's 'guard' attribute is set for a valid token and the 'script' attribute is defined as the successor activity (in the RHS) being assigned a valid token. The orthogonal component child state describes the behaviour of a decision node it is mapped to, as discussed in section 3.1.2.

```
{Contains} ForAll data_produces {
  LHS {
    Pre_PM/Exec pre_0 {
      label = "0"
    }
    Pre_PM/Data pre_1 {
      label = "1"
    }
    Pre_PM/Produces pre_2 (pre_0, pre_1) {
      label = "2"
    }
    Pre_SCCD/BasicState pre_3 {
      label = "3"
    }
  }
  constraint = $
    Boolean function constraint (model : Element, mapping : Element):
      Element children
      String parent
      Element class_beh
      String class
      children = allIncomingAssociationInstances(model, mapping["3"], "SCCD/
        composite_children")
      if list_len(children) >0:
        parent = readAssociationSource(model, set_pop(children))
        class_beh = allIncomingAssociationInstances(model, parent, "SCCD/behaviour")
        if list_len(class_beh) >0:
          class = readAssociationSource(model, set_pop(class_beh))
          if (value_eq(read_attribute(model, class, "name"), "Orchestrator")):
            return True!
          else:
            return False!
        else:
          return False!
      else:
        return False!
    $
}
RHS {
  Post_PM/Exec post_0 {
    label = "0"
  }
  Post_PM/Data post_1 {
    label = "1"
  }
  Post_PM/Produces post_2 (post_0, post_1) {
    label = "2"
  }
}
```

```

}
Post_SCCD/BasicState post_3 {
    label = "3"
    value_onEntryScript = $
        String function value(model : Element, name : String, mapping : Element):
            String script

            script = read_attribute(model, mapping["3"], "onEntryScript")
            if value_eq(has_value(script), False):
                script = ""
            script = string_join(script, "\tentry= dict_create()\n")
            script = string_join(script, "\tdict_overwrite(entry, 'tag', '')\n")
            script = string_join(script, read_attribute(model, mapping["2"], "tag"))
            script = string_join(script, "''\n")
            script = string_join(script, "\tdict_overwrite(entry, 'model', '')\n")
            script = string_join(script, read_attribute(model, mapping["1"], "name"))
            script = string_join(script, "''\n")
            script = string_join(script, "\tdict_overwrite(entry, 'metamodel', '')\n")
            script = string_join(script, read_attribute(model, mapping["1"], "type"))
            script = string_join(script, "''\n")
            script = string_join(script, "\tlist_append(attributes['data'][''])\n")
            script = string_join(script, read_attribute(model, mapping["0"], "name"))
            script = string_join(script, "''[['produces'], entry)\n")
            return script!
        }
    }
}

```

Listing 3.2: Model transformation rule in the Modelverse

The 'map_fork' rule, shown in figure 3.20, searches for all matches having a fork node, in addition to the parallel state of the Orchestrator SCCD. Note that the LHS does not specify a successor node to the fork, as they are at-least two and possibly more of them. Instead, action language code is used which makes it possible to search all successors at the same time when this rule executes. The successor nodes are then used to configure the *token* variable instance and the attributes of the transition created. For each match found, the RHS rule also adds a new orthogonal component to the parallel state and maps it with the Fork node using a *traceability* link. The behaviour of the newly created orthogonal component mapping a fork node is described in section 3.1.2.

Similarly, the 'map_join' rule, shown in figure 3.21 searches for all matches where a join node is connected to its successor using

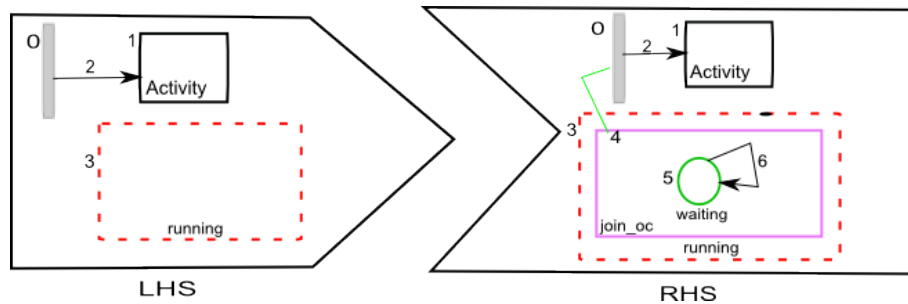


Figure 3.21: Model transformation rule enclosed in *ForAll* rule-block used for mapping *Join* node to an orthogonal component in the Orchestrator SCCD parallel state

a generic control-flow edge. The search also includes the parallel state of the Orchestrator SCCD. For each match found, the RHS rule adds a new orthogonal component to the parallel state and configures the *finish* variable and an attribute of the transition created. Similar to the 'map_fork' rule, once a match is found, all preceding nodes to the Join construct are searched for in action language code. It uses this search to construct the 'guard' condition attribute of the transition created targeting the 'waiting' state. This condition specifies that all preceding nodes have a valid 'finish' variable instance. The orthogonal component has a single basic state denoting the behaviour of a join construct being mapped to, as described in section 3.1.2.

4

Visual Editor

An application front-end is developed for both languages (PM and SCCD) which uses their graphical concrete syntax and essentially enables for visually modelling workflows at different levels of abstraction. This GUI editor consists of mainly two parts, one that make up for a PM editor and the other for SCCD. Hence, the overall goal was to obtain a user-friendly modelling environment which integrates the two languages and interacts with the Modelverse. The Modelverse [22], serves as a model repository in addition to being a multi-paradigm modelling and simulation back-end. This section discusses the design choices made during development, followed by highlighting essential functional requirements incorporated in both parts of the editor.

4.1 Design Choices and Application Architecture

The developed front-end application has high interactive features with complex user interface (UI) requirements, such as multiple instantiation of an element on canvas, hierarchical layering of constructs and drag and drop of such constructs. This and many more complex features of the editing environment requires to explicitly model its behaviour. Thus, SCCD is chosen as an appropriate formalism given the UI can be characterized with a behaviour that is reactive, timed, autonomous and dynamic- structure. In the implementation, SCCDXML [19], which is a textual concrete syntax of SCCD, is used. In addition, the application front-end utilizes external libraries to implement non functional requirements. These libraries include mainly widgets from Python's Tkinter [4] package, i.e. a GUI programming toolkit.

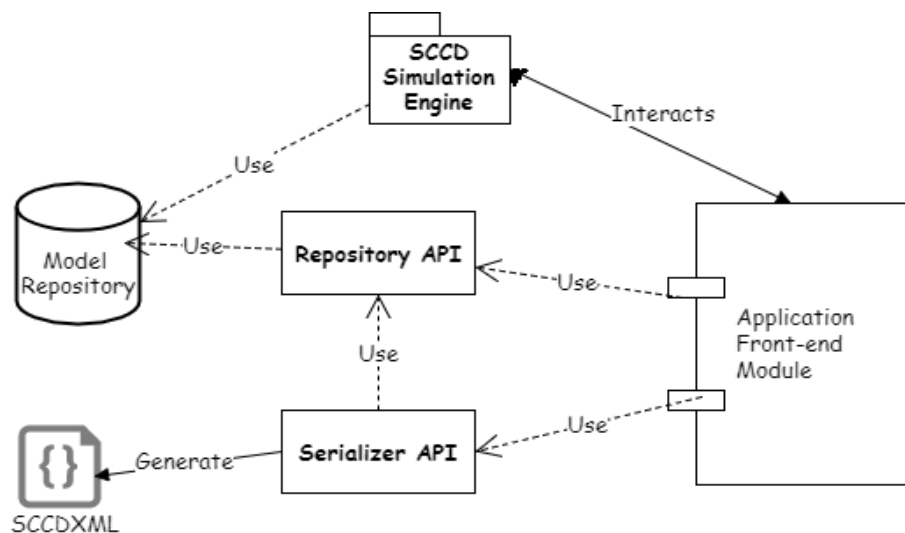


Figure 4.1: An overview of the implemented application architecture.

The application architecture can be summarized in figure 4.1. The *Application front-end Module* consists of SCCD classes and their Statecharts for modelling the behaviour of the UI. For example, figure 4.1 depicts a code fragment of the SCCDXML used to explicitly model all instances of a 'Button' entity. This SCCD class inherits from Tkinter's Button widget and, in its constructor, instantiates it to create a button for the associated "Toolbar" SCCD.

```

1 <class name="Button">
2   <relationships>
3     <association name="parent" class="Toolbar" />
4     <inheritance class="widget" priority='0' />
5     <inheritance class="tk.Button" priority='1' />
6   </relationships>
7   <constructor>
8     <parameter name="constructor_parameters" type='dict' default='{}' />
9     <super class="tk.Button">
10      <parameter expr="constructor_parameters['parent']" />
11      <parameter expr="*(constructor_parameters['visual']).get_params()" />
12    </super>
13    <super class="widget" />
14    <body>
15      self.label = constructor_parameters["event_parameters"]
16    </body>
17  </constructor>
18  <destructor>
19    <body>self.destroy()</body>
20  </destructor>
21  <scxml initial="main">
22    <state id="main" initial='initializing'>
23      <state id="initializing">
24        <transition event='set_association_name' target='../running">
25          <parameter name='association_name' type='str' />
26          <script>self.association_name = association_name</script>
27        </transition>
28      </state>
29      <state id="running">
30        <transition port='input' event="left-click" target='.' cond='tagorid == id(self)'>
31          <parameter name='tagorid' type='int' default='None' />
32          <raise event="button_pressed" scope="narrow" target="'parent'">
33            <parameter expr="self.label" />
34          </raise>
35        </transition>
36        <transition port='input' event="enter" target='.' cond='tagorid == id(self)'>
37          <parameter name='tagorid' type='int' default='None' />
38          <script>
39            self.tooltip.showtip()
40          </script>
41        </transition>
42        <transition port='input' event="leave" target='.' cond='tagorid == id(self)'>
43          <parameter name='tagorid' type='int' default='None' />
44          <script>
45            self.tooltip.hidetip()
46          </script>
47        </transition>
48      </state>
49    </state>
50  </scxml>
51 </class>

```

Listing 4.1: SCCD instance of a Button type used for developing the visual editor expressed in SCCDXML notation

The Statecharts of this SCCD model is enclosed with <scxml> tag, and in there, we can see that the state with id 'running' has a transition with an event 'left-click'. When this transition triggers, it will raise an event with a narrow scope to the associated Toolbar SCCD and sends as a parameter a unique identifier (name) of the Button instance. The transition also listens to an 'enter' and/or 'exit' events where upon triggering calls a method of the inherited 'Widget' external class which visually shows a tooltip for the button instance. The associated Toolbar SCCD, with an association name 'parent', is the one responsible for the creation and deletion of a Button SCCD instance.

The application front-end essentially interacts with a wrapper to the Modelverse, i.e. the *Repository Application Programming Interface (API)*. This wrapper initially connects to the Modelverse (via HTTP) and provides a high-level interface which initiates calls to the Modelverse that perform operations on models found in the repository. These operations mainly include creating, fetching, updating and deleting of relevant data pertaining to both model types (PM and SCCD). The Application front-end

module includes SCCDs which relate to the simulation interface developed. Thus, these classes serve as an interface to interact with the *SCCD Simulation Engine* to visually characterize the run-time behaviour of the application. Note that the SCCD simulator is in the Modelverse and has access to SCCD models. The *Serializer API* can optionally be used to generate SCCDXML from SCCD models in the Modelverse. A compiler already exists for SCCDXML and can be used to generate executable code for multiple platforms, such as Python, JavaScript, and CSharp.

4.2 Process Model Editor

When the application launches, the PM editor part of the application front-end is presented to the user. As depicted on figure 4.2, the editor window consists of a toolbar and a canvas. The labels on the toolbar include PM modelling constructs. When the toolbar loads these constructs, it requests for all types defined in the abstract syntax meta-model, via the Modelverse wrapper (Repository API). The abstract syntax is depicted in figure 3.11 of the previous chapter. Thus, a label in the toolbar is created if its type exists in the meta-model. In this manner the application front-end easily adopts to changes in the definition of the meta-model and enables to preserve consistency between the concrete syntax and abstract syntax of the modelling language.

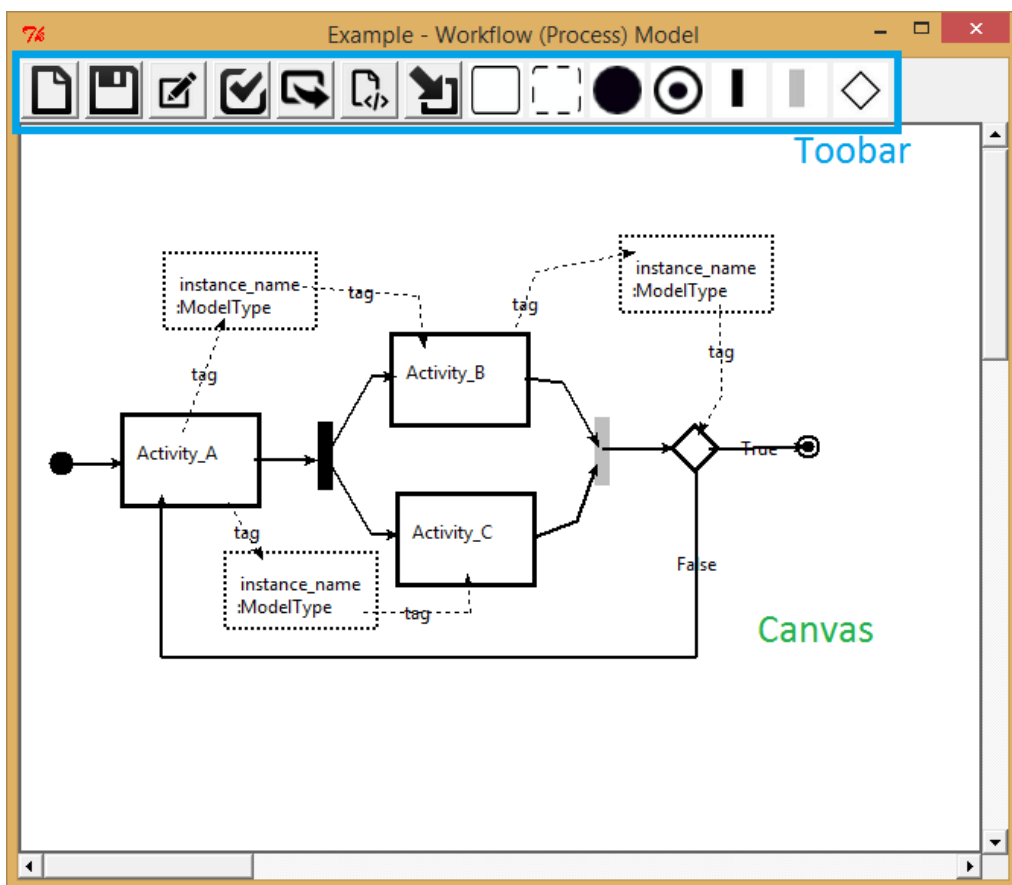


Figure 4.2: The developed visual editor for PM.

The concrete syntax of the PM language includes nodes, i.e. activities (of six types) and data, and edges associating these nodes. When edges are created, the canvas requests for allowed connection types based on the definition of the abstract syntax meta-model. This procedure involves requesting the Modelverse, via the repository API, for all association types between source and target constructs selected while creating an edge. If the result returned is empty, the user is prompted with an error message indicating no association exists for the selected source and target types. Similarly, if the result is a list of associations of size greater than one (such as the case for an executable activity and data), the user is prompted to select which association to create, as in figure 4.3. Otherwise, the association is automatically created given the result is only one. The abstract syntax has specialized associations that inherit from the generic 'Next' association which represents control-flow. These associations are used to constrain the user to create allowed connection types in the front-end, as discussed above. Moreover, this way of creating an association implies that the application front-end automatically adopts to modifications in the definition of the modelling

language abstract syntax, such as creating a new association type or deleting an existing one.

The graphical concrete syntax of the PM developed in this editor essentially provides a clear distinction among these various constructs. Data nodes and edges representing data-flow are created using dotted-lines. Similarly, a Join node and a Fork node have different visual representation, as opposed to the UML 2.0 AD's concrete syntax which makes no distinction between these two types.

Nodes can be created on canvas and positioned as desired. While positioning, using a drag-and-drop behaviour, an associated

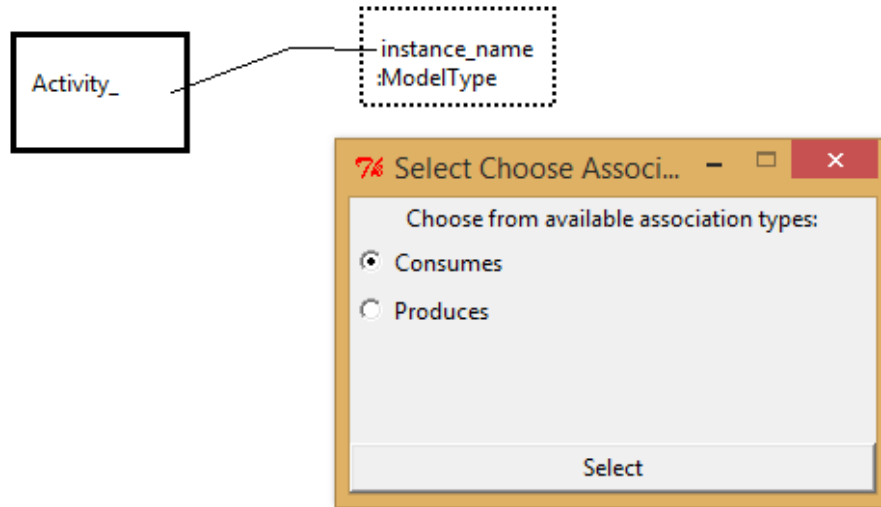


Figure 4.3: Prompting user for selecting association types defined in the abstract syntax of PM

edge (if any) is also automatically animated following the position of a node. When creating an edge, the user can be able to optionally add multiple in between angle points (control points) used to edit the position of the edge. The developed GUI provides separate windows for editing the attributes of a given construct. After editing an edge attributes, their values are displayed as labels on the edges, such as the one for data-flow edges or outgoing edges of a Decision node.

The application front-end also enables for rendering models on canvas from the repository. While loading a selected model instance, the constructs are placed on canvas preserving their last position saved.

4.3 SCCD Editor

As discussed in the previous chapters, SCCD models are encapsulated in PM executable activities. Hence, an instance of the SCCD editor is launched when an executable activity is selected with a double-click event. At this time, the application front-end uses the repository API to inquire which SCCD class is mapped to the activity and ultimately presents it on the editor, as shown in figure 4.4. The SCCD class can be positioned on canvas as desired and also has a separate window with input fields to update its attributes. These attributes are optional and include *attributes*, *methods*, *constructor*, and a *destructor*. The *name* property which is disabled, is by default set to the name of the executable activity the SCCD relates to. Class methods can be created in a separate window and has input fields for the *name*, *parameters*, *body* and *return type* attributes of a method. The behaviour edge, with a dotted-line, is used to associate the Class with its Statecharts.

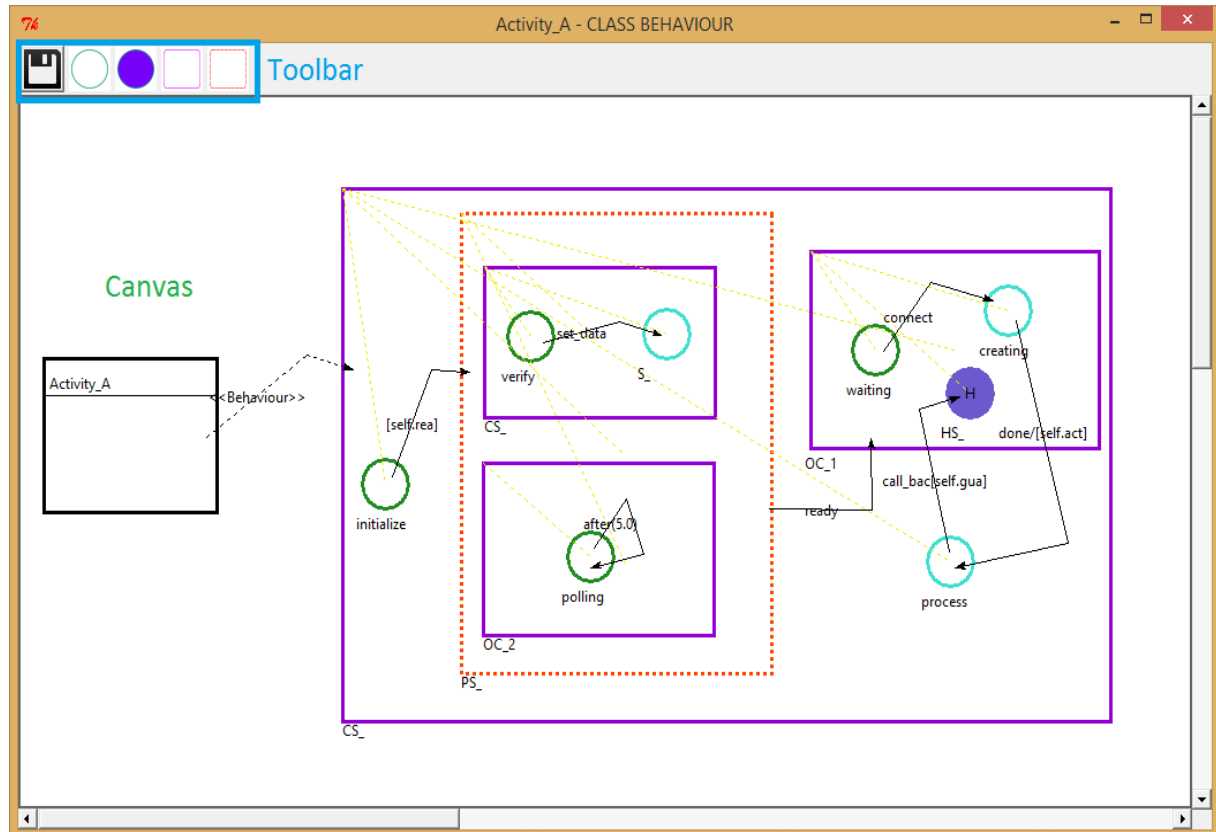


Figure 4.4: SCCD visual editor for Activity_A in figure 4.2

The SCCD editor also enables to create modelling constructs on canvas and position them as desired. Positioning of constructs also involves automatic positioning of associated edges and hierarchical children, if any. States can be connected via an edge, namely a *Transition* edge. Since Statecharts have four modelling constructs, the concrete syntax developed for this editor essentially provides clear distinction among them. In addition, Statecharts also adds a notion of hierarchy and this is represented in the editor with the most user intuitive manner. A state can be dragged into another state supporting hierarchy and be dropped. At this time, the parent state automatically adjusts its size to accommodate the new child, and creates a semi-transparent line (shown in yellow dotted-line on figure 4.4) to emphasize the child-parent relationship.

The SCCD editor enables to update settable properties of a state that includes its *name* and whether or not it is a *default* state. The latter is only available if the parent is not a Parallel state. If the state is a default one, the editor changes its appearance by painting its edge with a green colour, as shown in figure 4.4. The state attributes editor window, shown in figure 4.5, has input fields to define entry and exit actions for a state. As a state can have an unlimited number of raise events, it can be added by clicking the Add button. A separate pop-up window then appears with input fields for an *event*, *scope* and *target* attributes as well as the option to add parameters. For a History state, there is a radio-button to select its type between *shallow* and *deep*.

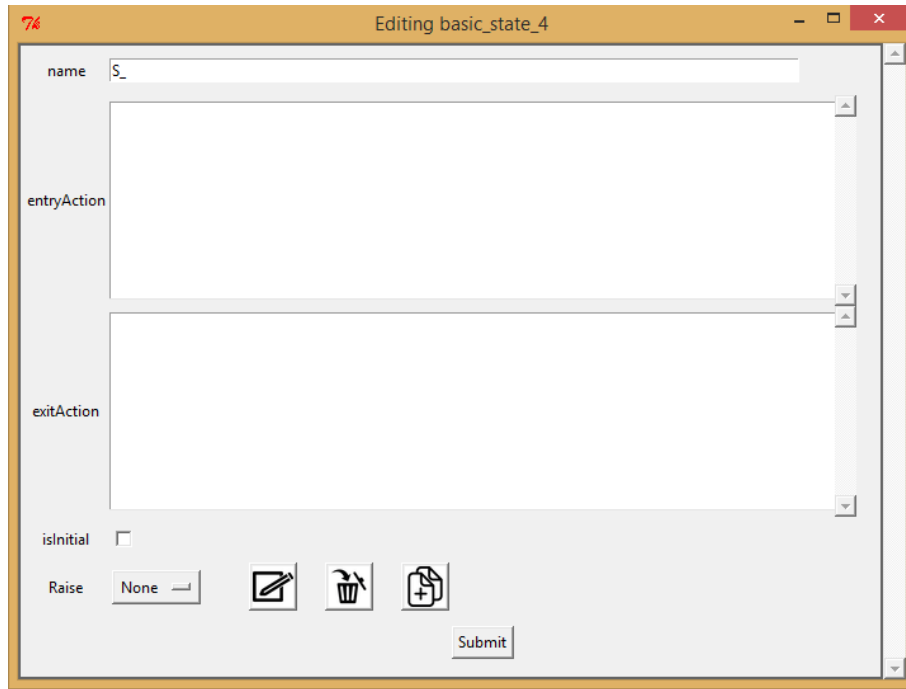


Figure 4.5: State attributes editor in SCCD

Similarly, a transition editor window has input fields to set its attributes including an *event*, *guard*, and *after*. Furthermore, there is an option to add formal event parameters and a raise entry associated to a transition. A transition has a label displaying a limited number of characters of its properties in the format $T[C] / R[A]$, where T is the trigger, C a condition, R raised events, and A the action. When creating a transition, the user can be able to optionally add multiple in between angle points (control points) used to edit the position of the edge, as depicted on figure 4.6.

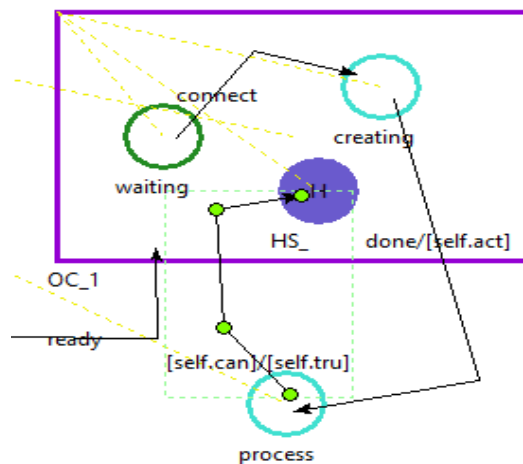


Figure 4.6: Multiple angle points of an edge used for editing its position

Finally, the editor canvas has zooming capabilities to optimally utilize the diagram layout and make the editor scalable for large models. Similar to the PM editor, SCCD models can also be rendered on canvas from the repository. While loading, the instance model constructs are placed in their last position saved.

5

Example

In this chapter, we will demonstrate the semantics of PM given in SCCD, by applying the power window case study discussed in chapter 2. In addition, unlike the process model previously discussed for the power window case, this instance of PM has an activity responsible for obtaining an external service. Thus, we will also discuss this activity's behaviour encapsulated in an explicitly modelled SCCD.

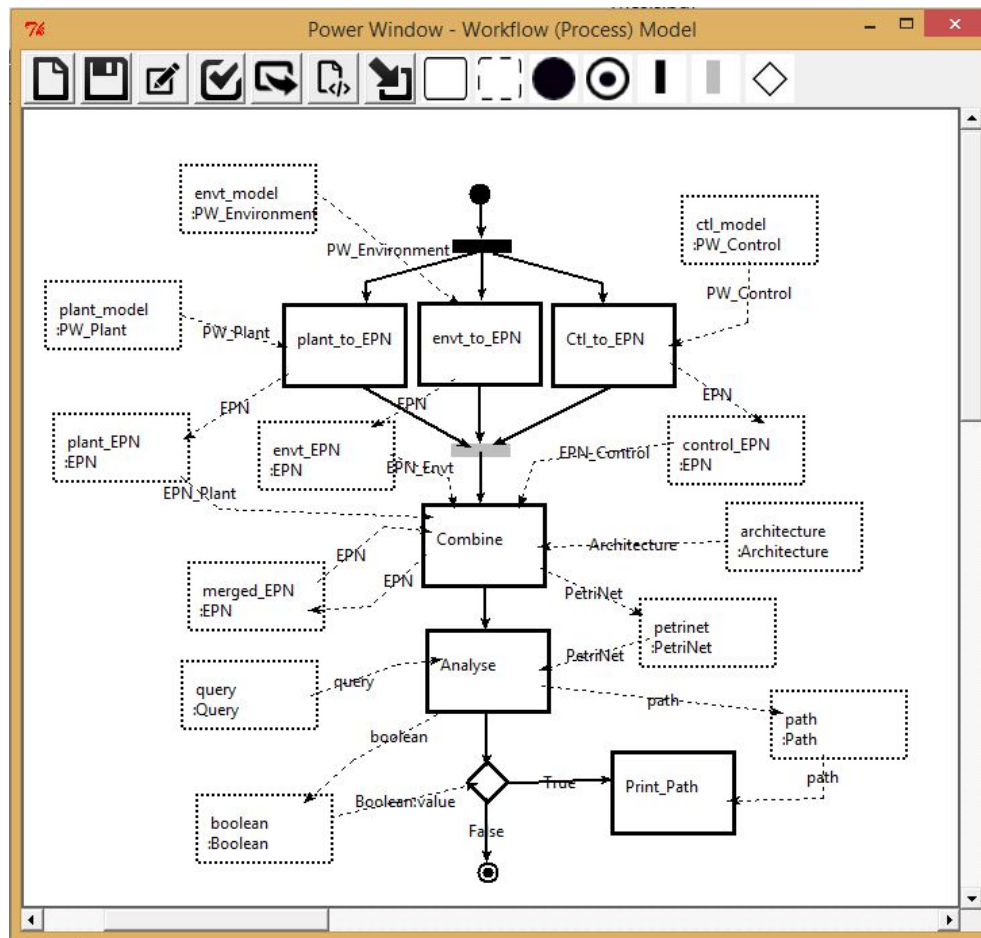


Figure 5.1: A process model developed in the visual editor for the power window case study.

5.1 Case Study

The example we use is the power window case study discussed in chapter 2, however with slight changes. Figure 5.1 depicts the PM of this example using the developed visual editor and includes automatic operations and the models used by these operations. Note that manual operations are not included, as the process is only used for demonstrating the semantics of PM given in SCCD. All DSMLs and model instances in the FTG are pre-loaded to the Modelverse during environment setup.

As discussed in chapter 2, the intention of the PM in 5.1 is on the verification aspect of the power window, i.e. the window should never go up when an object is inserted through. In that regard, the PM control-flow originates with a concurrent behaviour where model transformations of instance models, such as Plant, are mapped to Encapsulated Petri-nets (EPNs). These operations are synchronized by the Combine operation which consumes the EPNs and the Architecture model to produce a combined marked Petri-net model. This model in combination with a safety query model is subsequently used by the 'Analyse' activity, which represents an external service. The external service is not declared in the FTG, and the service obtained is an optimized reachability analysis of the combined Petri-net model on a state predicate specified by the safety query model, as shown in figure 5.1. If the query is found, the user is presented with a path from the Petri-net model that demonstrates why an unsafe situation is reached. Otherwise, the system is deemed safe, and the process terminates.

The behaviour of all executable activities in the PM is explicitly modelled in SCCD. In addition, the PM itself is mapped to SCCD, as discussed in chapter 3. The model operation, responsible for this mapping, takes as input the PM and the SCCD model which expresses the behaviour of each executable activity in the PM. Following its execution, the model operation yields a single SCCD model that includes the Orchestrator SCCD instance where the PM is mapped to. This output SCCD model, depicted on figure 5.2, is then simulated. Since the output model is a result of automatic model operation, it does not have data about canvas position of constructs to render on the developed visual editor.

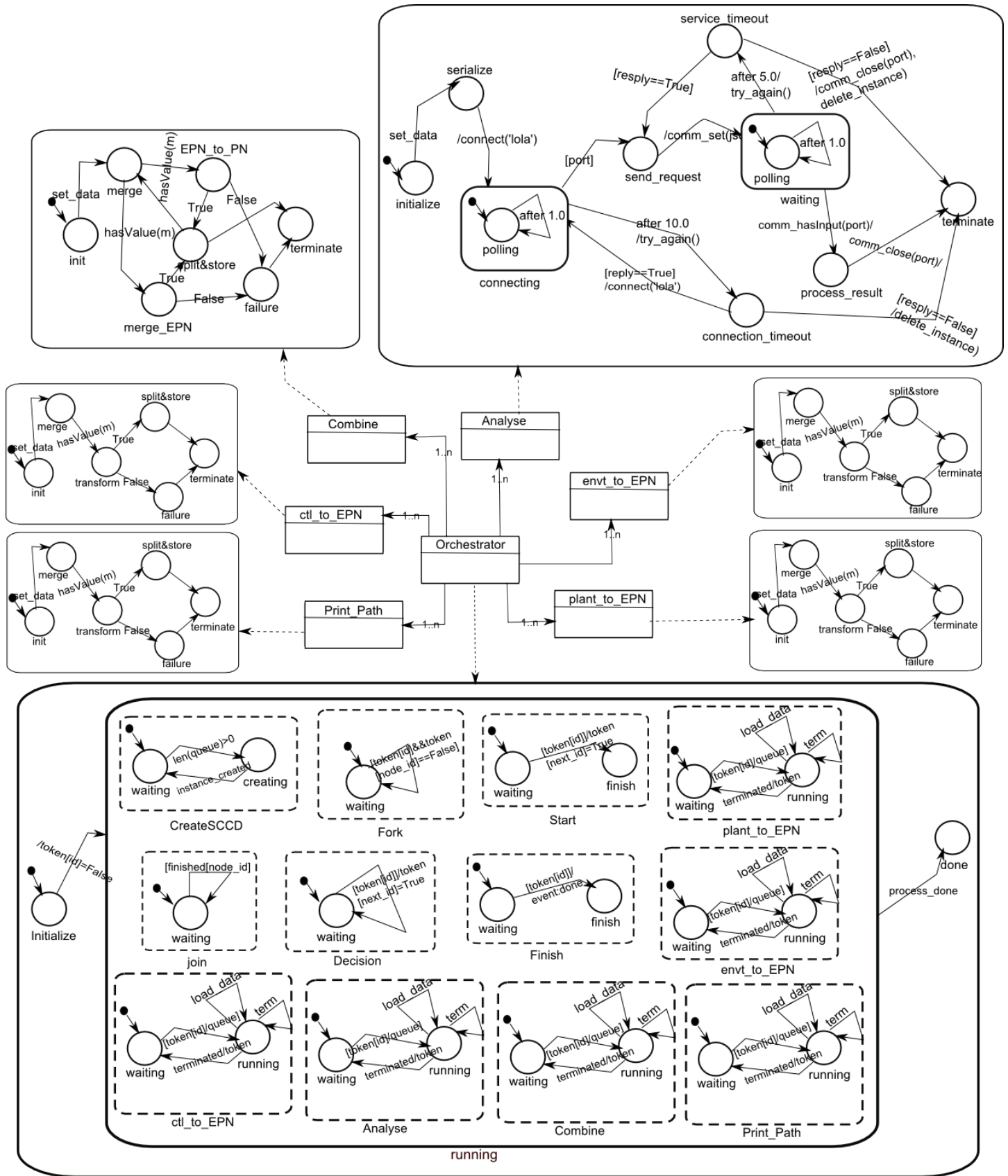


Figure 5.2: The executable SCCD model generated after mapping PM to the Orchestrator SCCD

We can see that the Statecharts of the Orchestrator Class has, in its parallel state, orthogonal components which are mapped to all activity types in the PM and uses them to manage control-flow and data-flow. As discussed in chapter 3, note the condition for a valid 'token' set upon the transitions attached to the initial state of these orthogonal components. When triggered, if the orthogonal component is a mapping for an executable activity, such as plant_to_EPN, it places its name in a 'queue' to be created. The 'CreateSCCD' orthogonal component has a transition attached to its initial state which gets triggered so long as the queue is not empty. At this time, it gets the name of the executable activity from the 'queue' and raises an event to the object manager for creating the SCCD instance corresponding to the activity with an identical name. The SCCD instance, by default, gets the name of the executable activity at design time in the visual editor. The 'running' state is then entered and waits until the SCCD instance informs of its termination by listening to an event on the attached transition, i.e. 'terminated' event. Upon triggering, this transition takes an action which then passes the 'token' to the successor node.

In addition, when orthogonal components mapped to an executable type enter the 'running' state, the transition targeting itself has an event 'load_data' that it listens to, which is a data request from the running SCCD instance. When triggered, the transition raises an event to send the appropriate data related to the executable activity. The data management, as discussed in chapter 3, takes place while mapping the PM to SCCD in the model transformation. In that regard, when the Orchestrator Statecharts enters the parallel state, where it manages the process flow, it remains well informed of which data is to be consumed and/or produced by which executable activity. Hence, transitions in orthogonal components make use of the *data* variable instance of the Orchestrator Class to extract the appropriate data related to the executable in which the orthogonal component is mapped to.

Figure 5.3 depicts a run-time behaviour of the Combine executable activity, modelled in SCCD. In the Statecharts, there are

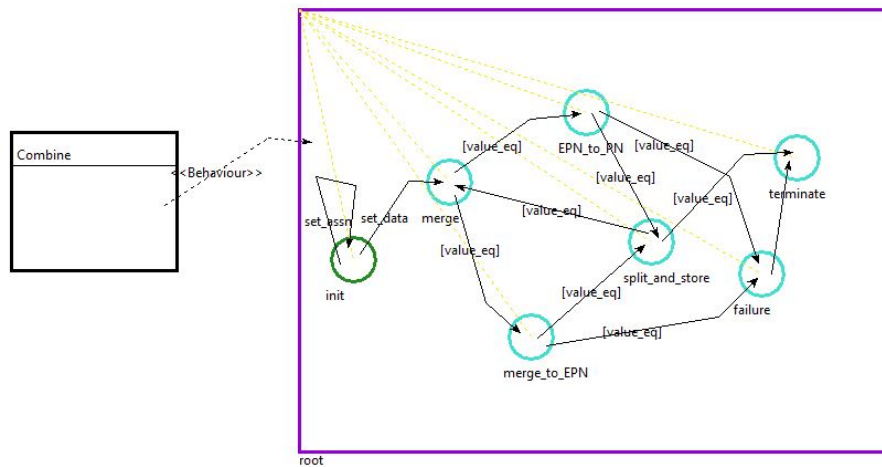


Figure 5.3: The behaviour of Combine activity in figure 5.1 modelled in SCCD

two types of model operations. The first one is for general model management purpose, represented by the basic states 'merge' and 'split', for model merging or splitting respectively. While the other is responsible for model operations which depend on the semantics of the languages of models consumed and produced by the operations. These operations in figure 5.3 are represented by the basic states 'merge.to_EPN' and 'EPN.to_PN'. Upon initialization of the Statecharts, the initial state requests for data to be used by the operations. The transition attached to the initial state listens to receive data from the Orchestrator SCCD. When triggered, the transition takes an action to get the actual models based on the data received, and merges them before the model operation is invoked in the 'merge.to_EPN' or 'EPN.to_PN' states. After entering these states a model transformation gets executed after RAMification of the meta-models, as discussed in chapter 2. The transformation expects as a result a Boolean value. If the result is False, the state transitions to a 'failure' state where it prompts the user with an error message. Otherwise, after splitting and saving the output models, the state transitions to the 'terminating' state, where it raises an event to notify the mapped orthogonal component in the parallel state of the Orchestrator SCCD instance. The behaviour of model operations declared in the FTG is almost similar to the one discussed now. In the PM, 'merge.to_EPN' and 'print_path' operations are defined in neutral action language code, while the others use model transformation through RAMification. In the following section, we will discuss the behaviour of the PM activity Analyse, which is explicitly modelled in SCCD and is responsible for external service interaction.

5.2 External Service Interaction

The Modelverse has a complete support of the FTG+PM framework as illustrated in the power window case study in chapter 2. In the FTG, there are various model operations defined there in, such as the one for conducting reachability analysis of a

Petri-net model. The reachability graph, from a marked Petri-net model, is developed first by using action language. This graph, coupled with a query model specifying the state predicates, are used as input models to a model operation which performs the reachability analysis. Although this method is feasible and already enacted, performance issues can still be at stake and obtaining an optimized reachability analysis is desirable.

Thus, we used the service of LoLA, which is a Petri net model analyzer tool available external to the Modelverse environment. LoLA [13] (a Low Level Petri Net Analyzer) has been implemented for the validation of reduction techniques for Petri-net reachability graphs. Most properties supported by LoLA can be specified in temporal logic, either using branching time logic CTL or the linear time logic LTL. Instead of running a general LTL or CTL model checking algorithm, LoLA first checks whether the property can be reduced to a search for the desired property, i.e. in this case a simple reachability query. If so, it runs an optimized state space exploration technique, such as partial order reduction (the *stubborn set* method), to that particular property. *Stubborn set* computation abides to rules ensuring the desired properties are preserved in the reduced state space.

5.2.1 Implementation

The Modelverse itself runs as a service, and enables for other (external) services to connect and to communicate data. An external service is required to connect (via HTTP) and register first before exchanging data with a client of the Modelverse. Once registered, the client can connect to the registered service which allows for synchronous data transfer protocol. LoLA does not require any user interaction (except for aborting execution) and purely works in batch mode. Its input and output can be organized by generating files, or it allows to read from standard input and write to standard output.

LoLA expects, as its input, a Petri-net model in a clearly arranged ASCII based language. It uses proprietary file formats which is simple to generate and easy to parse. LoLA's Petri-net file formats contains of three main parts as follows [13]:

- Places are defined as a comma-separated list of place names, beginning with keyword **PLACE**. Optionally, for a list (sub-list) of places, a token bound (capacity) can be defined using keyword **SAFE** followed by a number, 1 being the default. If a place is listed several times, the respective arc weights are added.
- An initial marking begins with keyword **MARKING** followed by a list of comma-separated assignments of token numbers to place names. An initial marking of one is assumed if only a place name is listed.
- A transition list begins with keyword **TRANSITION**, followed by a name and an optional fairness assumption (**STRONG FAIR** or **WEAK FAIR**). Then, the transitions preset (**CONSUME**) and postset (**PRODUCE**) is given similar to a marking. An arc weight of 1 is assumed if not given explicitly.

The extended BackusNaur form (EBNF) grammars of this file format is provided in figure 5.4.

A serializer is implemented (in action language) which translates a Petri-net model, in the Modelverse, to a format which uses the

```

net ::= 'PLACE' place_lists 'MARKING' marking_list? ';' transition+
place_lists ::= ( capacity? place_list ';' )+
capacity ::= 'SAFE' 'NUMBER'? ';'
place_list ::= nodeident ( ',' nodeident )*
nodeident ::= 'IDENTIFIER' | 'NUMBER'
marking_list ::= marking ( ',' marking )*
marking ::= nodeident ( ':' 'NUMBER' )?
transition ::= 'TRANSITION' nodeident fairness?
             'CONSUME' arc_list? ';' 'PRODUCE' arc_list? ';'
fairness ::= ( 'STRONG' | 'WEAK' ) 'FAIR'
arc_list ::= arc ( ',' arc )*
arc ::= nodeident ( ':' 'NUMBER' )?

```

Figure 5.4: EBNF grammars of LoLA's file format.

EBNF grammars of LoLA. In addition, the serializer also translates the Query model specifying the state predicate to a temporal logic formula to be used by LoLA. The output of this serialization is again encoded in a JSON format which can be transferred over a network. A wrapper for LoLA is also implemented which decodes the JSON input and invokes LoLA by providing its parameters, i.e. the input Petri-net model and state predicate.

After analysis is made, LoLA supports its results by providing a path from the initial marking to a marking that demonstrates why the property is (not) satisfied. This and other relevant data, such as reports in run time and size of the explored part of the state space, is provided as an output. The implemented wrapper uses this generated output to extract and send relevant data to

the client. This data can be a Boolean result of the reachability query and a state path, if the query is satisfied, encoded in a String format. In this implementation, the external service LoLA runs in an external virtual machine, and connected, via virtual network, to the Modelverse and registered as a service. Since the SCCD model for external service interaction, created in the developed visual editor in figure 5.5, uses neutral action language code, where the Modelverse has an interpreter for, transition labels in the figure may not clearly represent their attributes. Thus, figure 5.2 has a clear notation of these labels.

The SCCD model in figure 5.5 depicts an explicitly modelled run-time behaviour of the Analyse activity, in the PM on figure 5.1. Statecharts in SCCD inherently support modelling the behaviour of discrete-event systems that are *timed* (i.e. something must happen after x time unit), *autonomous* (i.e. it can trigger events and handle them autonomously), and *reactive* (i.e. it can react to external events, such as a user or external service). Hence, by modelling the behaviour of interaction in the PM at a low-level of abstraction using SCCD, we will demonstrate how the PM autonomously behaves in support of external service and user interaction.

This model raises an event to get data consumed and produced by the activity upon initialization. The initial state has a transition with an event 'set_data' that it listens to from the Analyse orthogonal component in the parallel state of the Orchestrator SCCD, as depicted in figure 5.2. After receiving data, the state transitions to a 'serialize' state where it gets the actual models (Petri-net

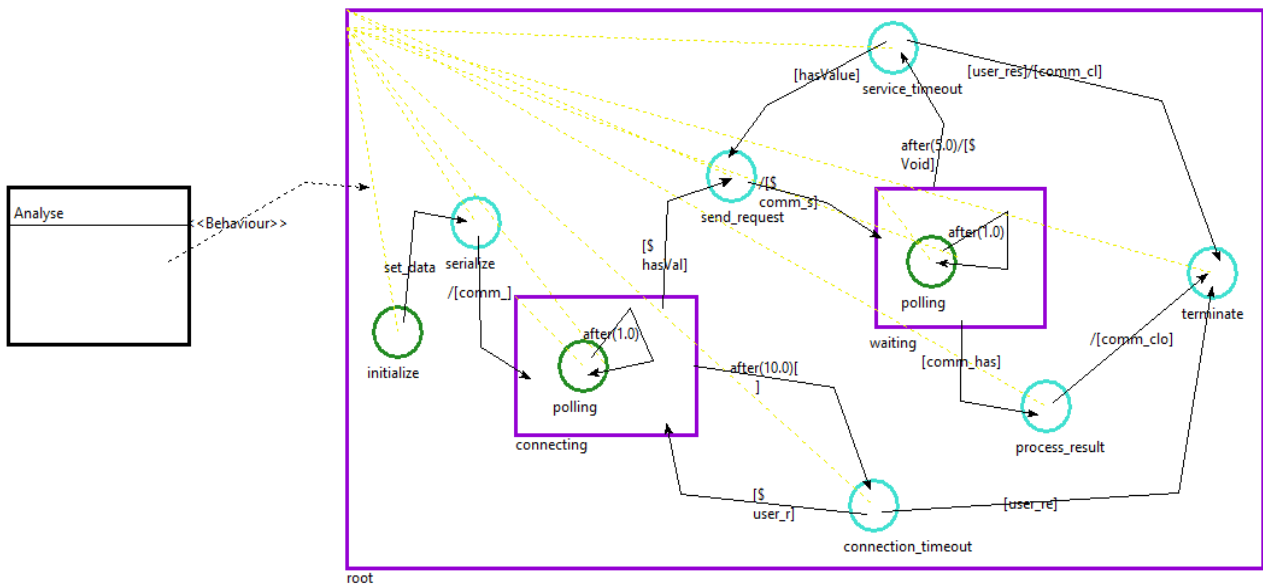


Figure 5.5: SCCD model in the developed visual editor which depicts the behaviour of an external service interaction.

and Query) based on the data received. The serializer is then invoked by passing the models as a parameter. The transition targeting the state 'connecting' takes an action which attempts to connect with a service name 'lola'. While in the 'connecting' composite state, note the initial state with a *timed* transition targeting itself. When triggered after every given time unit (1.0s), it polls the connection status by evaluating the condition placed on the transition outgoing from the 'connecting' state and targeting the 'send_request' state. This condition specifies that a port number is received when connection is made. If a port number is received, it transitions to the 'send_request' state where it sends the serialized data in JSON format. Otherwise, after a given time unit (10s), a state transition occurs to 'connection_timeout' state where a user gets a prompt to try again. If user responds with a yes, the Statecharts will transition back to the 'connecting' state and attempts to try again. Else, it raises an event to delete itself and transitions to the 'terminate' state.

After sending data in the appropriate format to the external service, the Statecharts transitions to a 'waiting' state until either a reply is received or a timeout occurs. This behaviour is also depicted on figure 5.5 where the 'waiting' composite state has a child state which polls after every given time unit for the reception of data, in a similar manner to the polling used for making connection. Note that the transition targeting the 'process_result' state has a condition which validates if there is a reply, by using the Modelverse interface that listens to the response of the external service. If a reply is received, the state transitions to 'processing_result' state, followed by terminating after closing connection. If a reply is not received after a given time unit (5s), a state transition occurs and the user is prompted for trying again. If user replies with a yes, the Statecharts goes back to the state where it sends the request again. Otherwise, it autonomously terminates after closing the connection with the external service.

6

Conclusion

In this thesis we assessed and demonstrated how it can be possible to model at multiple levels of abstractions to explicitly model workflow (process) behaviour such as concurrency, and essentially enact the Process Model (PM) itself. The semantics of a PM was given in a low-level language, i.e. Statecharts Class-diagram (SCCD), where its semantic domain is already defined and simulated. The PM used is a subset of the UML 2.0 Activity Diagrams, and in the context of Multi-paradigm Modelling (MPM), it is tightly coupled with the Formalism Transformation Graph, in the FTG+PM framework [9]. The PM describes, at a high-level of abstraction, the flow of execution of model operations (as activities) applied on a set of formalisms (as data) declared in the FTG. PM activities can also coordinate with external services and/or users. Hence, activities in PM are enacted either through their low level SCCD specification, or by calling upon external services. SCCD facilitates the specification of dynamic-structure systems that are timed, autonomous and reactive. This enabled the orchestration of PM activities with an external service and/or a user, by explicitly modelling its behaviour in Statecharts.

In order to denotationally map PM onto SCCD, we explored two approaches and showed their benefits and setbacks. Although sequential mapping approach is relatively intuitive and easy to debug, as it leverages common design patterns in both languages, it fails to easily map process models with complex behaviour, such as the interleaving of branches in parallel regions. Hence, a more generic (complete) approach was used which makes it possible to map any PM instance that conforms to its meta-model. This generic approach fully utilizes concurrency in SCCD, and maps each PM activity type to an orthogonal component inside a parallel state. Control-flow of the process is then managed using the routing of a shared "token".

Moreover, we applied the concept of *soundness* of workflow-nets [17] to identify undesired behaviour of PM, i.e. a control-flow with a Fork branch exiting a parallel region before synchronizing. Although such PM conforms to the meta-model of UML 2.0 Activity Diagrams, it can lead to process execution anomalies such as *improper termination*, where simultaneous multiple instantiation of activities occur. Thus, a thorough verification of PM instances based on the soundness property of workflow-nets is proposed.

An integrated visual modelling environment was developed in Tkinter on top of the Modelverse [23]. This visual editor combines the graphical concrete syntax of both (PM and SCCD) languages and enables for modelling at multiple levels of abstractions. SCCD was also used to explicitly model the user interface behaviour of the developed editor. We have used the power window case study in [3][9], as an example to demonstrate the semantics of PM given in SCCD. In the case study, we also included an activity in the PM which is responsible for interacting with an external service. This service is an optimal reachability analysis of a Petri-net model not available in the FTG. In addition, the external service tool, i.e. LoLA [13], is hosted in a different environment than the Modelverse, and a virtual network was used for connection and data exchange. While interacting, the PM activity, which is expressed in an explicitly modelled SCCD, autonomously handles its interactive behaviour, including lack of service response and/or connection timeouts.

6.1 Future Works

This thesis research can further be extended in the following manner:

- One of the common design patterns in both languages is hierarchical composition, which was not covered in the current research. The composition of hierarchy in PM enables for a given independent process to be encapsulated in another process. This can be used to modularize multiple process models and have them composed together for a common particular motive.
- Expressing the semantics of PM using SCCD in the current research provided an actual concurrent simulation of activities in a parallel region of PM. This means that SCCD instances, used to express activities in a parallel region of the corresponding PM, are created and executed concurrently. However, the Modelverse does not parallelize multiple model operations in which the activities represent. Parallelizing model operations can allow for fully collaborative usage of the Modelverse.
- In this research, PM instance can be verified for their conformance to the meta-model. However, process models which do not conform to the soundness property of workflow-nets discussed in chapter 3, can result in anomalies during execution, such as improper termination. A through verification of process models using the soundness property of workflow-nets can essentially prevent undesired behaviour of PM during simulation.

Bibliography

- [1] Behzad Bordbar, Luisa Giacomini, and D Holding. Uml and petri nets for design and analysis of distributed systems. *In IEEE Conference on Control Applications*, pages 610–615, 2000.
- [2] Juan De Lara and Esther Guerra. Deep meta-modelling with metadepth. *In In International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 1–20, 2010.
- [3] Joachim Denil. *Design, Verification and Deployment of Software Intensive Systems*. Universiteit Antwerpen, Faculteit Wetenschappen, Departement Wiskunde-Informatica, 2013.
- [4] Python Software Foundation. Graphical user interfaces with Tk - documentation. <https://docs.python.org/2/library/tk.html>. Accessed: 2017-03-8.
- [5] Vangheluwe Hans, Juan de Lara, and Pieter J. Mosterman. An introduction to multi-paradigm modelling and simulation. pages 9–20.
- [6] David Harel. Statecharts : a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.
- [7] John Hopcroft and Jeffrey Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.
- [8] Anneke Kleppe. The field of software language engineering. *Software Language Engineering Lecture Notes in Computer Science*, pages 1–7, 2009.
- [9] Levi Lucio, Sadaf Mustafiz, Joachim Denil, Hans Vangheluwe, and Maris Jukss. Ftg+pm: An integrated framework for investigating model transformation chains. *Lecture Notes in Computer Science SDL 2013: Model-Driven Dependability Engineering*, pages 182–202, 2013.
- [10] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.
- [11] OMG. UML 2.0 superstructure specification. Technical report, Object Management Group, 2005.
- [12] Grzegorz Rozenberg. *Handbook of graph grammars and computing by graph transformation*, volume 1. World Scientific Publishing Co., Inc, 1997.
- [13] Karsten Schmidt. Lola a low level analyser. *In IEEE Conference on Control Applications*, pages 465–474, 2000.
- [14] Harald Strle, Jan Hendrik Hausmann, and Universitt Paderborn. Towards a formal semantics of uml 2.0 activities. *In Proceedings German Software Engineering Conference, volume P-64 of LNI*, pages 117–128, 2005.
- [15] Eugene Syriani, Jeff Gray, and Hans Vangheluwe. Modeling a model transformation language. *In Domain Engineering*, pages 211–237, 2013.
- [16] Eugene Syriani, Hans Vangheluwe, Raphael Mannadiar, Conner Hansen, Van Mierlo, and Huseyin Ergin. Atompm: A web-based modeling environment, 2013.
- [17] W. M. P. van der Aalst, K. M. Van Hee, A. H. M. Ter Hofstede, N. Sidorova, H. M. W. Verbeek, M. Voorhoeve, and M. T. Wynn. Soundness of workflow nets: Classification, decidability, and analysis. *Formal Aspects of Computing*, 23(3):333–363, 2011.
- [18] Simon Van Mierlo. SCCD documentation. <https://msdl.uantwerpen.be/documentation/SCCD/>. Accessed: 2017-02-16.
- [19] Simon Van Mierlo, Yentl Van Tendeloo, Bart Meyers, Joeri Exelmans, and Hans Vangheluwe. SCCD: SCXML extended with class diagrams. *In 3rd Workshop on Engineering Interactive Systems with SCXML, part of EICS 2016*, 2016.

- [20] Simon Van Mierlo, Yentl Van Tendeloo, Bart Meyers, and Hans Vangheluwe. Domain-specific modelling for humancomputer interaction. *HumanComputer Interaction Series The Handbook of Formal Methods in Human-Computer Interaction*, pages 435–463, 2017.
- [21] Yentl Van Tendeloo. Modelverse’s documentation. <https://msdl.uantwerpen.be/documentation/modelverse/>. Accessed: 2017-04-29.
- [22] Yentl Van Tendeloo, Bruno Barroca, Simon Van Mierlo, and Hans Vangheluwe. Modelverse specification. 2016.
- [23] Yentl Van Tendeloo and Hans Vangheluwe. The modelverse: a tool for multi-paradigm modelling and simulation. In *Proceedings of the 2017 Winter Simulation Conference*, 2017.