The Design and Implementation of a Query Platform and Simulation Tool for the Analysis of UML State Machines through Declarative Modeling

Zohreh Mehrafrooz Mayvan

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of

Master of Computer Science (Computer Science) at

Concordia University

Montréal, Québec, Canada

August 2023

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Zohreh Mehrafrooz Mayvan**

Entitled: **The Design and Implementation of a Query Platform and Simulation Tool for the Analysis of UML State Machines through Declarative Modeling**

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science (Computer Science)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

_____ Chair
Dr. Sabine Bergler

_____ Examiner
Dr. Sabine Bergler

_____ Examiner
Dr. Hovhannes Harutyunyan

_____ Supervisor
Dr. Constantinos Constantinides

Approved by _____
Joey Paquet, Chair
Department of Computer Science and Software Engineering

_____ 2023

_____
Mourad Debbabi, Dean
Faculty of Engineering and Computer Science

Abstract

The Design and Implementation of a Query Platform and Simulation Tool for the Analysis of UML State Machines through Declarative Modeling

Zohreh Mehrafrooz Mayvan

Among the various aspects of the UML, a state machine is part of the specification used to model the dynamic behavior of systems. In developing complex systems, state machines can be deployed to capture use cases and thus contribute towards requirements validation. During testing, a state machine can contribute towards requirements verification. In our proposal, we treat a state machine as a directed mathematical graph and transform it into a declarative model that is implemented as a database of clauses using Prolog. To tackle the complexity of composite states, we propose an algorithm for flattening the representation of a state machine. This model transformation occurs behind the scenes and provides the same semantic model at a lower level of abstraction. The initial and flattened declarative models provide the factbase on which we build a set of rules to study the behavior, the complexity and the structure of a state machine.

Furthermore, we treat the machine's flattened model as a platform over which we simulate the machine's behavior given a scenario. We support the simulation process with a tool that we developed. The tool is implemented in Java using the Java Prolog Library (JPL) that provides an interface between the two technologies. Our simulator reads in a scenario and proceeds to generate the machine's behavior including its state at discrete time steps as output. We demonstrate the process through a case study.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction to the research and expected benefits

The Unified Modeling Language (UML) is an industrial de facto standard that supports the modeling of software processes and artifacts. Among the various aspects of the UML, a state machine is part of the specification used to model the dynamic behavior of components, ranging from the very small (e.g. a single complex object) to the very large (e.g. a use case).

In this thesis we focus on the deployment of state machines used to model use cases, thus providing a detailed description of how the entire system under development is expected to behave given a certain scenario of family of related success and failure scenarios. An overview of top-level UML activity diagram is presented in figure 1.1. Our goal is to transform the state machine into a representation that can be automatically examined and queried. More specifically, once a use case is transformed into a state machine, we want our model and its query system to examine both its observable behavior as well as its quality attributes. These two aspects of the study roughly correspond to the machine's (and by extension the use case's) functional and non-functional requirements.

On the former, we want to be able to see, for example, whether and under what conditions the machine can terminate, or the conditions under which the machine can reach a certain state etc. On the latter, we view the machine as a directed mathematical graph, and so by examining properties as well as by applying measures on the graph, we expect to be able to draw conclusions on the machine's quality attributes.

We will map the state machine (its mathematical model and its visual counterpart,

the state transition diagram as supported by the UML) into a declarative model and thus represent the machine as a database of clauses (facts). We will extend the database with a set of rules. The expected benefit from this approach is to assist developers during requirements validation to use rules and execute queries in order to be able eventually to answer the key question of requirements validation: *"Are we building the right product?"*.

We will extend our tool to allow developers define a scenario which can then serve as input to the machine through simulating its behavior. The simulation will demonstrate how a machine behaves under a given sequence of events that the system receives by producing an output composed of the reaction of the system to these events in terms of a) modification of system state (the set of global variables held by the machine) and b) execution of actions expected. In the case of the latter, we are also examining whether a system action upholds its associated invariant at two levels: First, at the level of its current state, and second, at the top-most level of the machine. The expected benefit from this approach is to assist developers run various what-if scenarios and examine the behavior of their system under development.

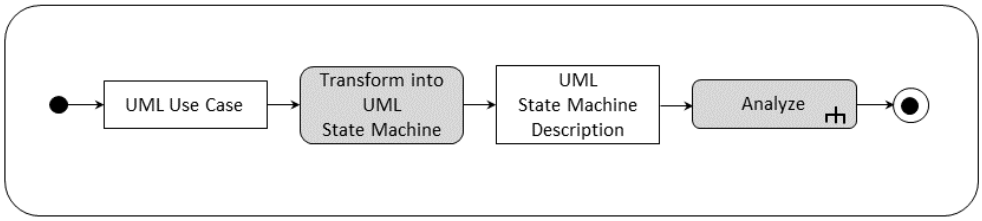

Figure 1.1: Overview of top-level UML activity diagram.

1.2 Overview of the approach

We provide an overview of our approach in figure 1.2. Our system produces an initial and a flattened declarative representation of a state machine which form the factbase for a query system. Furthermore, the flattened model provides the factbase for our simulator. The simulator additionally reads data from an imperative model produced earlier and walks over a scenario in order to produce an output in two parts: a) the behavior of the state machine as a sequence of discrete timed events and b) a snapshot of its state along a discrete time. The output can be visualized for easier analysis.

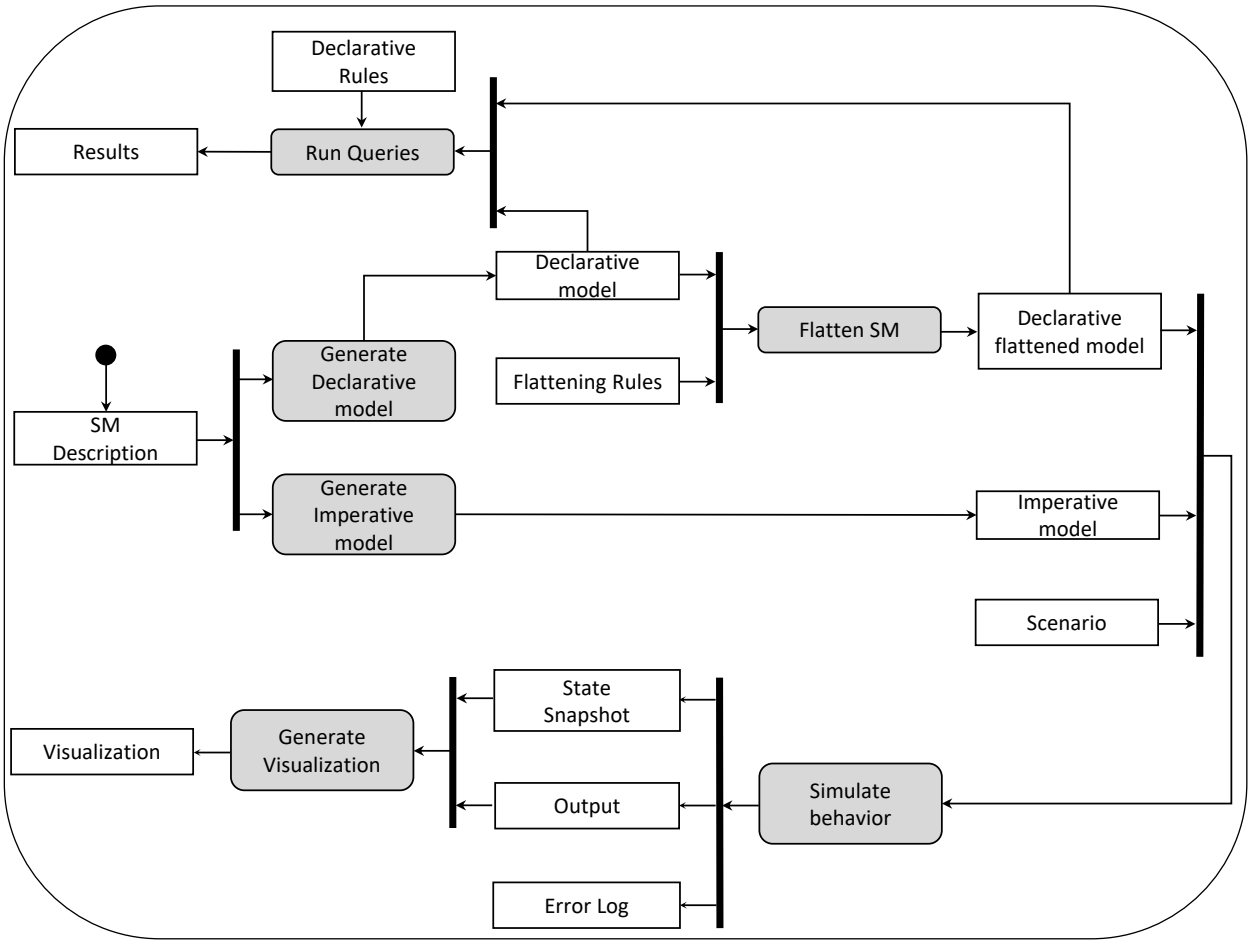

Figure 1.2: UML activity diagram of our approach.

1.3 Our contributions

In this study, we define a declarative representation of a state machine, and construct a platform to analyze the machine through queries and simulation. Our simulator is developed as a imperative model. Deploying both declarative and imperative models to study and analyze state machines is a novelty in the literature. The objective is to assist in the validation of system requirements captured by the machine and the methodology entails the study of quality attributes, behavior and well-formedness of the machine as well as simulation.

Deploying a declarative representation is one of our main contributions in this thesis. Declarative modeling is a powerful and intuitive way to represent state machines, offering numerous advantages in terms of maintainability, scalability, and analysis. In fact, declarative representation expresses the behavior and transitions of the state machine

using logical clauses and rules. This model can be implemented in Prolog, which provides capabilities like pattern matching and backtracking, making it well-suited for modeling complex behavior in state machines [1].

We propose an algorithm to flatten the declarative model in Prolog, introducing ϵ-transition and supporting various event and action types through related clauses for events and actions in the initial declarative model. Our flattened declarative model gives the flexibility to incorporate complex UML features and defines the correct sequence of actions in transitions.

The key contribution of this thesis is developing a tool to simulate the behavior of the system and generate outputs that aid in requirements validation and verification, as well as analyzing the behavior of the system. Furthermore, this research simulates the state machine which is deployed to capture use cases and contribute towards requirements validation. In fact, state machines can be used as a tool to validate the requirements against real-world scenarios and ensure that all necessary behaviors and conditions are accounted for. Our tool can help in ensuring that the system requirements are complete, consistent, and accurately captured by modeling different use cases.

Additionally, during testing, state machines can contribute towards requirements verification. This means that the state machine model can be used as a reference to verify whether the system implementation meets the specified requirements or not. By comparing the actual system behavior against the expected behavior defined by the state machine, it is possible to verify if the system behaves as intended and fulfills the specified requirements.

Last but not least, we address contract considerations including state invariants and pre/post-conditions of actions in our declarative model, and then we can assert them during the simulation.

1.4 Organization of the rest of the thesis

This thesis is structured as follows: In Chapter 2, we present an overview of the mathematical specification of state machines. A review on related works is presented in Chapter 3. In Chapter 4, we introduce our case study, and in Chapter 5, we describe the framework that provides a model transformation of a state machine into an initial declarative representation. Also, in this chapter, an algorithm is introduced to flatten the

initial declarative representation of state machines into a flattened model at a lower level of abstraction. In Chapter 6, we deploy our two declarative models (initial and flattened) as factbases in order to build a query system, by defining rules that can study the various aspects of a state machine. Moreover, In this research, we considered contracts as state invariants and pre/post-coditions of actions, which is presented in Chapter 7. In Chapter 8, we describe our developed tool for simulating the behavior of state machines, including the architecture of it. In Chapter 9, we demonstrate result of our approach on the case study.

Chapter 2

Background

2.1 From Harel's statecharts to UML state machines

Originally introduced by Gill (1962) [2] and later proposed by Harel in 1984 [3] as a significant extension over traditional (deterministic) finite state machines, a statechart is a formalism to model the dynamic behavior of a component at any level of abstraction like e.g. an object, a system unit, a use case, or the entire system itself. The Unified Modeling Language adopted Harel's statecharts in its specification and extended them (see Figure 2.1). A *state transition diagram* is the visual counterpart of a state machine. This study is on the extended statechart model which is part of the OMG UML specification[1], referred to in the literature as *UML state machine* (or *UML statechart*) and referred to throughout the thesis simply as *state machine*.

UML 2.5.1[4] provides numerous complex features, such as composite and nested states; entry and exit pseudostates; entry, exit, and do state behaviours; implicit region completion transition. This leads into a complex behaviorial analysis. We simplify the machine by converting it into a modified EFSM, as specified in the subsequent section. Moreover, the standard UML does not allow ϵ-transitions. An ϵ-transition is a transition whose *event* and *guard* are empty. It is important to recognize that empty transitions are only permissible in pseudostates, such as entry and exit points, as well as during region completion, such as when a do action is completed or when reaching a final substate.

[1] https://www.omg.org/spec/UML/2.5.1/PDF

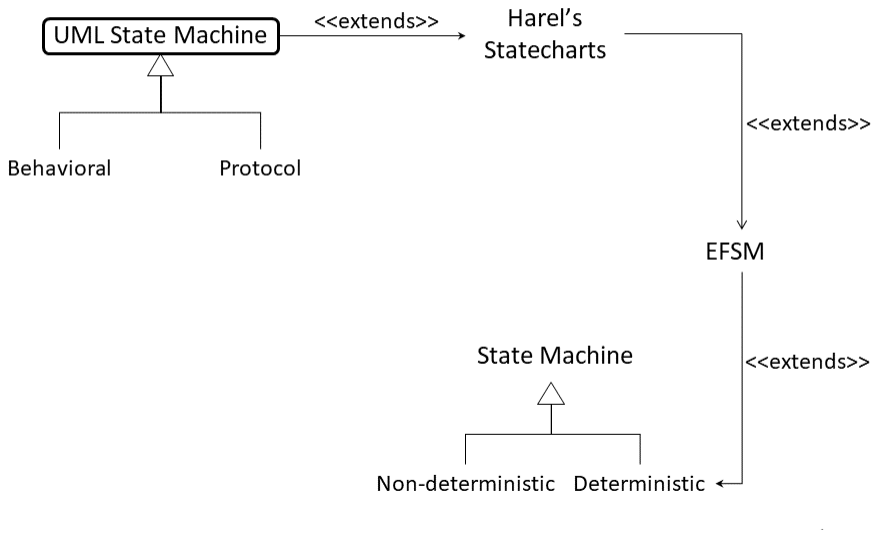

Figure 2.1: A family tree of state machines and the UML state machine.

2.1.1 Modified EFSM

The EFSM is formally defined using a 7-tuple [5]. Our definition of EFSMs uses 7-tuple as well, with a slight modification on the inputs of the transition (see Figure 2.2). An EFSM M is defined using a 7-tuple $(Q, \Sigma_1, \Sigma_2, q_0, V, \Gamma, \Lambda)$, where

- Q is a finite set of *states*,
- $\Sigma_1 = \{e_i\}$, is a non-empty finite set of *events*,
- $\Sigma_2 = \{a_i\}$, is a finite set of *actions*,
- $q_0 \in Q$ is the *starting state*,
- $V = \{v_i\}$ is a finite set of *mutable* global *variables*,
- $\Gamma = \{g_i\}$ is a finite set of *guards*,
- $\Lambda = \{\lambda_i : q \xrightarrow{e[g]/a} q'\}$, is a finite set of *deterministic* transitions defined on $Q \times \mathring{\Sigma}_1 \times \mathring{\Gamma} \rightarrow Q \times \mathring{\Sigma}_2$, where $\mathring{\Sigma}_1 = \{\epsilon\} \cup \Sigma_1$, $\mathring{\Gamma} = \{\epsilon\} \cup \Gamma$, $\mathring{\Sigma}_2 = \{\epsilon\} \cup \Sigma_2$, ϵ denotes *null*, $q, q' \in Q$, and $e \in \mathring{\Sigma}_1$, $g \in \mathring{\Gamma}$, and $a \in \mathring{\Sigma}_2$ are all *bindable* string literals.

A *bindable* expression is a well-formed expression in string literal format that consists of literal values and keys. "`tCurrent >= tThreshold`" and "`echo('Configuring mode');`" are examples of bindable string for a guard and an actions, respectively. The guard uses two

variables: "`tCurrent`" and "`tThreshold`", and the action represents a method invocation "`echo()`". A guarded ϵ-transition is represented by $\lambda : q \xrightarrow{e[g]/a} q'$ where $e = \epsilon$. In case $g = \epsilon$, the transition is referred to as ϵ-transition. In order for Λ to be deterministic, for every state $q \in Q$, at most one possible transition must exist. In other words, $\forall q \forall \lambda_i : q \xrightarrow{e_i[g_i]/a} q'$, the satisfiability of (e_i, g_i) must be exclusive. While this property holds for all EFSMs, we enforce the following restrictions:

1. If state q has an outgoing ϵ-transitions, no other outgoing transitions are allowed on the state q.

2. If state q has an outgoing guarded ϵ-transitions, only other guarded ϵ-transitions are allowed on the state. Let $\{g_i\}$ be the set of all guards for all guarded ϵ-transitions on state q. i) $\cup g_i = \mathbf{True}$; ii) $\forall i \forall j \neq i \ (\neg(g_i \wedge g_j))$.

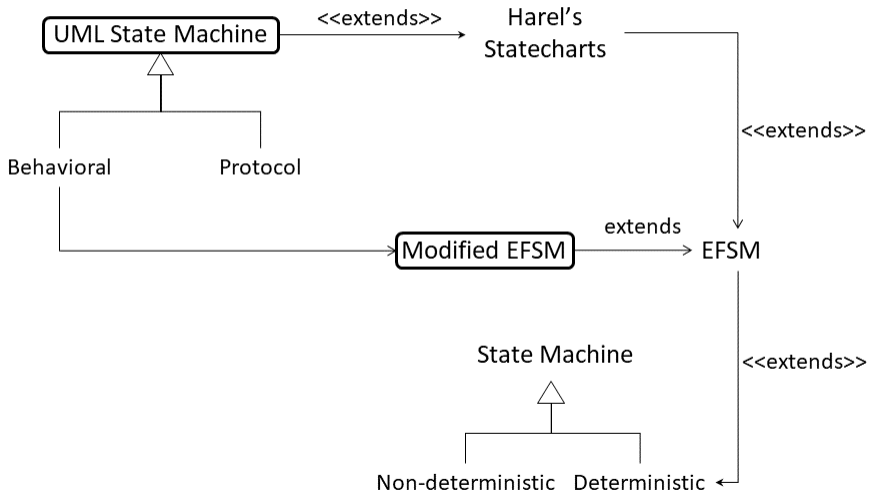

Figure 2.2: our modified EFSM.

Chapter 3

Related work

Statecharts are a widely-used notation for representing the executable behavior of complex reactive event-based systems, and they are a part of the UML standard. They are commonly used in industry for real-time systems and embedded systems development, and they are supported by various commercial tools, such as IBM Rational Rhapsody, The Mathworks Stateflow, itemis Yakindu Statechart Tools, IAR Systems visualSTATE, and QuantumLeaps QM [6]. These tools typically offer features like statechart visualization, modification, and simulation, as well as code generation from statechart models. The more advanced tools may also provide support for model debugging and verification [6]. Considering this wide varaiety of statecharts features and their applications, we go through its related works in this chapter.

Using Prolog for defining declarative representation of state machines provides a simple and concise way to represent the various elements of a state machine such as states and transitions. State machines can have complex and branching behavior, which may make it challenging to verify that all possible paths have been tested. Prolog's capabilities like pattern matching and backtracking make it well-suited for modeling the behavior of complex systems [1].

Declarative model can be extended by introducing some Prolog rules to study the behavior, complexity, and design of the underlying state machine. Using Prolog facts and rules for modeling UML diagrams has been considered in many researches. Sheng et al. [7] present a Prolog-based consistency checking for UML class diagram and object diagram. They formalize the elements of a model and then convert the model into Prolog facts.

Consistency rules are also defined in Prolog, along with interfaces that enable querying of properties, elements, and submodels of the model. Khai et al. [8] propose an Prolog-based approach for consistency checking of class and sequence diagrams. Consistency checking rules as well as UML models are represented in Prolog. The reasoning engine of Prolog is then utilized to automatically identify any inconsistencies in the models.

State machines are widely utilized in the field of software testing, which involves evaluating an application's performance and quality to verify that it meets the requirements specified prior to its development. Hashim and Dawood [9] conducted a review of test case generation methods that use UML statecharts. They found that most of the reviewed papers converted the UML statechart diagram into an intermediate model, such as a graph or a table, which was then used to generate test cases using different algorithms. Chen and Lin [10] emphasize the growing interest in software testing methods which utilize UML models. Their test case generation strategy improves the test efficiency and guarantees high test coverage and accuracy.

Aktaş and Ovatman [11] introduce UML statechart anti-patterns which may occur during software development process. These problematic design practices have the potential to cause adverse effects throughout the software development life cycle. The anti-patterns they considered are cross-level transitions, missing events, generic state names, unreachable states, cascaded conditions, isolated states, and complex statechart diagrams.

Using declarative model, static behaviour of system can be studied, and requirements can be tested. However, statecharts are a widely-used notation for representing the dynamic and executable behavior of complex reactive event-based systems [6]. This highlights the significance of having tools for visualizing, modifying, and simulating statechart models. Mens et al. [12] introduce a technique to improve statechart design. The proposed method comes with specialized tools, including a modular Python library called Sismic [6]. It is an open-source Python library for statechart interpretation that supports various techniques such as test-driven development, behavior-driven development, design by contract, and property statecharts to facilitate the testing and validation process.

Van Mierlo and Vangheluwe [13] present an approach for modeling, simulating, testing, and deploying statecharts. In their approach, they utilize Yakindu to model their case study. The state of the system can be altered when a transition is triggered by an event or timeout, as well as a condition related to system's variable values. Executing a transition

leads to the execution of an action that can modify the system's variable values or raise an event.

Balasubramanian et al. [14] introduce Polyglot, a comprehensive framework for analyzing models described using multiple statechart formalisms. Their approach involves translating statechart models into Java and analyzing them using pluggable semantics for different variants. The translation process captures the structure of the statechart model, while behavior is defined in separate Java modules. They also provide an implementation of their framework and present a case study where interacting components are modeled using different statechart formalisms.

Modeling state machines with nested composite states and flattening the model is a challenging task. One major issue in this context is the potential occurrence of unwanted non-determinism [13]. This scenario can arise when a state in a state machine has an outgoing transition that is triggered by the same event as an outgoing transition defined on one of its parent states, resulting in non-determinism. For addressing this problem, two potential solutions have been proposed in the literature: either the outermost transition can be chosen or the innermost transition can be chosen [15, 13, 16].

E. V. and Samuel [17] describe a technique to transform hierarchical, concurrent, and history states into Java code using a design pattern-based methodology. They provide an approach to implementing composite states with parallel regions in an object-oriented manner. They demonstrate that their approach generates less complex code and produces promising outcomes when compared to alternative tools.

A wide variety of testing techniques and associated tools is available for developing source code in programming languages. These techniques include test-driven development (TDD), behaviour-driven development (BDD), and design by contract (DbC) [6]. Mens et al. [18] emphassize the point that it is poorly understood how such techniques can be used for testing and validating executable statechart models. Indeed, designing statecharts and their interaction with the environment can be quite complex and error-prone, partly because of the statechart formalism itself, and partly because of the complex behaviour that these statecharts are modelling. They suggest a solution to tackle this issue by introducing a technique to improve statechart design. Their proposed method incorporates a statechart interpreter and additional libraries that facilitate the testing and validation of executable statecharts. These techniques, such as TDD, BDD, DbC, and property statecharts,

help monitor behavioral properties during statechart execution and detect any violations. Moreover, they mention that several companies are using their tool successfully, particularly for model-in-the-loop testing and simulations, as well as for supporting workflows and business processes. Sismic is also being used for executing and validating concurrent distributed statecharts.

Software testing is the process of evaluating the quality and performance of an application to ensure that it satisfies the requirements established prior to development. Test case generation is a critical phase of software testing. As the size and complexity of a software system grow, generating efficient and effective test cases becomes more challenging. Jin and Lano [19] conducted a systematic literature review on the topic of Generating Test Cases from UML Diagrams. In contrast to source code level testing, design level testing can begin testing before implementation. Therefore, generating test cases from the design level is an important area of research. UML diagrams can represent various aspects of a system, including structural and behavioral components. Despite being incomplete and ambiguous at times, UML diagrams provide crucial information for test case design and serve as guidance for automatic test case creation. Therefore, utilizing UML to design test cases is a significant and challenging topic in Model-Based Testing.

They presents the findings of a comprehensive systematic literature review focused on generating test cases using UML diagrams. After applying selection and exclusion criteria, 62 primary studies were chosen from a pool of 443 publications that spanned from 1999 to 2019. The review primarily examined the model type, intermediate format, and coverage criteria. Based on the review points, five research questions were formulated, and five assessment questions were used to evaluate the quality of the selected primary studies. Of the 22 studies reviewed, State diagram for UML model was used in 22 of them [19].

Hashim and Dawood [9] conducted a review of test case generation methods that use UML statecharts. This study utilized content analysis to examine 24 primary studies in this domain. The review focused on various aspects, including the input model used, the method or algorithm applied, any intermediate model, coverage criteria achieved, and the mode of evaluation performed on the proposed method. They concluded that the majority of the reviewed papers converted the UML statechart diagram into another representation, such as a graph or a table (intermediate model), which were later used to derive test cases by applying various algorithms, such as GA, DFS, BFS, or other customized algorithms.

They also noted that there is still room for further research in this domain, as the existing work still has claims to reach an optimal way in their test case generation approach.

Murthy et al [20] proposed a new approach for UML Statechart based test generation. They introduced Test Ready UML Statechart models that can be used by testers during the testing phases just as the conventional UML Statecharts are used during the design and development phases. They provided a detailed description of the Test Ready Statechart models, including the modifications required for the statechart semantics, the guidelines for creating Test Ready Statechart models, and a mapping algorithm for generating test cases from the Test Ready models. They also compared their approach with other UML Statechart based testing methods and provided experimental results to show the effectiveness of their approach.

Test Ready UML Statechart models are designed to provide the necessary information for a test generator to generate test scripts automatically. Test scenarios are created as instances of paths in the model, and the event generation capability captures different variations or characteristics of events. This approach aims to enable testers to generate test cases from the UML statechart models in a more efficient and effective way, reducing the time and effort required for test case generation. The Test Ready UML Statechart model is intended to be used during the testing phase, just as conventional UML statecharts are used during the design and development phases [20].

Test Ready UML Statechart model provides the necessary information for a test generator to automatically generate test scripts, which could potentially save a lot of time and effort in the testing phase. This methodology has been effective in modeling commercial applications and GUI systems. They mentions that their methodology evolves in the future, especially it can be applied to state diagrams with nested states and concurrency [20].

Use of pre- and post-contracts for each state is considered by Decan and Mens [6]. In fact, Sismic's statechart interpreter has a useful capability which is automatically checking for contract violations and unwanted behaviors during runtime. Sismic's statechart interpreter performs checks at various stages: preconditions are verified before entering a state and before processing a transition, postconditions are verified after exiting a state and after processing a transition, and invariants are checked for active states at the end of each macrostep. Transition invariants serve as both preconditions and postconditions. In our research, we focus on checking invariants for every state as well as pre- and post-conditions

of each action done. a detailed description of our approach regarding contract considerations in state machines is provided in Chapter 7 of this research.

Mens et al.[12] propose a comprehensive approach to enhance statechart design by incorporating contracts, preconditions, and postconditions. Contracts are used to define the expected behavior and constraints of the statechart model, while preconditions and postconditions specify the conditions that must be met before and after the execution of states and transitions. By monitoring contracts and properties at runtime using the Sismic tool, violations can be detected, ensuring the quality and reliability of the statechart models. This method provides valuable techniques for testing and validating statechart designs, addressing the complexities and potential errors associated with reactive event-based systems.

Sekerinski[21] proposes a method for statically verifying the design expressed by a statechart, aiming to increase confidence in the design and complement validation through testing. The approach involves supplementing a statechart with state invariants, which are conditions attached to individual states that specify what must hold in a state configuration. These invariants are not meant for execution but allow for consistency checking beyond structural well-formedness. The paper presents an algorithm that generates "local" verification conditions based on the locality of state invariants. By decomposing potentially large invariants into smaller parts, which are in visual proximity to affected transitions, complex invariants become more comprehensible. The approach offers a way to specify correctness conditions for statecharts and makes verification easier.

Chapter 4

Introducing the case study: alarm system

We define an alarm system that monitors the temperature as a case study in this thesis. The alarm system is presented in Fig. 4.1. This case study is designed in a way that it comprehensively covers as many features of UML state machines as possible. Initially the system would be *idle*, and from *idle* it can be activated, or shut off. If activated, it goes to *active* which is a composite state. Once activated, the system enters a *configuring* mode. In *configuring*, there is an internal transition for setting the temperature threshold (tThreshold) by which tThreshold can be set as many times as the user wants. After setting tThreshold, provided that its value is in the valid range, system goes to *reading*. Otherwise, another internal transition in *configuring* may trigger. It is assumed that there is a possibility for the user to skip configuring (if tThreshold is not null) and enter a *reading* state. Under certain conditions, the system can go to *emergency* as a nested composite state. State *emergency* has nested final state by which the *emergency* region will conclude and the system goes to *reading*. The UML state machine features covered in this case study are summarized in Table 4.1.

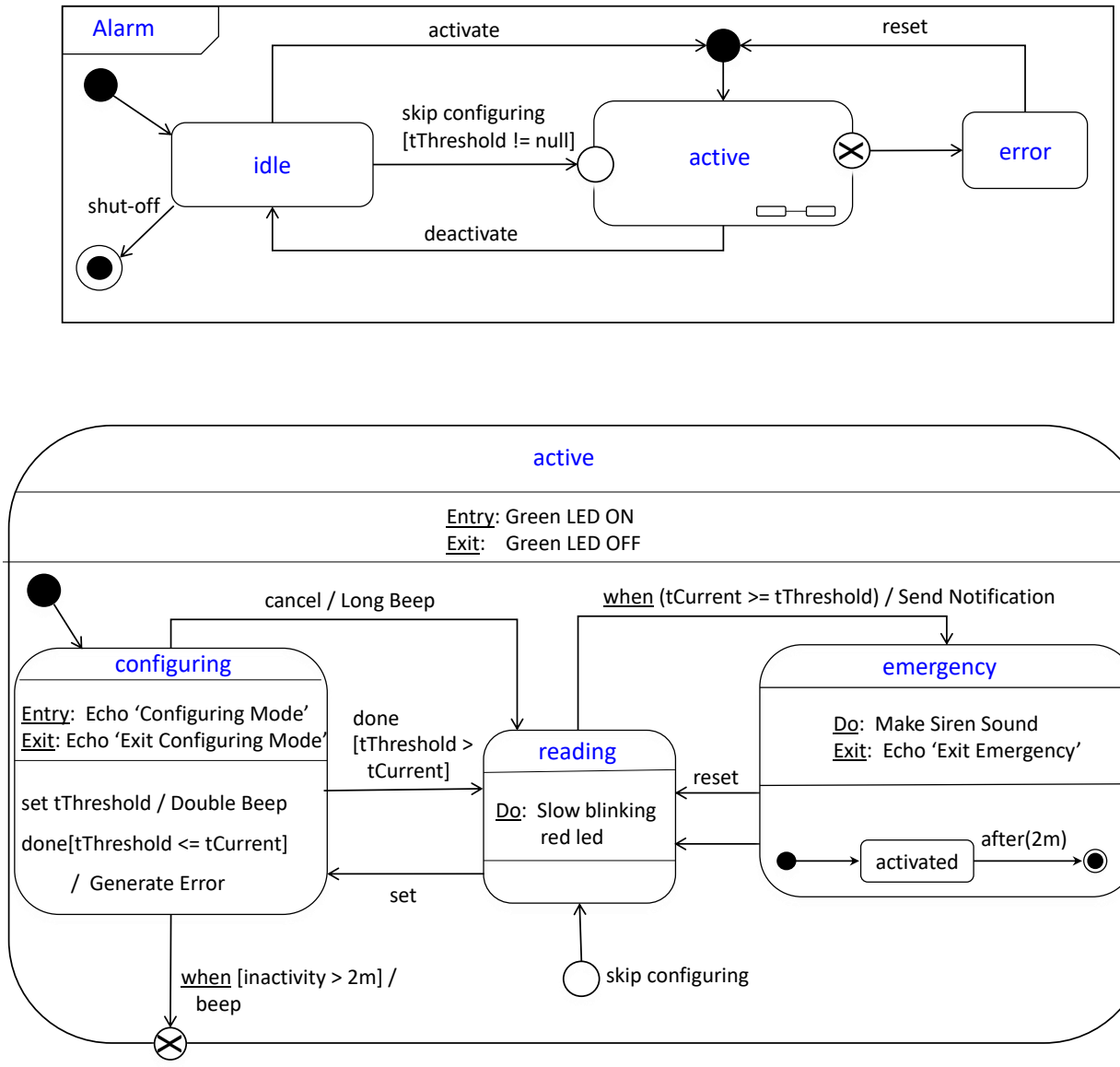

Figure 4.1: Case study: alarm.

Table 4.1: Case study coverage.

UML Feature	Coverage in case study
composite state	*active*, *emergency* (nested)
entry behaviour	considered for both simple and composite states in *active*, *configuring*
exit behaviour	considered for both simple and composite states in *active*, *configuring*, *emergency*
do behaviour	considered for both simple and composite states in *reading*, *emergency*
entry point pseudostate	"skip configuring" event in *idle*
exit point pseudostate	when "inactivity > 2m" event in *configuring*
final state (nested)	in *emergency* region
internal transition	"set tThreshold" event in *configuring*
call event	"shut-off", "activate", "deactivate", "skip configuring", "reset" in the highest level of FSM; "done", "set", "cancel", "reset" in *active*
set event	"set tThreshold" in *configuring*
time event	"after 2m" in *emergency* region
completion event	it is covered for both cases. Case 1 is completion of do behaviours in the model. Case 2 is conclusion of *emergency* region
timeout event	"inactivity > 2m" in *active* region
change event	"when [tCurrent > tThreshold]" in *active* region

Chapter 5

Model transformation to a declarative representation

5.1 Introduction

We model the entire state machine as a cyclic directed multigraph where the elements in the tuple representation of the state machine map to an ordered pair that represents the equivalent graph, namely $Graph(G) = (V, A)$, where V is a set of nodes, and A is a set of ordered pairs of nodes called (directed) edges. We map the graph's elements into a set of clauses (facts) that is maintained in a declarative model (factbase).

Various textual representations of UML diagrams may be found in the literature [22]. While UML is a graphical language, elements of a UML diagram may be represented in a textual format. We use Prolog to represent the elements of a state diagram as Prolog facts.

5.2 Why Prolog

Utilizing Prolog to establish a declarative representation of state machines offers an simple and powerful method to depict the diverse constituents of a state machine, encompassing states and transitions. State machines often encompass intricate and diverging behavior, introducing complexity in ensuring exhaustive testing of all conceivable paths. Prolog's functionalities such as pattern matching and backtracking render it particularly apt for simulating the intricate behavior of complex systems.

Furthermore, Prolog offers the valuable assets of a query engine and a query interface,

which play a pivotal role in streamlining the process of flattening a state machine. This technology enables us to seamlessly navigate the intricacies of state machines by formulating and executing queries that extract essential information about states and transitions. Additionally, Prolog's declarative nature provides the flexibility to expand the model's capabilities. By introducing custom Prolog rules, we gain the ability to delve into the study of behavioral patterns, complexity analysis, and overall design intricacies inherent in the underlying state machine. This strategic incorporation of Prolog not only facilitates our immediate goals but also lays a solid foundation for comprehensive exploration and understanding of the state machine's behavior and structure.

5.3 Declarative representation of the state machine: initial model

In transformation from state machine elements to declarative representation, composite states are defined using `superstate/2`; initial and final inner-states are represented using `initial/1` and `final/1`; entry pseudostates are represented using `entry_pseudostate/2` whose second argument is the target inner-state within the super state; exit pseudostates are represented using `exit_pseudostate/2` whose second argument is the superstate itself; entry, exit, and do state behaviours are represented using `onentry_action/2`, `onexit_action/2`, and `do_action/2`, respectively; and lastly, the internal transitions are represented using `internal_transition/4`. Note that an internal transition is similar to a reflexive transition with the exception that the entry / exit behaviors are not fired.

Table 5.1 shows all clause structures deployed to produce a model transformation from the visual representation of the UML state machine into a declarative representation.

Table 5.1: Clause signatures of the initial state machine.

FACT	DESCRIPTION
`state/1`	`state(?Name)` implies that `?Name` is a state.
`alias/2`	`state(?Name, ?Alias)` implies that `?Alias` is a new name for `?Name`.
`initial/1`	`initial_state(?Name)` implies that `?Name` is the initial state of the state machine.
`final/1`	`final(?Name)` implies that `?Name` is the exit (final) state of the state machine.
`entry_pseudostate/2`	`entry_pseudostate(?Entry, ?Substate)` implies that `?Substate` is the target inner-state whose superstate is already defined by `superstate(?Superstate, ?Substate)`.
`exit_pseudostate/2`	`exit_pseudostate(?Exit, ?Superstate)` implies that `?Exit` is an exit state within the superstate `?Superstate`.
`superstate/2`	`superstate(?Superstate, ?Substate)` implies that `?Superstate` is a composite state with `?Substate` being a nested state.
`onentry_action/2`	`onentry_action(?Name, ?Action)` implies that `?Name` defines `?Action` as an entry behavior.
`onexit_action/2`	`onexit_action(?Name, ?Action)` implies that `?Name` defines `?Action` as an exit behavior.
`do_action/2`	`do_action(?Name, ?Proc)` implies that `?Name` defines `?Proc` as a do behavior.
`transition/5`	`transition(?Source, ?Destination, ?Event, ?Guard, ?Action)` indicates that while the system is in state `?Source`, should `?Event` occur and with `?Guard` being true, the system performs a transition to state `?Destination` while performing `?Action`. All elements of the triple `(?Event, ?Guard, ?Action)` are optional, and the absence of an element is codified as `nil`.
`internal_transition/4`	`internal_transition(?State, ?Event, ?Guard, ?Action)` indicates that while the system is in `?State`, should `?Event` occur and with `?Guard` being true, the system performs `?Action`. In the triple `(?Event, ?Guard, ?Action)`, only `?Guard` is optional, the absence of which is codified as `nil`.
`event/2`	`event(?Type, ?Argument)` indicates an event where `?Type` shows event type and `?Argument` is a literal.
`action/2`	`action(?Type, ?Argument)` indicates an action where `?Type` shows action type and `?Argument` is a literal.
`proc/1`	`proc(?Procedure)` implies that `?Procedure` is a do behaviour.

Discussion - Representing transitions: In this example, a transition over a change event such as

$$reading \xrightarrow{\text{when (T\_current >= T\_threshold) / sendNotification()}} emergency$$

is codified by `transition/5` as

```
transition(
            reading,
            emergency,
            event(when, 'T_current >=T_threshold'),
             nil,
             'sendNotification();'
             ).
```

5.3.1 Modeling events

In the declarative model, we model an event as `event/2`. The UML specification includes four types of events: *call*, *signal*, *time* and *change*. We support all these types, and we introduce three more event types: *inactivity*, *update* and *completion*. We provide a description of all these types below:

call: A *call* event in a finite state machine represents an external event that triggers a state transition. We model a call event as `event(call, ?Argument)`, where `?Argument` is an external event.

signal: A *signal* event is triggered by an internal or external clock and it indicates a specific time for triggering a transition. We model a *signal* event as `event(at, ?Argument)`, where `?Argument` represents the time.

time: When the source state has been active for a specified length of time, the guard (if present) is evaluated, and a transition occurs if the guard is true. If no guard is present, a transition occurs automatically. In our model, we have this event as `event(after, ?Argument)` which make use of the `?Argument` variable to represent time that should be passed.

change: This event is triggered by a condition which is constantly evaluated. We model this event as `event(when, ?Argument)`, where `?Argument` is a condition that will trigger a transition once true.

inactivity: The system is expected to be inactive over a given amount of time. In our model, we have this event as `event(timeout, ?Argument)` which make use of the `?Argument` variable to represent time that system should be inactive.

update: Updates the value of a variable or attribute, which may subsequently trigger a transition if the new value satisfies the conditions for the transition. It makes use of keyword set as `event(set, ?Argument`, where `?Argument` is a new variable assignment.

completion: A *completion* event can occur in two cases: when a region concludes and there is an outgoing transition without any associated event, or when there is a do behavior with an outgoing transition without any event. We model a *completion* event as `event(completed, ?State)`, where `?State` is the name of the current state or region.

5.3.2 Capturing actions

We classify actions into *Exec*, and *Log*. This classification provides us with a way to manage each action type differently, allowing for greater flexibility in the model. This classification is particularly useful when we need to flatten the model, as it allows us to easily identify and apply appropriate processing to each type. In our declarative model we introduce `action/2` to codify actions.

The case study illustrates actions that are executed by the script engine (e.g. invoking the `echo()` method) as well as actions that are logged in the system (e.g. `Green LED OFF`). Note that a do action is a *process* that is *started* when the machine enters a state, and may be *stopped* (upon normal termination) or *aborted* (triggered by a final event).

5.4 Flattened representation of UML state machines

We provide a flattened representation of a state machine which can be processed by our tool support. This model transformation would occur behind the scenes, and provide the

same semantical model at a lower level of abstraction, the same way a program (written and debugged by a developer at the level of source code) is transformed into bytecode (or other intermediate representation) to be processed by a compiler and other tools. The flattened representation is also maintained in a declarative model (as a database). Some of the facts of the initial model dropped in the flattened representation since they are covered differently after processing. So, the flattened model is presented using less number of facts (see Table 5.2).

5.4.1 The flattening process

In a complex UML machine, various sequences of actions may be triggered based on the transition. Take the transition from *idle* to *active*, for example. While the transition itself does not define any action, we observe that upon triggering the *activate* event, the on-entry action of *active* state should execute before the machine goes to *configuring*. A more complex example would be the transition from *active* to *idle*. While the system is in *activated* substate, by which the following sequence of actions are executed: *aborting* `'Make Siren Sound'` process, *executing* `echo('Exit Emergency')`, and *logging* `'Green LED OFF'`.

The behavioral analysis of the UML state machine is achieved by converting the state machine into a *flattened* EFSM machine by chaining the subsequent actions using ϵ-transitions. The flattening algorithm is implemented in multiple passes and steps, that incrementally modifies the facts and eliminates the complex UML features: composite states, pseudostates, state behaviours, and internal transitions, one-by-one, resulting an equivalent machine with less complex features until the resulting machine is flattened. Finally, the resulting machine is minimized by reducing the number of states and combining equivalent transitions.

The outline of the Flattening algorithm is given in the following. The algorithm is implemented in $4 + 1$ passes. We use Prolog queries as selectors to process the input database. The output of each pass and step will be used as the input to the consequent step or phase. Pass 0 involves pre-processing that handles *completion* events as well as setting up the placeholders for *chains* of *actions*. To support chain of actions, all transition actions are converted into action lists. During the final pass, they will be converted into separate transitions. Pass 1 resolves the pseudostates and *entry* behaviours. *Do* behaviors are expanded here as well. Pass 2 performs full top-to-bottom *full state resolution*, by which

top-level states are removed and their *exit* behaviours are handled. The *exit* behaviours for non-composite states as well as the internal transitions are resolved in this pass as well. Pass 3 involves post-processing, in which a) *stop* logs for *do* processes are resolved, and b) compound actions are converted into separate transitions. The *stop* event logs are produced in two phases: book-marked in Pass 1, Step 1-V, and resolved in Pass 3, Step 2. In Pass 4, the resulting EFSM is minimized.

Procedure *flatten*

Input: The UML machine in Prolog.
Output: The EFSM machine in Prolog.

Pass 0**:** *Pre-processing*

- *Step* 1*:* Convert implied region-completion events:
  Change all `transition(S, T, nil, G, A)` where `state(S)` or `superstate(_, S)` to `transition(S, T, event(completed, S), G, A)`.
- *Step* 2*:* Convert all actions to action list:
  Rewrite all `transition(S, T, E, G, A)` where `A` $\neq$ nil as `transition(S, T, E, G, [A])`. Rewrite all `transition(S, T, E, G, nil)` as `transition(S, T, E, G, [])`.

Pass 1**:** *Processing pseudostates, entry behaviors, and do behaviors*

- *Step* 1*: Resolving do behaviors*:
  i) For
  each `do_action(S, proc(P))`, obtain `onentry_action(S, EL)`, `onexit_action(S, XL)`. Default `EL`, `XL` to `nil`, if not available.
  ii) Remove `do_action(S, _)`.
  iii) Append `action(log, "START '<P>'")` to `EL` and insert `action(log, "ABORT '<P>'")` to `XL`.
  iv) Update `onentry_action(S, EL)` and `onexit_action(S, XL)` with the new values.
  v) For every `transition(S, _, event(completed, S), _, A)`, insert `action(log, "STOP '<P>'")` to `A`.

- *Step* 2*: Resolving entry / exit pseudo states*:

  i) Replace every `entry_pseudostate(S, T)` with `transition(S, T, nil, nil, [])` and `superstate(P, S)` where `superstate(P, T)`.

  ii) Replace all `exit_pseudostate(S, P)` to `superstate(P, S)`.

- *Step* 3*: Resolving entry behaviors*:

  Starting from top to bottom, find `onentry_action(S, A)`. The iteration order may be obtained by querying state `S` in `toptobottom(L), member(S, L)` where `toptobottom/1` is defined as follows:

```
toptobottom(L) :- toplevel(TL), expand(TL, L).
toplevel(L) :- findall(S, state(S), L).
children(S, L) :- findall(C, superstate(S, C), L).
expand([S], [S]) :- not(superstate(S, _)), !.
expand([H], [H|L]) :- children(H, L2), expand(L2, L), !.
expand([H|T], [H|L]) :- children(H, CL), append(T, CL, L2), expand(L2, L).

isexternal(Sfrom, Sto) :- not(descendentof(Sto, Sfrom)).
descendentof(S, C) :- superstate(S, C), !.
descendentof(S, C) :- superstate(S, X), descendentof(X, C), !.
```

  i) Remove `onentry_action(S, A)`.

  ii) For each incoming transition from an external state `X` to `S`: `transition(X, S, _, _, L)` where `isexternal(X, S)`, append `A` to `L`.

  iii) For each incoming transition from an external state `X` to a sub-state `B` of `S`: `transition(X, B, _, _, L)` where `isexternal(X, S), descendentof(S, B)`, append `A` to `L`.

  iv) If `state(S)` and `initial(S)`, add `state(PS)`; change `initial(S)` to `initial(PS)`, and add `transition(PS, S, nil, nil, A)`.

Pass 2**:** *Full State Resolution*

- *Step* 1*: Resolving composite states*:

  For each composite state `P` where `state(P), superstate(P, _)` do the following; repeat

until no more composite state exists (including the newly created top-level states in (iii)-(c)).

i) Obtain the list of immediate sub-states of P into L: `findall(S, superstate(P, S), L)`.

   Obtain the exit behavior of the super state `onexit_action(P, EA)`, default `EA` to `nil`, if not available.

ii) Change all incoming transitions to the super-state, to the initial sub-state:

   Replace every `transition(_, P, _, _, _)` with `transition(_, I, _, _, _)` where `initial(I), superstate(P, I)`; remove `initial(I)`.

iii) For each sub-state `S` where `not(final(S))` repeat the following:

   a) Inherit all outgoing `nil`-transitions from the super-state, if the child state does not contain a `nil`-transition:

      For every `transition(P, T, E, G, A)`, add `transition(S, T, E, G, A)`, if `not(exists(transition(S, _, nil, _, _)))`.

   b) For every outgoing non-`nil` transition from the state `S` to a state that is not in `L`, including the above (a), insert `EA` to the transition action, if not `nil`:

      For all `transition(S, T, E, G, A)` where `E` $\neq$ nil and `not(member(T, L))`, insert `EA` to `A`, if `EA` $\neq$ nil.

   c) Move the sub-state to the top level:

      Replace `superstate(P, S)` with `state(S)`.

iv) Process the internal final state, if applicable (Inherit the completed event and override it as nil):

   Find `final(F)` where `superstate(P, F)`; Remove both `superstate(P, F)` and `final(F)`; Add `state(F)`; For all `transition(P, T, event(completed, P), G, A))`, add `transition(S, T, nil, G, A))` (insert `EA` to `A`, if `EA` $\neq$ nil).

v) Remove the composite state, its action, and all outgoing transitions:

   Remove `state(P)`, `onexit_action(P, _)`, `transition(F, _, _, _, _)`.

- *Step* 2*: Resolving the remaining exit behaviors*:

  For each `onexit_action(S, EA)`, find `transition(S, T, E, G, L)`; insert `EA` to `L` and replace it in the transition.

- *Step* 3*: Convert every internal transition to a regular self transition*:

  Replace `internal_transition(S, E, G, A)` with `transition(S, S, E, G, A)`.

Pass 3**:** *Post-Processing*

- *Step* 1*: Resolving STOP notifications*:

  For all `transition(S, T, G, E, AL)` where `member(action(log, "STOP '<P>'"), AL)`, remove `action(log, "STOP '<P>'")`; replace `action(log, "ABORT '<P>'")` with `action(log, "STOP '<P>'")`.

- *Step* 2*: Resolving compound actions*:

  For all `transition(S, T, E, G, [H|T])`, where `length(T, L), L > 0`, create intermediary `SI`, replace the transition with `transition(S, SI, E, G, H)` and `transition(SI, T, nil, nil, [T])`. Resolve `transition(SI, T, nil, nil, [T])`, recursively.

  Replace all `transition(S, T, E, G, [])` with `transition(S, T, E, G, nil)`.

Pass 4**:** *State Reduction / Minimization*

Merge all `nil`-transitions with common target state, guard, and action:

For each `transition(S, T, E, G, A)`:

- Find all `transition(S2, T, E, G, A)` where `S2` $\neq$ S and not(initial(S2)).
- Replace all `transition(X, S2, E2, G2, A2)` with `transition(X, S, E2, G2, A2)`.
- Remove all `state(S2)` and `transition(S2, T, E, G, A)`.

Repeat until no more transitions can merge.

Having produced a flattened model, we now produce a model transformation into a declarative representation, by deploying the clause structures shown in Table 5.2. The output of this algorithm can be found in Chapter 9 Section 9.1, which is applied to our case study.

5.4.2 An example of the order of actions in the flattened model

In this section, an example is provided to show the order of actions executed in the flattened model. Consider the state machine of our case study (Figure 5.1) when the

Table 5.2: Clause signatures of the flattened state machine.

FACT	DESCRIPTION
`state/1`	`state(?Name)` implies that `?Name` is a state.
`alias/2`	`state(?Name, ?Alias)` implies that `?Alias` is a new name for `?Name`.
`initial/1`	`initial_state(?Name)` implies that `?Name` is the initial state of the state machine.
`final/1`	`final(?Name)` implies that `?Name` is the exit (final) state of the state machine.
`transition/5`	`transition(?Source, ?Destination, ?Event, ?Guard, ?Action)` indicates that while the system is in state `?Source`, should `?Event` occur and with `?Guard` being true, the system performs a transition to state `?Destination` while performing `?Action`. All elements of the triple `(?Event, ?Guard, ?Action)` are optional, and the absence of an element is codified as `nil`.

currect state is `activated`. Upon event `deactivated`, system goes from `active` to `idle`. For this transition, we expect a sequence of actions as illusterated in Figure 5.1, considering `activated` is a nested state in `emergency` which also is a nested state of `active`. First the `do` of `emergenct` will be aborted. Next, `exit` behavior of `emergency` and `active` will be executed. Finally, the `entry` behavior of `idle` is executed.

Here is a sequence of transitions in the flattened model which will happen automatically after each other to produce this required sequence of actions which is needed as the result of transition specified in Figure 5.1 from `active` to `idle` in the state machine when current state is the nested state `actived`; while `deactivate` event occurs.

```
transition(activated, s21, event(call, deactivate), nil, action(log, "ABORT 'Make
Siren Sound'")).
transition(s21, s22, nil, nil, action(exec, "echo('Exit Emergency');")).
transition(s22, pre_idle, nil, nil, action(log, "Green LED OFF")).
transition(pre_idle, idle, nil, nil, action(log, "System Startup")).
```

5.4.3 More examples of the flattening procedure

In this section, some example of different steps in the flatten procedure is provided to elaborate more on our flatenning process.

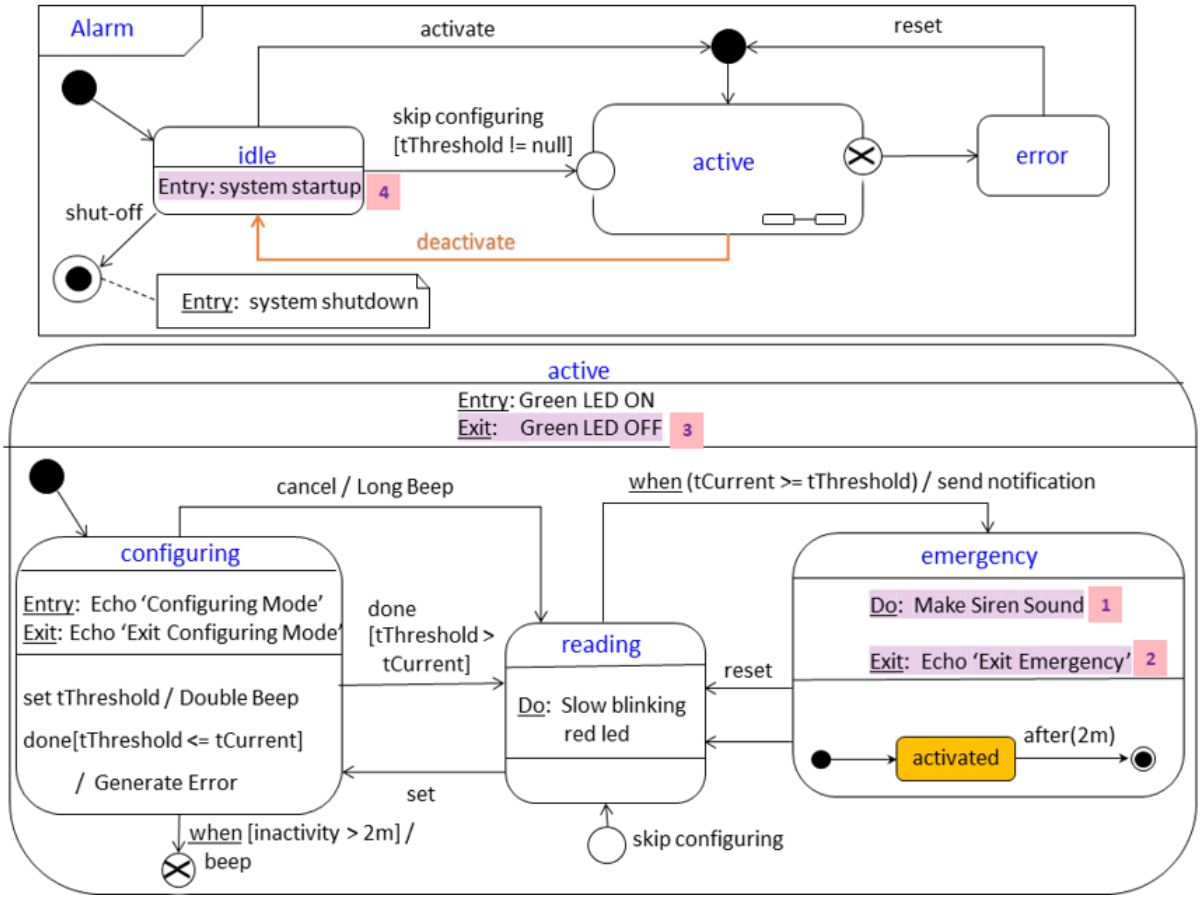

Figure 5.1: A transition and its corresponding order of actions.

An example of Pass 1 - step 1

In step 1 of Pass 1, for processing `do` behavior of `reading`, the following facts will be removed from the initial model:

```
do_action(reading, proc('Slow blinking red LED'))
onentry_action(reading, [])
onexit_action(reading, [])
```

As the result of this step, the following facts is added to the flattened model:

```
onentry_action(reading, [action(log, 'START SlowblinkingredLED')])
```

```
onexit_action(reading, [action(log, 'ABORT SlowblinkingredLED')])
```

An example of Pass 1 - step 3

In step 3 of Pass 1, for processing `entry` behavior of `active`, the following facts will be removed from the initial model:

```
onentry_action(active, action(log, 'Green LED ON'))
transition(idle, active, event(call, activate), nil, [])
transition(error, active, event(call, reset), nil, [])
transition(idle, active_skip_config_entry, event(call, 'skip configuring'),nil,[])
```

As the result of this step, the following facts is added to the flattened model:

```
transition(idle, active, event(call, activate), nil, [action(log, 'Green LED
ON')])
transition(error, active, event(call, reset), nil, [action(log, 'Green LED ON')])
transition(idle, active_skip_config_entry, event(call, 'skip configuring'), nil,
[action(log, 'Green LED ON')])
```

Chapter 6

Building a declarative query platform

6.1 Introduction

We view the state machine as a graph, where states correspond to nodes and transitions correspond to edges. Given this view, we can study properties of graphs to see how they reflect to state machines. The objective here is to identify graph properties which we expect to have them in state machines.

6.2 Executing queries on the declarative database

With the declarative model as is, we can execute simple ground queries that can give us some basic knowledge of the machine such as *"Is there a transition from state idle to state configuring?"*

```
? transition(idle, configuring, _, _, _).
Yes.
```

We can also execute non-ground queries such as *"Under what conditions, if any, would the state machine perform a transition to the emergency state?"* This would entail capturing any and all `state-event-guard` triples that can cause such a transition.

```
? transition(State, emergency, event(_,Event), Guard, _).
Event = "tCurrent >= tThreshold",
Guard = null,
State = reading
```

6.3 Extending the declarative model with rules

We can extend the declarative model by introducing rules. We can identify three types of rules: (1) Rules that reason about the *behavior* of the state machine by examining the traversal of the underlying graph under various different conditions, (2) Rules that reason about the *quality attributes* of the state machine by examining the properties and measurements of the underlying directed graph. We argue that the two types of rules roughly correspond to the state machine's functional and non-functional requirements. Finally, (3) Rules that succeed by studying characteristics of the graph that correspond to the design of the state machine.

6.3.1 Studying behavior

In this section, we define some rules to study the observable behavior of the machine, such as the exposed interface of the machine seen as a black box, as well as small what-if scenarios such as how the machine reacts given certain conditions.

Exposed interface: The *call* and *set* events correspond to messages sent to the system and they collectively constitute the exposed interface of the system. The following rule succeeds by collecting any and all such events:

```
%% get_interface/1: Succeeds by returning the exposed interface of the SM
%%                  as a collection of call and set events.
%%                  Consults:  Initial model.
get_interface(Interface) :-
   findall(Event,
           (transition(_, _, event(call, Event), _, _) ;
            transition(_, _, event(set, Event), _, _)),
            EventList),
   list_to_set(EventList, Interface).
```

Legal events at a given state: Given the system exposed interface, it is important to note that not all events can be acted upon unconditionally. An event can be accepted based on the system's current state. It will be acted upon provided the associated guard (if one is present) evaluates to true.

```
%% is_legal(?State, ?Event).
%% Succeeds if ?Event is legal under ?State.
is_legal(State, Event) :-
           transition(State, _, event(_, Event), _, _);
           internal_transition(State, event(_, Event), _, _).
```

In the case study we have the following:

```
? is_legal(reading, Event).
Event = disable
```

6.3.2 Studying complexity

We study properties and measurement of a mathematical directed graph, and provide rules for properties and measurements which are applicable for state machines. These rules will provide information about complexity of state machines. Some of these rules are as follows.

6.3.2.1 Directionality

A graph is directed (as opposed to undirected) if its edges have a direction. Related to this is the notion of mixed graph in which some edegs may be directed and some may not be directed. Directionality is present in state machines, as each transition, by definition, has a direction from a source to a target state. Additionally, a state machine cannot be viewed as a mixed graph.

6.3.2.2 Rooted

A rooted graph is one where there is a path from one node, qualified as the root, to every other node in the graph. In a state machine, we expect this property to be present as there would have to be a path from the initial state to every other state in the machine.

6.3.2.3 Connectivity

A walk is a sequence of nodes which we obtain while traversing a graph (allowing both nodes and edges to be revisited). The definition leaves space for a walk to be an infinite sequence. A walk can be open or closed. A trail (or path) is an open walk where all edges are distinct. A closed trail is a circuit (or cycle) and a simple circuit is one where that repeats only the first and last node. In the presence of cycles, a graph is called cyclic (as opposed to acyclic). A simple path is a trail where all nodes are distinct. Note that there is some discrepancy in the literature. Not all authors identify *path* with *trail*, thus distinguishing between *path* and *simple path* as shown in Table 6.1.

In the context of state machines, of interest to us would be paths and simple paths. A simple path can demonstrate reachability from a source state to a destination state. Additionally, a path from the initial state to the final state can indicate a scenario where the machine can function and conclude. In fact, the number of paths between the start and final states would indicate the number of different scenarios where the machine can achieve termination. We refer to these paths as *legal trajectories*. Furthermore, a state machine should have no restrictions on cycles as it would normally perform transitions between states.

	Repeated edge	Repeated node	Starts and ends at same node
Walk [open]	Allowed	Allowed	No
Walk [closed]	Allowed	Allowed	Yes
Trail (Path)	No	Allowed	No
Simple Path	No	No	No
Closed trail (Circuit, Cycle)	No	Allowed	Yes
Simple Circuit	No	First and last only	Yes

Table 6.1: Types of graph walks and paths in traversing a graph.

There are two distinct aspects of connectivity in a directed graph: A directed graph is (a) weakly connected if there is an undirected path between any pair of vertices, or (b) strongly connected if there is a directed path between every pair of vertices.

From the definition we see that strong connectivity is not present in state machines as there would exist at least one state (the exit state, at the top-level machine and possibly exit states in substates) from which there is no transition to any other state. On the other hand, weak connectivity would be a property of state machines as we expect that there exists no isolated state, i.e. a state with an in-degree or out-degree of zero.

6.3.2.4 Completeness

A graph is complete if there exists an edge between each pair of nodes. However, this property does not necessarily need to be present in a state machine.

6.3.2.5 Simplicity

A simple graph is one where a pair or adjacent nodes is connected by only one edge. Additionally, there is no edge that connects a node to itself. If either condition is not met, then the graph is called a multigraph. A directed graph is a multigraph if there is more than one edge between two nodes, pointing to the same direction. We expect the multighraph property to be present in a state machine as there may be more than one transition from a source to a destination state.

6.3.3 Measurements

Measurements in graphs can be global or nodal. Global measurements refer to global properties of the graph and consist of a single number for any given graph. Nodal measurements refer to properties of the nodes and consist of a number for each node for any given graph.

6.3.3.1 Order of graph

This measurement refers to the number of nodes in a graph. In the context of state machines, we believe that the initial model may not give us an accurate picture due to the presence of composite states. The flattened model would be more accurate for this measurement. For the initial model the rule is shown below:

```
%% order/1: Succeeds by returning the order of the machine
%%          Consults:  Initial model.
order(N) :-
   findall(
      State,
      (state(State); superstate(_, State)),
      StateList
     ),
   list_to_set(StateList, States),
   length(States, N).
```

For the flattened model the rule is shown below:

```
%% order/1: Succeeds by returning the order of the machine
%%          Consults:  Flattened model.
order(N):-
   findall(S,
      state(S),
      Length),
   length(Length, N).
```

6.3.3.2 Number of nil transitions

The number of *nil* transitions in a flattened model can be a measure of the complexity of a state machine. The following rule succeeds by returning the number of *nil* transitions:

```
%% order/1: Succeeds by returning the number of nil transitions
%%          Consults: Flattened model.

nil_transition(N) :-
   findall(
      Nilevents,
      (transition(_, _, Nilevents, _, _), Nilevents=nil),
      Transitions
      ),
   length(Transitions, N).
```

6.3.3.3 Size (or length) of graph

This measurement refers to the number of edges in a graph. In the context of state machines, we believe that the initial model may not give us an accurate picture due to the fact that in the presence of composite states, their nested states inherit the transitions of their superstate. The flattened model would be more accurate for this measurement.

```
%% size/1: Succeeds by returning the size of the machine
%%         Consults: Flattened model.

size(N):-
    findall(S,
    transition(S,_,_,_,_),
    Length),
    length(Length, N).
```

6.3.3.4 Degrees

The notion of degree of a node refers to the number of edges that are incident on that node. In a directed graph, this notion splits into in-degree and out-degree. A regular graph is one where each node has the same degree. Additionally, an isolated node is one with degree zero (not including self-loops).

As nested states inherit the transitions of their superstates, the out-degree of a nested state would include any and all outgoing transitions of all superstates. The nested final state is an exception here as there is only one transition by the superstate once the nested final state is reached. In the case of in-degree, the same rule applies.

In a state machine, we cannot have an isolated state. Additionally, we expect the start (initial) state to have a positive out-degree. We would also expect the exit (final) state to have (a) a positive in-degree and (b) be a sink (i.e. having a zero out-degree). A state machine need not necessarily correspond to a regular graph.

6.3.4 Studying the design of the state machine

In this section, we define rules to study the design of the state machine and find cases such as dead ends, conflicts, or inconsistencies among state machine's elements. In fact, we can study the design of the state machine considering the following issues that may arise in the design of a state machine.

- Dead ends and infinite loops.
- Internal transition without an action.
- Multiple change events originating from the same state.
- Non mutually exclusive guards originating from the same state.
- The absence of a do behavior in the presence of an external transition with no event.
- As the previous item for a composite state, with the absense of an exit substate.

Dead ends: We are interested in finding out if the machine can enter a state from which the final state is not reachable. Rule `dead_end/0` succeeds by obtaining a non-empty list of states from each of which there is no path to state `final`.

```
path(X, Y) :- path(X, Y, [X]).
path(X, Y, V) :- transition(X, Y, _, _, _), \+ member(Y, V).
path(X, Y, V) :- transition(X, Z, _, _, _),
   \+ member(Z, V), path(Z, Y, [Z|V]).


dead_end :-
   findall(
      State,
      \+path(State, final),
      L),
   L \= [].
```

Chapter 7

Contract considerations

7.1 Overview

The UML specification distinguishes between behavioral and protocol state machines, where the latter is a specialization of the former, allowing developers to introduce (a) assertions over events in transitions and (b) an invariant to each state.

We propose an extension to the behavioral model, with (a) assertions on actions and (b) the inclusion of invariant properties at two levels: (1) At a global level (i.e. an invariant property of the top-most level state machine that must be observed by all individual states) and (2) An invariant property present in a state (transitive into any and all of its nested states, if present).

7.2 Assertions on actions

In a state machine, actions can be present (a) within a state as internal behavior (Entry, Do, Exit) and (b) over a transition. For the latter, we case we have three cases: (1) A transition from one state to another, (2) A recursive transition and (3) An internal transition. An action would be associated with a pair of assertions (pre-condition and post-condition). As the theory suggests, for a triple `{P} A {Q}` where `A` denotes an action, `P` denotes the pre-condition and `Q` denotes the post-condition. This approach enriches a transition λ which is now defined as follows:

$$\lambda \in \Lambda : q \xrightarrow{\text{e [g] / \{P\} a \{Q\}}} q'$$

Note that in the absence of either assertion, one assumes that the assertion is true, which is the same assumption that we make in the absence of a guard on a transition. We must make a distinction here between the guard and the pre-condition to an action: On one hand, the guard determines the eligibility of a transition to take place upon occurrence of an event, or the completion of the region of the source state q. On the other hand, the pre-condition determines the eligibility of an action to execute while the transition is in effect.

The expected benefits from this approach are (a) to explicitly distinguish between benefits and obligations between the two parties (an actor and the system under development) and (b) to assign blame in the case of a contract failure. The triple above would read as follows: *"If the actor can ensure the pre-condition, then upon termination of the action, the system guarantees the post-condition."*

Actions are also present in scenarios. Here, a developer who prepares a scenario may intentionally want to inject a condition which should lead to failure and they would expect that the contract system will catch it while at the same time assigning proper blame.

7.2.0.1 Orthogonality of actions

In the presence of more than one action (either on a transition or as part of state behavior), we need to examine whether the order of action execution is important. In cases where the order is important, we must enforce ordering. In such a case, we refer to these actions as non-orthogonal, as opposed to orthogonal actions where their order of execution would not have any effect on the requirements.

7.2.1 Global and state invariant properties

The invariant property of the state machine should be expected to hold in all states. We call this the global invariant. In addition, each state can have its own invariant property. The global invariant is conjoined with that of all its states. The same applies to the invariant of a composite state: it is conjoined with the invariant of each of its nested states, etc.

7.3 Event and action processing

In this section, we consider cases of event and action processing for different types of transitions (recursive, internal and external) in the presence of contracts. We should note that when do_behavior is aborted, its post-condition will not be evaluated since upon termination of the action, the system guarantees the post-condition.

Scenario 1: Recursive transition: For the triple `event[guard]/action` on a recursive transition λ over a state `S`, the system exits and re-enters `S`. Thus, we have the following upon reception of `event` while `guard` is evaluated true:

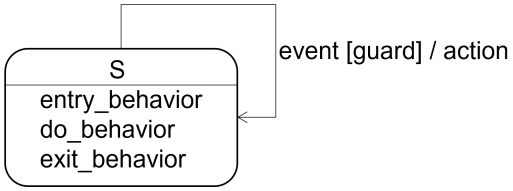

Figure 7.1: Recursive transition.

```
abort do_behavior
evaluate S.invariant + global invariant
evaluate exit_behavior.pre-condition
execute exit_behavior
evaluate exit_behavior.post-condition
evaluate global Invariant
evaluate action.pre-condition
execute action
evaluate action.post-condition
evaluate global invariant
evaluate entry_behavior.pre-condition
execute entry_behavior
evaluate entry_behavior.post-condition
evaluate S.invariant + global invariant
evaluate do_behavior.pre-condition
execute do_behavior
evaluate S.invariant + global invariant
```

Scenario 2: Internal transition: For the triple `event[guard]/action` on an internal transition λ over a state `S`, the system does not exit and re-enter state `S`. Thus, we have the following upon reception of `event`:

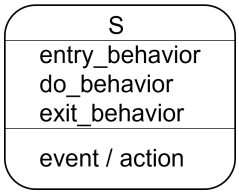

Figure 7.2: Internal transition.

```
abort do_behavior
evaluate S.invariant + global invariant
evaluate action.pre-condition
execute action
evaluate action.post-condition
evaluate S.invariant + global invariant
evaluate do_behavior.pre-condition
execute do_behavior
evaluate S.invariant + global invariant
```

Scenario 3.1: External transition in the presence of event: For the triple `event[guard]/action` on a transition λ from a source state `S` to a destination state `D`, we have the following upon reception of `event` while `guard` is evaluated true, assuming that both `S` and `D` include their own state behavior:

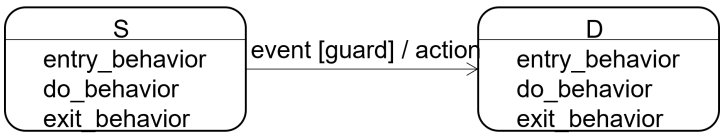

Figure 7.3: External transition in the presence of event.

```
abort S.do_behavior
evaluate S.invariant + global invariant
evaluate S.exit_behavior.pre-condition
execute S.exit_behavior
evaluate S.exit_behavior.post-condition
evaluate global invariant
evaluate action.pre-condition
execute action
evaluate action.post-condition
evaluate global invariant
evaluate D.entry_behavior.pre-condition
execute D.entry_behavior
evaluate D.entry_behavior.post-condition
evaluate D.invariant + global invariant
evaluate D.do_behavior.pre-condition
execute D.do_behavior
evaluate D.invariant + global invariant
```

Scenario 3.2: External transition with completion event: If the guard evaluates to $false$, then that would be a badly formed model. In the absence of a guard, the transition would occur immediately upon the completion of the `S.do_behavior`. In the absence of `S.do_behavior`, then this would be a badly formed model as nothing can trigger a completion event. Note that in the case of `S` being a composite state, it would be necessary to have an final substate to trigger a completion event (unless there is a do behavior). So, we have completion event either under completion of S.do_behavior or concluding the S.region in case S is a composite state. We have the following upon reception of `completion` event while `guard` is evaluated true:

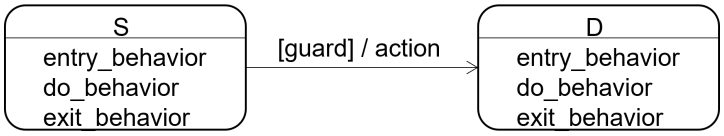

Figure 7.4: External transition with completion event.

```
evaluate S.do_vehavior.post-condition
evaluate S.invariant + global invariant
evaluate S.exit_behavior.pre-condition
execute S.exit_behavior
evaluate S.exit_behavior.post-condition
evaluate global invariant
evaluate action.pre-condition
execute action
evaluate action.post-condition
evaluate global invariant
evaluate D.entry_behavior.pre-condition
execute D.entry_behavior
evaluate D.entry_behavior.post-condition
evaluate D.invariant + global invariant
evaluate D.do_behavior.pre-condition
execute D.do_behavior
evaluate D.invariant + global invariant
```

Scenario 3.3: External transition with change event: We have the following upon `guard` becomes true which raise a change event:

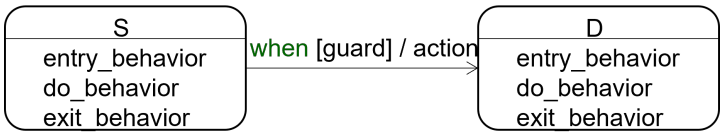

Figure 7.5: External transition with change event.

```
abort S.do_behavior
evaluate S.invariant + global invariant
evaluate S.exit_behavior.pre-condition
execute S.exit_behavior
evaluate S.exit_behavior.post-condition
evaluate global invariant
evaluate action.pre-condition
execute action
evaluate action.post-condition
evaluate global invariant
evaluate D.entry_behavior.pre-condition
execute D.entry_behavior
evaluate D.entry_behavior.post-condition
evaluate D.invariant + global invariant
evaluate D.do_behavior.pre-condition
execute D.do_behavior
evaluate D.invariant + global invariant
```

7.4 Incorporating contracts in the declarative model

As we disscussed in the overview of this section, we can cover two types of contracts including action assertions and state invariant. In this section, we present some facts to incorporate these assertions in our declarative model. These assertions can be validated later through simulation.

We use `assert/2` to define an invariant for each state as well as global invariants of the state machine. Clause `assert/2` is defined as `assert(?State, ?Invariant)` which implies `?Invariant` is an invariant for state `?State` ,and this invariants must hold true while system is in state `?State`. For example, we can define the following facts in our case study.

```
assert(globalSM, "tThreshold>= 0").
assert(reading, "tThreshold!=null").
```

```
assert(emergency, "tThreshold<=tCurrent").
```

Furthermore, for defining pre/post-conditions for actions, we can change our clause for actions from `action/2` to `action/4` in order to incorporate actions assertions. The clause `action/4` can be defined as `action(?Type, ?Name, ?Pre-condotion, ?Post-condition)` in which `?Type` shows the type of the action, `?Name` shows the action, `?Pre-condotion` and `?Post-condition` as their names suggest are pre- and post-conditions of the action. For example in our case study, we can define the following fact for the action of the internal transition of `configuring`.

```
action(exec, "generateError();", "tThreshold!=0", "tThreshold=40").
```

Chapter 8

Simulating machine behavior

8.1 Conceptual overview

The query system provides a level of analysis that is complemented with a simulation of the machine. We extend the query system we describe in the Chapter 6 in order to be able to simulate the behavior of a state machine. The main idea behind a simulation is to study the complete behavior of the machine under a sequence of events that occur in its environment. This sequence of events is captured by a scenario. At the highest level of abstraction, and given a scenario, the simulation would be performed as Read-Evaluate-Execute Cycle, described from the point of view of a simulator.

8.1.1 Initialization and persistence of state

The simulator must initialize its own state to correspond to the initial state of the machine, captured by the values of all of the machine's (global) variables. Additionally the simulator must make sure that its state is synchronized with that of the machine, i.e. any change of state in the machine should be reflected in the simulator.

The simulator needs to access the machine's state (1) in order to initialize its own state, (2) in order to perform a change of state (initiated by an action during a transition) and (3) in order to observe any change of state in the machine. Operations (1) and (3) are Read operations, where operation (2) is a Write operation. To support persistence, we realize that we need to introduce an imperative model that will hold the machine's state. Moreover, an additional component is introduced that will serve as the interface between the declarative

and imperative models.

8.1.2 Global variables

The system keeps track of its operating environment through maintaining certain variables (e.g. temperature). We refer to these as *environment variables*, denoted by V_e. Environment variables are read-only. Note that the simulator (which lies outside the boundary of the system) is free to modify these variables for the sake of simulation. On the other hand, the system maintains variables for aspects that can be fully controlled by the system (e.g. setting the temperature threshold). We refer to these as *controlled variables*, denoted by V_c. The two sets are disjoint and they collectively make up the set of all global variables that are handled by the system, i.e. $V_e \cap V_c = \emptyset$, and $V_e \cup V_c = V$.

8.1.3 Structure of a scenario

A scenario is a sequence of commands consisting of three types of tags: `EVENT`, `EXECUTE`, and `TIME`. `EVENT` tags can be of type `call`, `set`, or `completion`, and must trigger the corresponding transition in the simulator. `EXECUTE` tags contain expressions that modify variable values during simulation, and may trigger a transition. `TIME` tags can be either `after` or `at`, and update the global time variables 'duration' and 'absoluteTime' if applicable. Any time tag may trigger a timed-transitions or not. A timed-transition is defined as a transition that its event type is `after` or `at`. After any `TIME` tag, the simulator will keep time information while it proceeds to the next command in the scenario.

8.1.4 The Read-Evaluate-Execute cycle

In UML, it is assumed that a state machine processes one event at a time and finishes all the consequences of that event before processing another event [22]. At the highest level of abstraction, and given a scenario, the simulation would be performed using a Read-Evaluate-Execute Cycle. When a command in a scenario is `EVENT` e, where $e \in \Sigma_1$, given the current state and the event, the simulator would construct a `transition` query and consult the declarative model. We query the database and find all transitions $\lambda_i \in \Lambda$ with event e. The result of the query is a set of λ_i, associated with tuples $\{(q, g, a)_i\}$ where $q \in Q$ is the target state, $g \in \Lambda$ is a guard, and $a \in \Sigma_2$ is an action. Each tuple is also associated with a set of $v_i \subset V$, containing all variables used in g_i and a_i. The query is

successful only if one transition is possible. This is achieved by instantiating all variables in v_i and evaluating g_i. Upon success, a single transition is fired. The simulator consequently checks if any additional transitions can be triggered, following the most recent transition. The process continues until no further possible transition is applicable. We can show this cycle as follow:

```
(set current state to initial state)
while (scenario not exhausted)
do
   (read line)
   (transform line into declarative query)
   if (evaluate query) {
      (execute query)
      (obtain results)
   }
   else Error
end
```

Given a state-event pair, the first thing the system needs to do is event validation: Is this a legal event, given the current state? If this is indeed the case (i.e. there exists at least one transition in the declarative model with this state-event pair), then the second thing the system needs to do is to search for a transition that it can potentially perform. It is important to note that there will be at most one such transition, and we can identify the following cases:

1. Single transition, guard is absent: The transition is performed.

2. Single transition, guard is present: The guard will be evaluated and, if true, the transition will be performed.

3. Multiple transitions: We need either at most one transition where a guard is absent and all other guards are false, or at most one guard being true. Otherwise, the declarative model corresponds to a badly formed state machine. The different cases to consider are tabulated below:

EXAMPLE	WHAT TO DO
`transition(?S, ?D, ?E, nil, ?A);`	Perform a transition to `?D`
`transition(?S, ?D, ?E, nil, ?A);`	Model is badly formed.
`transition(?S, ?D, ?E, nil, ?A);`	Generate an error
`transition(?S, ?D, ?E, nil, ?A);`	`?G` is expected to be False.
`transition(?S, ?D, ?E, ?G, ?A);`	Perform a transition to `?D`
`transition(?S, ?Di, ?E, ?Gi, ?A);`	`?Gi` is True and `?Gj` is False.
`transition(?S, ?Dj, ?E, ?Gj, ?A);`	Perform a transition to `?Di`
`transition(?S, ?Di, ?E, ?Gi, ?A);`	Both `?Gi` `?Gj` are True.
`transition(?S, ?Dj, ?E, ?Gj, ?A);`	Model is badly formed. Generate an error.
`transition(?S, ?Di, ?E, ?Gi, ?A);`	None of `?Gi` `?Gj` are True.
`transition(?S, ?Dj, ?E, ?Gj, ?A);`	No transition performed.

8.2 Unveiling system behavior and process flows

In this section, we provide a clear and concise depiction of the sequential steps and decision points involved in our algorithm's execution of our tool.

8.2.1 Switch-case command processing

In our system, command processing involves handling sequences of commands that consist of three types of tags: `EVENT`, `EXECUTE`, and `TIME`. A scenario is composed of these tags, each serving a specific purpose in driving the simulation.

The `EVENT` tag, which can be of type `call`, `set`, or `completion`, plays a crucial role in triggering the corresponding transition within the simulator. When an `EVENT` tag is encountered, the switch-case paradigm is leveraged to efficiently handle the different event types and initiate the appropriate transitions.

The `EXECUTE` tags, on the other hand, contain expressions that modify variable values during the simulation. These tags provide flexibility in altering the state of variables within the system, potentially triggering transitions based on the updated values. The execution of `EXECUTE` tags ensures the efficient modification of variable values and integration with the simulation flow.

Additionally, the `TIME` tags serve to update global time variables, namely `duration` and

`absoluteTime`, if applicable. These tags can be either `after` or `at`, providing flexibility in specifying timing constraints within the scenario. Whether triggering timed-transitions or not, the system check FTransitions enabling the effective handling of `TIME` tags. Also, it ensurs accurate tracking of time-related information while progressing to the next command in the scenario.

8.2.2 FTransitions

In our system, transitions play a key role in driving the behavior and flow. After any transition occurs, there is a possibility of triggering additional transitions known as Following-Transitions (FTransitions). These FTransitions are mainly a result of the action sequence associated with the source state's onexit action, transition action, and destination state's onentry action (when a transition is fired) taking into account any inherited actions from super states. Furthermore, the state do-behavior is another reason that may trigger a FTransition.

To handle these action sequences effectively, our flattening algorithm breaks them down and assigns nil-transitions (or ϵ-transitions are transitions whose *event* and *guard* are empty) to individual actions, considering their specific order. This process ensures that each action within the sequence is properly accounted for and executed accordingly in a seperate transition.

Another triggering mechanism for FTransitions is the `when` event type (see Section 5.3.1). After each transition, the system checks for `when` events to determine if the event argument evaluates to true, thereby triggering the corresponding FTransitions if the condition is met. By considering both the action sequences and `when` events, our system dynamically determines and triggers the appropriate FTransitions, facilitating a comprehensive and flexible system behavior.

8.3 Simulator architecture

To perform a simulation, we need to provide storage of all variables (machine and environment) while keepping track of any changes. We also need to provide storage and keep track of the machine's current state. To support these requirements, we provide an imperative model. Our simulator as a imperative model is developed in Java. It uses JPL

which is a Prolog/Java interface. The simulator reads each scenario line and defines a query considering the current state of the state machine. Then it consults the declarative model (factbase in Prolog) to run the query through JPL. Furthermore, we use Javascript for defining and maintaining system variables, direct manipulation of variables in actions, evaluating events, guards, and executing actions which is done by GraalVM JavaScript engine.

The architecture of the simulator will be desceribed in terms of its static and its dynamic models which are presented respectively in section 8.3.1 and section 8.3.2 .

8.3.1 Static model

The static model focuses on capturing the structural aspects of the system architecture. It provides a comprehensive view of the system's organization, including its components, relationships, and dependencies. In this section, we employ two key UML diagrams, namely the component diagram and the class diagram, to describe and illustrate the static model of our simulator. Visually representing components, classes and their relationships through these diagrams helps us understand the overall organization and composition of the system.

8.3.1.1 The UML component diagram

A visual representation of the components that make up our simulator and how they interact with each other is presented in figure 8.1. It depicts the high-level structure and relationships between components in the system. As the component diagram shows, the main component is the *Simulator* which contains *Scenario Executer*, *Imperative Model*, *JPL Mediator*, and *Script Handler*. This component (*Simulator*) has dependency to other components which are *Declarative Model*, *JPL*, *GraalVM Javascript Engine*, *Scenario*, and *Output* components. Below is a concise description of the components depicted in the component diagram.

Simulator: This object runs the simulation logic. It receives three input arguments including declarative model, system state, and a scenario for simulation. It initialize the system state, consult declarative model, and then read the scenario line by line. In a loop, it parses and executes every scenario line, and records the result in some output files. Also, it contains the System State which is a collection of (global)

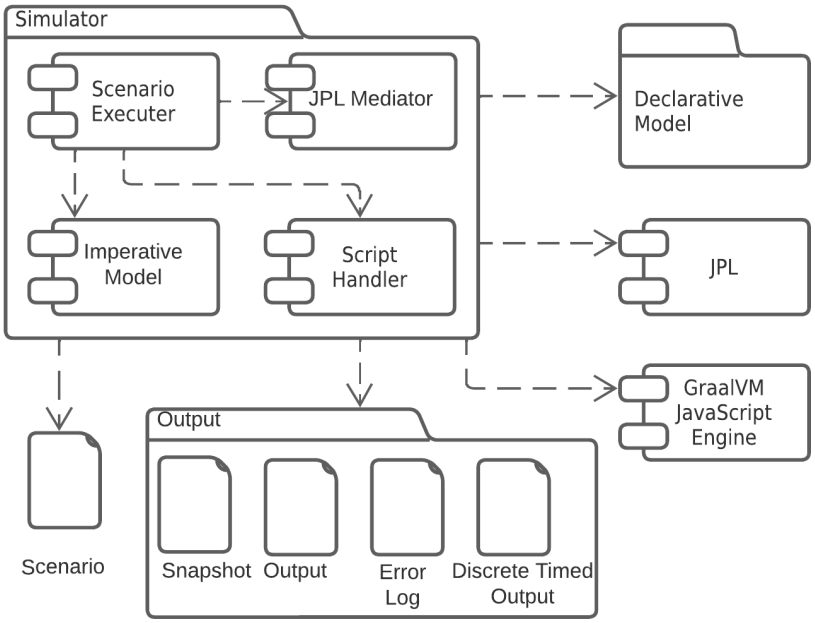

Figure 8.1: The simulator UML component diagram.

variables, defined and initialized in a Javascript file. They are shared among the different functions in the system, and can be modified during the simulation.

Declarative model: comprised of Prolog facts and rules.

Scenario: A sequence of events occurring in the simulation which represents the simulated interaction with the state machine and it is comprised by events or statements (that modify global variables).

JPL: This component is an interface between the simulator in java and declarative model in Prolog. It runs a query and returns transitions which may be triggered under each event or special condition of the system.

GraalVM JavaScript Engine: This component is used by the simulator to evaluate scripts like guards, state's invariants, pre- and post-conditions.

Outputs: This component contains different elements generated by the simulator. It include an output which shows what will happen in each time, a snapshot of the system state, an error log as well as a discrete timed output shows the duration and absolute time information if applicable.

8.3.1.2 The UML class diagram

The UML class diagram for the simulator is shown in figure 8.2. It provides a high-level view of the system's architecture, highlighting the essential classes, their attributes, methods, and relationships between classes. It serves as a blueprint for our system implementation and ensures adherence to the intended design.

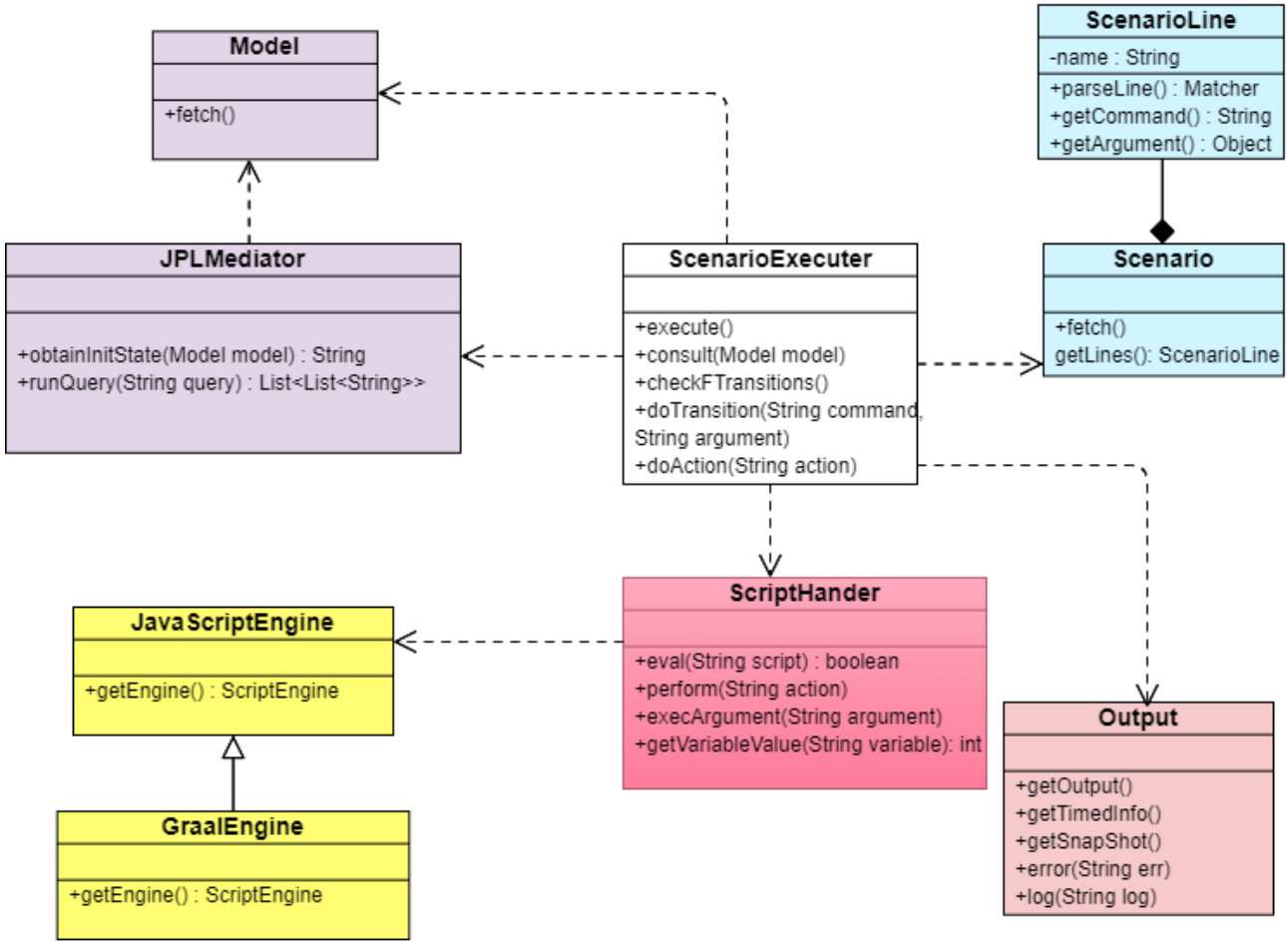

Figure 8.2: The UML class diagram.

8.3.2 Dynamic model

The dynamic model delves into the behavioral aspects of the system, focusing on the interactions and actions that occur during runtime. In this section, we provide an overview of the system's dynamic behavior through the use of several sequence diagrams. These Sequence diagrams illustrate the flow of messages and method calls exchanged between

objects, showcasing the dynamic interactions within the system in response to specific events based on input which is a scenario.

8.3.2.1 The UML sequence diagram

In order to depict the sequential order of interactions between the model key objects, we utilize a UML sequence diagram. The diagram illustrates the interactions among various high-level objects involved in the system. These objects encompass a *Simulator executer* implemented in Java, *JPLMediator* by which simulator consults the declarative model (Prolog facts and rules), *ScriptHandler* which evaluate guards, actions, and set new values for variable, a *Scenario* defined as a text file containing a sequence of events that occur during the simulation, and *Output* generated by the system. Figure 8.3 shows the interaction between these highest level objects.

The system's overall sequence diagram for the successful scenario is depicted in figure 8.3. The outer loop in the sequence diagram illustrates the `Read-Evaluate-Execute` cycle and the inner loop mostly covers ϵ-transitions in our flattened model. However, the system's behavior varies based on the commands specified in the scenario line. Therefore, in the upcoming sections, we present sequence diagrams illustrating the system's behavior in different situations.

8.3.2.2 Visualizing the `EVENT` tag

Sequence diagram in figure 8.4 shows both seccess and failure scenarios when scenario line is an `EVENT` type. After evaluating queried transitions based on current state and event(command of the related scenario line), system checks the number of transitions which their guard is evaluated as true. if the number of true transitions is one, it shows a "success senario", and the corresponding transition will trigger. In this case, system performs action sequence of the triggered transition and then log the result in the output file. Also, system goes from source state to destination state of the transition and the current state will be updated.

After this transition, some following transitions (FTransitions) may trigger. These FTransitions mostly are due to the action sequence of source state onexit action, transition action, and destination state onentry action. The other reason for triggering the FTransitions is due to `when` event type. If the number of true transitions is zero, or the

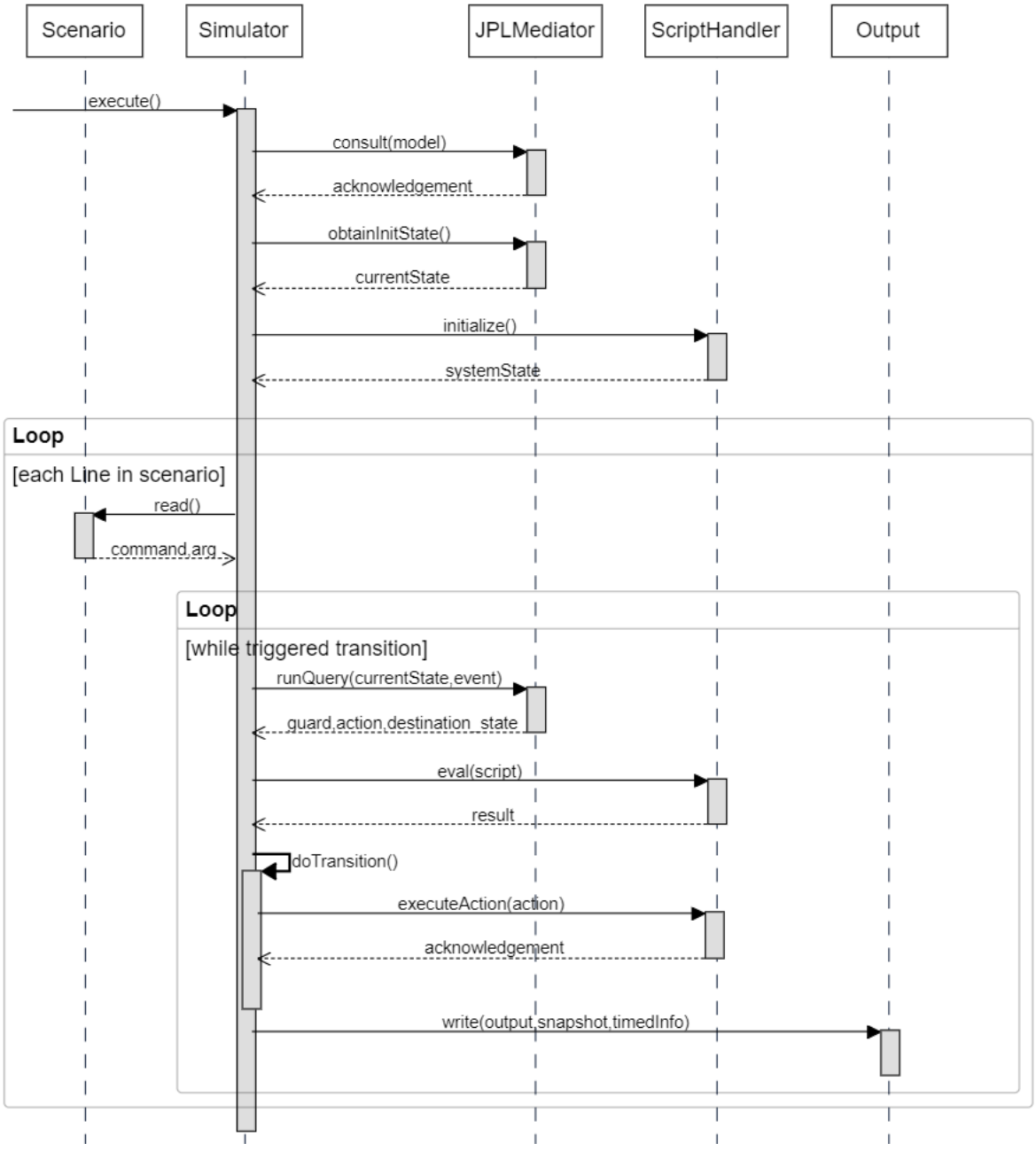

Figure 8.3: The simulator UML sequence diagram.

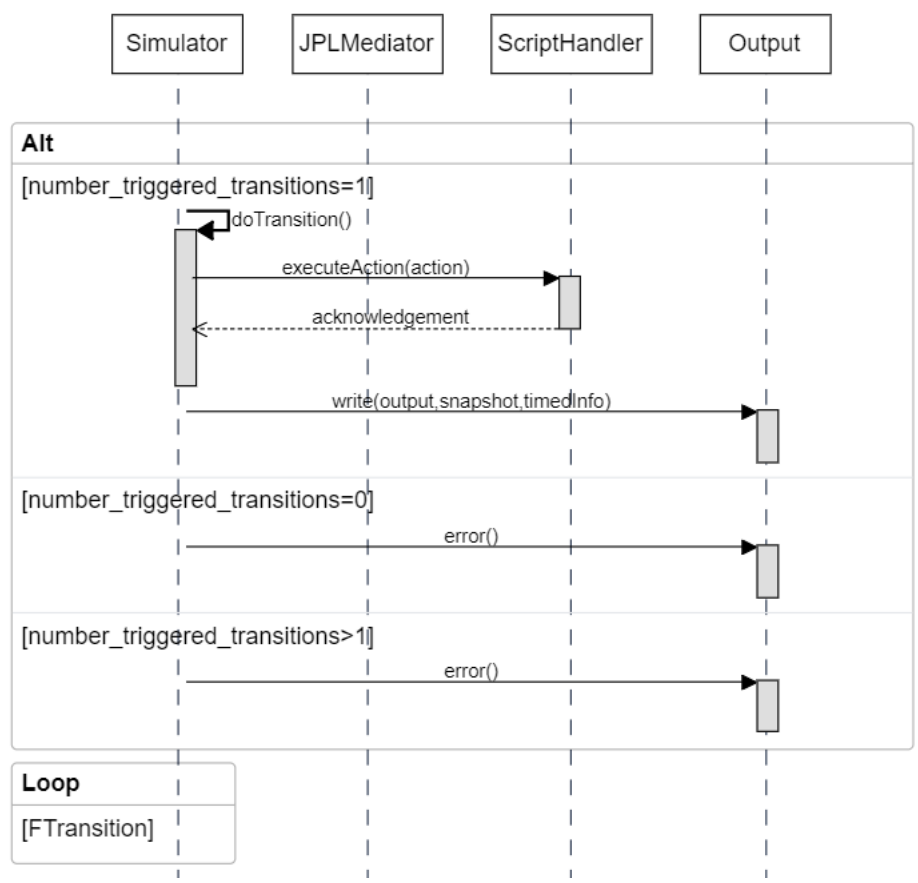

Figure 8.4: The UML sequence diagram for success and failure scenarios for an `EVENT` tag.

number of transitions is greater than one, it shows a "failure scenario" in which system throws an error and log that in the log file.

8.3.3 Visualizing the FTransitions

As explained in section 8.2.2, in FTransitions, system run a query and its related transition until no more transition triggers. In processing of every line of the scenario (after any tag), the last step is checking FTransitions. In this section, the sequence diagram for the FTranstions is presented which is illustrated in figure 8.5.

8.3.4 Visualizing the `Execute` tag

The `EXECUTE` tag is utilized to modify environmental variables within the system, allowing for the simulation of specific conditions and the analysis of system behavior. Following each `EXECUTE` command, the system examines the possibility of triggering any transitions

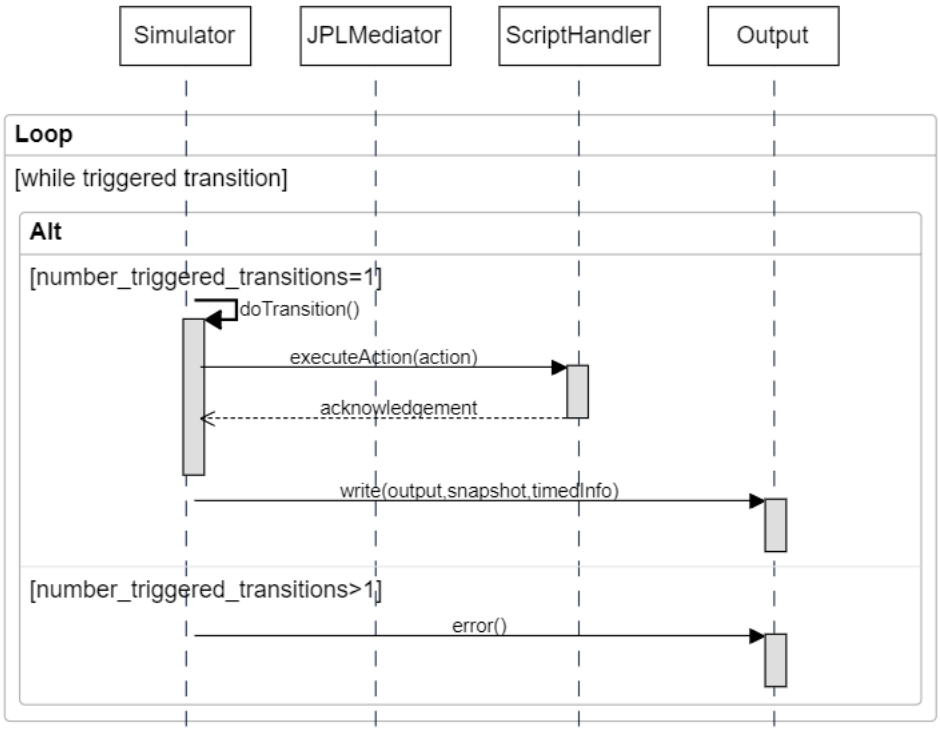

Figure 8.5: The UML sequence diagram for FTransition loop.

through the FTransition loop. Figure 8.6 illustrates the corresponding sequence diagram for this tag, showcasing the dynamic flow of the system during the execution phase.

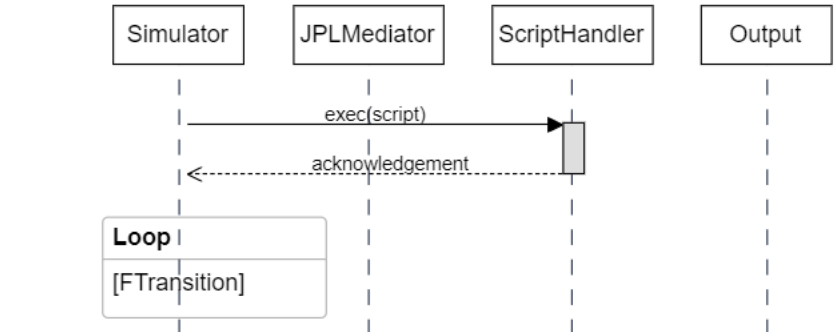

Figure 8.6: The UML sequence diagram for success and failure scenarios for an `EXECUTE` tag.

8.3.5 Visualizing the `TIME` tag

Following the occurrence of any `TIME` tag within the scenario, the system proceeds to update the global time variables, namely duration and absoluteTime if applicable. Subsequently, it evaluates whether any timed-transitions, characterized by an event type of `AFTER` and

AT, are triggered. Finally, the system proceeds to check for any potential FTransitions, ensuring comprehensive examination of the dynamic behavior. The sequence diagram for this situation is illustrated in figure 8.7.

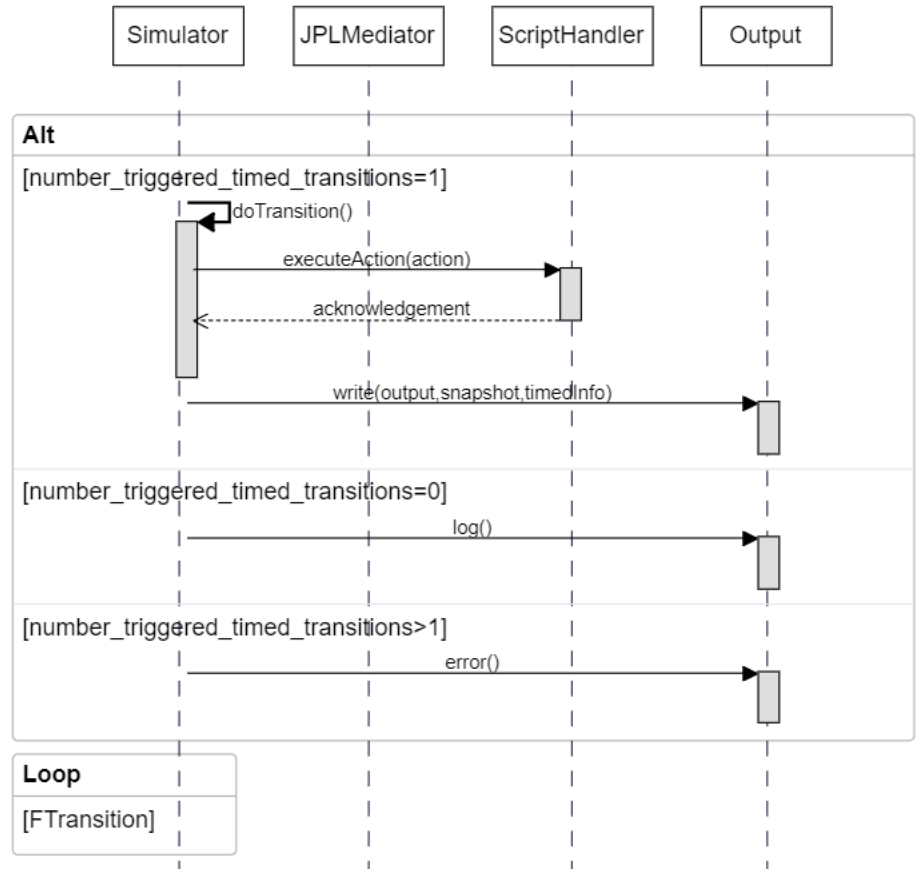

Figure 8.7: The UML sequence diagram for success and failure scenarios for a TIME tag.

8.4 Implementation

8.4.1 Environment

We use SWI-Prolog for representing the declarative model as facts and rules in Prolog. For simulation, we need the persistence of states; however, declarative model does not have this property. On the other hand, We could use Prolog features and define variables in Prolog and achieve some sort of persistency. But, complexity of this approach is too much.

We implement the imperative model in Java for simulating state machines. Prolog has the JPL component which is a Prolog/Java interface and can be used to connect declarative model in Prolog to imperative model in Java. Using the imperative model in Java gives more

capabilities to our model. We can save a history of model variables in a file and organize it more, something that we cannot do in a declarative model in Prolog. Furthermore, we use Javascript for defining and maintaining system variables. Evaluation of events, guards, states's invariants, and pre-/post-conditions of actions will be done by JavaScript engine (GraalVM) through the imperative model in Java.

8.4.2 JPL: A Prolog/Java interface

JPL is a set of Java classes and C functions providing a bidirectional interface between Java and Prolog. JPL uses the Java Native Interface (JNI) to connect to a Prolog engine through the Prolog Foreign Language Interface (FLI). JPL is not a pure Java implementation of Prolog; it makes extensive use of native implementations of Prolog on supported platforms. JPL has been integrated into the full SWI-Prolog distribution starting with version 5.4.x [23]. The Java API comprises public Java classes which support [23]:

- constructing Java representations of Prolog terms and queries.
- calling queries within SWI-Prolog engines.
- retrieving (as Java representations of Prolog terms) any bindings created by a call.

Also, the Prolog API comprises Prolog library predicates which support [23]:

- creating instances (objects) of Java classes (built-in and user-defined).
- calling methods of Java objects (and static methods of classes), perhaps returning values or object references.
- getting and setting the values of fields of Java objects and classes.

8.4.3 GraalVM JavaScript engine

GraalVM's JavaScript engine is a Java application that works on any Java 8+ implementation. The javax.script.ScriptEngine implementation is used in our tool. GraalVM provides an ECMAScript-compliant runtime to execute JavaScript and Node.js applications. It is fully standard compliant, execute applications with high performance, and provide all benefits from the GraalVM stack, including language interoperability and

common tooling [24]. In our tool, We use a javaScript Engine for evaluatin scripts in events, guards and actions.

Chapter 9

Results of simulation

In this chapter, we present the results of modeling, simulation, and studying the behavior of our case study introduced in Chapter 4, which is a state machine of the alarm system.

9.1 The initial and flattened declarative models

Fig. 9.1 illustrates a declarative representation of our case study written in Prolog. This is the initial model which serves as the input for our flattening algorithm. Fig. 9.2 illustrates the output of the flattening algorithm applied on the declarative model in Fig. 9.1, resulting in 17 states and 28 transitions.

```
% level 1
state(idle).
state(active).
state(error).
state(final).
initial(idle).
final(final).
alias(final, '').
entry_pseudostate(active_skip_config_entry, reading). % active super-state is implied
exit_pseudostate(active_exit, active). % the out transition may be guarded
transition(idle, active, event(call, activate), nil, nil).
transition(idle, active_skip_config_entry, event(call, 'skip configuring')), nil, nil).
trasition(error, active, event(call, reset), nil, nil).
transition(active, idle, event(call, deactivate), nil, nil).
transition(idle, final, event(call, shutoff), nil, nil).
transition(active_exit, error, nil, nil, nil).

% level 2
superstate(active, configuring).
superstate(active, reading).
superstate(active, emergency).
initial(configuring).
onentry_action(active, action(log, "Green LED ON")).
onexit_action(active, action(log, "Green LED OFF")).
onentry_action(configuring, action(exec, "echo('Configuring mode');")).
onexit_action(configuring, action(exec, "echo('Exit configuring mode');")).
do_action(reading, proc('Slow blinking red LED')).
transition(configuring, reading, event(call, cancel), nil, action(exec, "longBeep();")).
transition(configuring, active_exit, event(timeout, "2:00"), nil, action(exec, "beep();")).
transition(reading, emergency, event(when, "tCurrent >= tThreshold"),
    nil, action(exec, "sendNotification();")).
transition(reading, configuring, event(call, set), nil, nil).
transition(emergency, reading, event(call, reset), nil, nil).
transition(emergency, reading, nil, nil, nil). % completed emergency
transition(configuring, reading, event(call, done), "tThreshold > tCurrent", nil, nil).
internal_transition(configuring, event(set, tThreshold), nil, action(exec, "doubleBeep();")
internal_transition(configuring, event(call, done),
    "tThreshold <= tCurrent", action(exec, "generateError();")).

% level 3
do_action(emergency, proc('Make Siren Sound')).
onexit_action(emergency, action(exec,"echo('Exit Emergency');")).
superstate(emergency, activated).
superstate(emergency, efinal).
initial(activated).
final(efinal).
```

Figure 9.1: Initial model.

```
state(active_skip_config_entry).
state(idle).        state(error).
state(final).       state(pre_configuring).
state(configuring). state(reading).
state(active_exit). state(activated).
state(efinal).      state(s2).
state(s3).          state(s61).
state(s71).         state(s81).
state(s91).         state(s92).
initial(idle).      final(final).
transition(idle, final, event(call, shutoff), nil, nil).
transition(idle, active_skip_config_entry, event(call, 'skip configuring'), nil,
    action(log, "Green LED ON")).
transition(idle, pre_configuring, event(call, activate), nil, action(log, "Green LED ON")).
transition(error, pre_configuring, event(call, reset), nil, action(log, "Green LED ON")).
transition(configuring, configuring, event(set, tThreshold), nil,
    action(exec, "doubleBeep();")).
transition(configuring, configuring, event(call, done), "tThreshold <= tCurrent",
    action(exec, "generateError();")).
transition(configuring, s2, event(call, deactivate), nil,
    action(exec, "echo('Exit configuring mode');")).
transition(configuring, s3, event(after, '2:00'), nil,
    action(exec, "echo('Exit configuring mode');")).
transition(configuring, s81, event(call, cancel), nil,
    action(exec, "echo('Exit configuring mode');")).
transition(configuring, active_skip_config_entry, event(call, done),
    "tThreshold > tCurrent", nil, action(exec, "echo('Exit configuring mode');")).
transition(reading, s91, event(when, "tCurrent >= tThreshold"), nil,
    action(log, "ABORT 'Slow blinking red LED'")).
transition(reading, s2, event(call, deactivate), nil,
    action(log, "ABORT 'Slow blinking red LED'")).
transition(reading, pre_configuring, event(call, set), nil,
    action(log, "ABORT 'Slow blinking red LED'")).
transition(activated, efinal, event(after, "2:00"), nil, nil).
transition(activated, s61, event(call, reset), nil,
    action(log, "ABORT 'Make Siren Sound'")).
transition(activated, s71, event(call, deactivate), nil, action(log, "ABORT 'Make Siren Sound'")).
transition(activated, s61, event(completed, emergency), nil, action(log, "STOP 'Make Siren Sound'")).
transition(active_exit, error, nil, nil, action(log, "Green LED OFF")).
transition(pre_configuring, configuring, nil, nil, action(exec, "echo('Configuring mode');")).
transition(active_skip_config_entry, reading, nil, nil, action(log, "START 'Slow blinking red LED'")).
transition(efinal, s61, nil, nil, action(log, "STOP 'Make Siren Sound'")).
transition(s2, idle, nil, nil, action(log, "Green LED OFF")).
transition(s3, active_exit, nil, nil, action(exec, "beep();")).
transition(s91, s92, nil, nil, action(exec, "sendNotification();")).
transition(s92, activated, nil, nil, action(log, "START 'Make Siren Sound'")).
transition(s61, active_skip_config_entry, nil, nil, action(exec,"echo('Exit Emergency');")).
transition(s81, active_skip_config_entry, nil, action(exec, "longBeep();")).
transition(s71, s2, nil, nil, action(exec,"echo('Exit Emergency');")).
```

Figure 9.2: Flattened model.

9.1.1 Comparison the complexity of two declarative models

Having the initial and flattened declarative models of our case study, we can compare these two models in terms of complexity. This comparison may include comparing the number of

states and nested states or number of transitions in each model. The result of comparison for the case study is summarized in Table 9.1.

Table 9.1: Complexity comparison of initial and flattened model for the case study.

MEASURE	INITIAL MODEL	FLATTENED MODEL
number of states and substates	9	18
number of nested states	5	0
number of internal initial states	2	0
number of transitions	16	29
number of internal transitions	2	0
number of entry pseudo states	1	0
number of exit pseudo states	1	0
number of entry behavior	2	0
number of do behavior	2	0
number of exit behavior	3	0
number of guards	2	2
number of actions	10	26
number of nil transitions	2	11
number of levels	3	1

9.2 Simulation scenarios

We can define any sequences of `EVENT`, `EXECUTE`, and `TIME` tags as scenarios and study the behavior of the alarm system under each scenario. Each line of scenario consists of a tag and the associated command. After running each scenario, the simulator will generate three files including system behavior as discrete timed events, a snapshot of system state in each time step, and system time information.

Line numbers in the scenario help us track the output generated by the simulator processing the command of each scenario line. Each line in the output includes the discrete time id, followed by the scenario line number, the output type, and its corresponding arguments. Using the output, we can analyze the behaviour of the system including sequence of states or actions performed by the system. Also, it can highlight any potential flaws in the design of the state machine.

9.2.1 Output of the simulation: scenario 1

The first scenario is shown in Fig.9.3. The outputs of simulation this scenario as scenario 1 in presented in Fig.9.4, Fig.9.5, and Fig.9.6. The line number of commands in the scenario allows us to track the output of each command in the output files.

After simulation this scenario, outputs are as expected according to the state machine. System starts from `idle` state, goes to `configuring` which is a substate of `active`, then it has a transition to `error`, goes back to `configuring`, goes to `reading`, `activated`, `idle`, and `final`. Transitions to intermediate states (produced in flattening process) show the sequence by which actions are executed including on_entry, on_exit, do_behaviour, and transition actions.

In time information output (Fig.9.6), the last column shows the duration in each time_id based on the ISO 8601 duration format. In this format, the letter "P" stands for "Period." Following the "P" designator, it can specify the duration using a combination of time elements like "H" for hours, "M" for minutes, and "S" for seconds.

```
% LINE COMMAND ARG
1 EVENT call activate
2 EVENT set tThreshold=30
3 AFTER '3:00'
4 EVENT call reset
5 EXECUTE tCurrent=20
6 EVENT call done
7 EXECUTE tCurrent=40
8 EVENT call deactivate
9 EVENT call shutoff
```

Figure 9.3: Scenario 1.

```
% TIME,LINE,TYPE,ARGS
1,1,EVENT,call,activate
1,1,transition,pre_configuring
1,1,action,"Green LED ON"
1,1,transition,configuring
1,1,action,"EXEC echo('Configuring mode');"
2,2,EVENT,set,tThreshold=30"
2,2,transition,configuring
2,2,action,"EXEC doubleBeep();"
3,3,transition,s3
3,3,action,"EXEC echo('Exit configuring mode');"
3,3,transition,active_exit
3,3,action,"EXEC beep();"
3,3,transition,error
3,3,action,"Green LED OFF"
4,4,EVENT,call,reset
4,4,transition,pre_configuring
4,4,action,"Green LED ON"
4,4,transition,configuring
4,4,action,"EXEC echo('Configuring mode');"
5,5,EXECUTE,"tCurrent=20"
6,6,EVENT,call,done
6,6,transition,active_skip_config_entry
6,6,action,"EXEC echo('Exit configuring mode');"
6,6,transition,reading
6,6,action,"START 'Slow blinking red LED'"
7,7,EXECUTE,"tCurrent=40"
8,7,transition,s91
8,7,action,"ABORT 'Slow blinking red LED'"
8,7,transition,s92
8,7,action,"EXEC sendNotification();"
8,7,transition,activated
8,7,action,"START 'Make Siren Sound'"
9,8,EVENT,call,deactivate
9,8,transition,s71
9,8,action,"ABORT 'Make Siren Sound'"
9,8,transition,s2
9,8,action,"EXEC echo('Exit Emergency');"
9,8,transition,idle
9,8,action,"Green LED OFF"
10,9,EVENT,call,shutoff
10,9,transition,final
10,9,action,nil
```

Figure 9.4: Scenario 1: discrete timed output of system events.

```
% TIME,tThreshold,tCurrent
1,30,5
2,30,5
3,30,5
4,30,5
5,30,5
6,30,20
7,30,20
8,30,40
9,30,40
10,30,40
```

Figure 9.5: Scenario 1: discrete timed snapshot.

```
% TIME,absoluteTime,duration
1,null,PT0S
2,null,PT0S
3,null,PT3M
4,null,PT3M
5,null,PT0S
6,null,PT0S
7,null,PT0S
8,null,PT0S
9,null,PT0S
10,null,PT0S
```

Figure 9.6: Scenario 1: discrete timed system time information.

9.2.2 Output of the simulation: scenario 2

The second scenario is shown in Fig.9.7. The outputs of simulating this scenario as scenario 2 is presented in, Fig.9.10, and Fig.9.11. Running this scenario, the simulator identifies it as an `Exceptional Condition` which triggers one of the error conditions handled in implementation of the tool in Java program. The exception is shown in Fig.9.8. It is related to line 13 of scenario 2, which is a `call` event, but no transition is triggered, causing this exception.

Looking into the state machine, we come to the conclusion that the machine correctly captures what requirements specify. By line 10 of scenario 2, the machine goes into `activated` substate of `emergency` because tCurrent>=tThreshold. After that, there are two consequtive events in the scenario (`reset` and `set`) to bring the machine to `configuring` state. However, this expected sequence of transitions will not happen since the system goes back to `emergency` from `reading`. The reason is that in `reading`, the machine first will check when transition condition which is true and fires this transition.

This scenario can serve as an example which shows requirements of the system are not defined completely. In this specific case, we can modify the design of the state machine by modifying requirements to define an action for changing the tThreshold before entering `reading` or changing the transition from `emergency` to `configuring` instead of `reading`.

```
% LINE COMMAND ARG
1 EVENT call 'skip configuring
2 EXECUTE tCurrent=0
3 AT '10:00'
4 EXECUTE tCurrent=50
5 EVENT call reset
6 AT '11:00'
7 EXECUTE tCurrent=40
8 EVENT call reset
9 AFTER '3:00'
10 EXECUTE tCurrent=30
11 EVENT call reset
12 EVENT call set
13 EVENT call deactivate
```

Figure 9.7: Scenario 2.

```
SEVERE  Error: Number of transitions with true guards is zero!

Process finished with exit code 0
```

Figure 9.8: Discovery of a compliance requirement gap by simulating the state machine.

```
% TIME,LINE,TYPE,ARGS
1,1,EVENT,call,"skip configuring"
1,1,transition,active_skip_config_entry
1,1,action,"Green LED ON"
1,1,transition,reading
1,1,action,"START 'Slow blinking red LED'"
2,2,EXECUTE,"tCurrent=0"
3,4,EXECUTE,"tCurrent=50"
4,4,transition,s91
4,4,action,"ABORT 'Slow blinking red LED'"
4,4,transition,s92
4,4,action,"EXEC sendNotification();"
4,4,transition,activated
4,4,action,"START 'Make Siren Sound'"
5,5,EVENT,call,reset
5,5,transition,s61
5,5,action,"ABORT 'Make Siren Sound'"
5,5,transition,active_skip_config_entry
5,5,action,"EXEC echo('Exit Emergency');"
5,5,transition,reading
5,5,action,"START 'Slow blinking red LED'"
6,5,transition,s91
6,5,action,"ABORT 'Slow blinking red LED'"
6,5,transition,s92
6,5,action,"EXEC sendNotification();"
6,5,transition,activated
6,5,action,"START 'Make Siren Sound'"
7,7,EXECUTE,"tCurrent=40"
8,8,EVENT,call,reset
8,8,transition,s61
8,8,action,"ABORT 'Make Siren Sound'"
8,8,transition,active_skip_config_entry
8,8,action,"EXEC echo('Exit Emergency');"
8,8,transition,reading
8,8,action,"START 'Slow blinking red LED'"
9,8,transition,s91
9,8,action,"ABORT 'Slow blinking red LED'"
9,8,transition,s92
9,8,action,"EXEC sendNotification();"
9,8,transition,activated
9,8,action,"START 'Make Siren Sound'"
10,9,transition,efinal
10,9,action,nil
10,9,transition,s61
10,9,action,"STOP 'Make Siren Sound'"
10,9,transition,active_skip_config_entry
10,9,action,"EXEC echo('Exit Emergency');"
10,9,transition,reading
10,9,action,"START 'Slow blinking red LED'"
11,9,transition,s91
11,9,action,"ABORT 'Slow blinking red LED'"
11,9,transition,s92
11,9,action,"EXEC sendNotification();"
11,9,transition,activated
11,9,action,"START 'Make Siren Sound'"
12,10,EXECUTE,"tCurrent=30"
13,11,EVENT,call,reset
13,11,transition,s61
13,11,action,"ABORT 'Make Siren Sound'"
13,11,transition,active_skip_config_entry
13,11,action,"EXEC echo('Exit Emergency');"
13,11,transition,reading
13,11,action,"START 'Slow blinking red LED'"
14,11,transition,s91
14,11,action,"ABORT 'Slow blinking red LED'"
14,11,transition,s92
14,11,action,"EXEC sendNotification();"
14,11,transition,activated
14,11,action,"START 'Make Siren Sound'"
15,13,EVENT,call,set
```

Figure 9.9: Scenario 2: discrete timed output of system events.

```
% TIME,tThreshold,tCurrent
1,10,5
2,10,5
3,10,0
4,10,50
5,10,50
6,10,50
7,10,50
8,10,40
9,10,40
10,10,40
11,10,40
12,10,40
13,10,30
14,10,30
```

Figure 9.10: Scenario 2: discrete timed snapshot.

```
% TIME,absoluteTime,duration
1,null,PT0S
2,null,PT0S
3,10:00,PT0S
4,10:00,PT0S
5,null,PT0S
6,null,PT0S
7,11:00,PT0S
8,null,PT0S
9,null,PT0S
10,null,PT3M
11,null,PT3M
12,null,PT3M
13,null,PT0S
14,null,PT0S
```

Figure 9.11: Scenario 2: discrete timed system time information.

9.2.3 Output of the simulation: scenario 3

The third scenario is shown in Fig.9.12. This scenario is somehow similar to scenario 2; but, we updated tCurrent value before entering to `reading`, and as a result system will not come back to `emergency` again.

Line 5 of this scenario demonstrates region completion of `emergency`, which triggers the transition from `emergency` to `reading` with no event. Since `emergency` has a do_behavior, we expect that completing its do_behavior yields the same result. We changed line 5 to "EVENT completed emergency" and observed the same outputs, thereby verifying the correct behavior of the simulator. The outputs of simulating this scenario as scenario 3 is

presented in Fig.9.13, Fig.9.14, and Fig.9.15.

```
% LINE COMMAND ARG
1 EVENT call activate
2 EVENT call cancel
3 EXECUTE tCurrent=35
4 EXECUTE tCurrent=25
5 AFTER '2:00'
6 EVENT call deactivate
7 EVENT call shutoff
```

Figure 9.12: Scenario 3.

```
% TIME,LINE,TYPE,ARGS
1,1,EVENT,call,activate
1,1,transition,pre_configuring
1,1,action,"Green LED ON"
1,1,transition,configuring
1,1,action,"EXEC echo('Configuring mode');"
2,2,EVENT,call,cancel
2,2,transition,s81
2,2,action,"EXEC echo('Configuring mode');"
2,2,transition,active_skip_config_entry
2,2,action,"EXEC longBeep();"
2,2,transition,reading
2,2,action,"START 'Slow blinking red LED'"
3,3,EXECUTE,"tCurrent=35"
4,3,transition,s91
4,3,action,"ABORT 'Slow blinking red LED'"
4,3,transition,s92
4,3,action,"EXEC sendNotification();"
4,3,transition,activated
4,3,action,"START 'Make Siren Sound'"
5,4,EXECUTE,"tCurrent =25"
6,5,transition,efinal
6,5,action,nil
6,5,transition,s61
6,5,action,"STOP 'Make Siren Sound'"
6,5,transition,active_skip_config_entry
6,5,action,"EXEC echo('Exit Emergency');"
6,5,transition,reading
6,5,action,"START 'Slow blinking red LED'"
7,6,EVENT,call,deactivate
7,6,transition,s2
7,6,action,"ABORT 'Slow blinking red LED'"
7,6,transition,idle
7,6,action,"Green LED OFF"
8,7,EVENT,call,shutoff
8,7,transition,final
8,7,action,nil
```

Figure 9.13: Scenario 3: discrete timed output of system events.

```
% TIME,tThreshold,tCurrent
1,30,5
2,30,5
3,30,5
4,30,35
5,30,35
6,30,25
7,30,25
8,30,25
```

Figure 9.14: Scenario 3: discrete timed snapshot.

```
% TIME,absoluteTime,duration
1,null,PT0S
2,null,PT0S
3,null,PT0S
4,null,PT0S
5,null,PT0S
6,null,PT2M
7,null,PT2M
8,null,PT0S
```

Figure 9.15: Scenario 3: discrete timed system time information.

9.3 Visualizing the results

In this section, we visualize the results of simulating scenarios and show the model of behavior for each scenario. This diagram shows the current state of the state machine as well as state of the system in each time id.

The model of behavior of the system under scenario 1 through 3 is presented in Fig.9.16, Fig.9.17, and Fig.9.18 respectively.

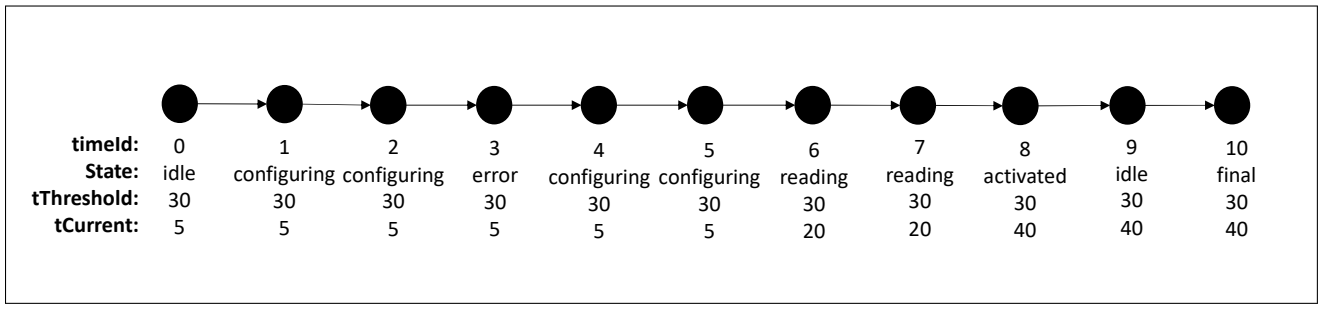

Figure 9.16: Scenario 1: model of behavior.

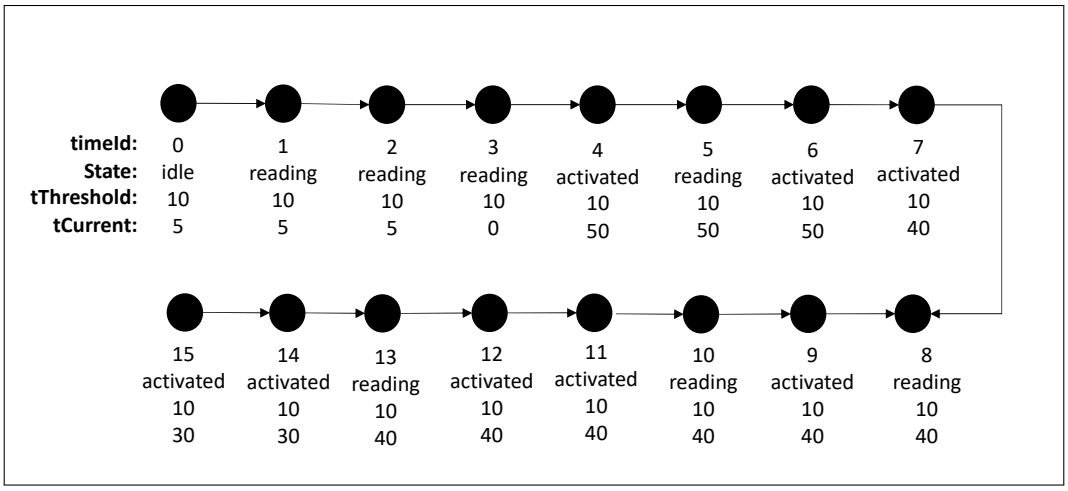

Figure 9.17: Scenario 2: model of behavior.

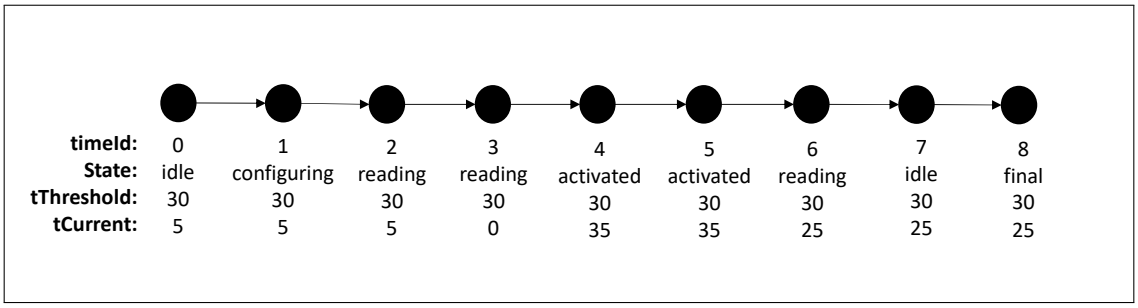

Figure 9.18: Scenario 3: model of behavior.

Chapter 10

Conclusions and future work

10.1 Conclusions

State machines are widely used to model the dynamic behavior of systems, and in this thesis, we propose transforming these state machines into a database of clauses in Prolog, enabling a declarative representation. We introduced an algorithm to flatten the UML state machine and convert it into an extended finite state machine. Our algorithm supports major UML 2.5.1 features including single and composite states; exit and entry pseudostates; state behaviors including entry, do, and exit; in addition to the UML events including call, signal, time, change, as well as three newley introduced events namely inactivity, update, and completion. We used a modified version of the extended finite state machine to support guarded and unguarded ϵ-transitions that are required for handling complex sequences of actions and notifications in a non-flattened model.

Both initial and flattened models provides a factbase that serves as a foundation for analyzing the behavior, complexity, and structure of the state machine. Therefore, the declarative model is extended by building a query system as Prolog rules by which we can study properties and the behaviour of a state machine.

In order to study the dynamic behaviour of a state machine under different scenarios which may happen to the system, we developed a simulator. To support simulation, we developed a tool in Java using the Java Prolog Library (JPL) and JavaScript Engine. This tool takes a given scenario as input and generates the machine's behavior at discrete time steps as output. The simulation process allows the study of the state machine's behavior under different conditions and helps identify potential flaws in its design.

We present a case study of an alarm system to demonstrate the effectiveness of our approach. We successfully apply the flattening algorithm and simulate the behavior of the state machine under three different scenarios, showcasing the power and utility of our proposed modeling and simulation tool. After study the case study, we can conclude this research as the following points:

- Using Prolog facts and rules has been used in literature for modeling UML diagrams and consistency checking. In this research, we used prolog for modeling state machine and proposed our declarative model which is a novelty compared to literature which studied other UML diagrams. Our both declarative model (initial and flattened) are used in our querying platform and also as a facbase for our simulator tool. Prolog gives us capabilities like pattern matching/backtracking and good engine to use selectors for high level queries.

- This thesis provides a powerful tool for analyzing the two aspects of the state machines (imperative analysis as well as the declarative analysis).

- The output of the query platform and simulation can be used for requirement validation and verification providing feedback for developers and stakeholders of the system.

- Our approach is limited and the search space is not discrete (there are infinite number of scenarios and with regards to the continuous nature of the variables, it would be impossible to find all possible success/fail cases). Therefore, the simulator may be used as a test environment in the same way that a test engine is used to validate test cases.

- We visualize the outputs of the simulation as the models of behavior. Visualization of models of behavior would be nice and provides a good representaion of the state of the system during the simulation.

10.2 Future work

The results of the simulation can be used to analyze the behaviour of the machines. Our discrete timed output can be used to run some temporal logic queries which can be done

as a future work. Also, as another potential area for future work, the simulator can be extended to incorporate contract considerations for state invariants, as well as pre- and post-conditions for actions. Furthermore, our model can be extended to include additioanl UML features like History pseudostate and orthogonal regions.

In this research, we treated `timeout` event type as `after` in our simulation that makes sense since we define and have control over the sequence of events in scenarios. However, the simulation model can be extended to distinguish between `timeout` and `after` event types. a `timeout` event shows that system is idle for the specified time; in contrast, `after` points out to passing the specified time.

References

[1] L. Sterling and E. Shapiro, *The art of Prolog: advanced programming techniques*. Cambridge, MA: MIT Press, second ed., 1994.

[2] A. Gill, *Introduction to the theory of finite-state machines*. Electronic science series, McGraw-Hill, 1962.

[3] D. Harel, "Statecharts: a visual formalism for complex systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, 1987.

[4] Object Management Group, "UML® 2.5.1." `https://www.omg.org/spec/UML/2.5.1/`, 2017.

[5] K.-T. T. Cheng and A. Krishnakumar, "Automatic generation of functional vectors using the extended finite state machine model," *ACM transactions on design automation of electronic systems*, vol. 1, 05 1999.

[6] A. Decan and T. Mens, "Sismic - a Python library for statechart execution and testing," *SoftwareX*, vol. 12, p. 100590, 2020.

[7] F. Sheng, H. Zhu, Z. Yang, J. Yin, and G. Lu, "Verifying static aspects of UML models using Prolog," in *The 31st international conference on software engineering and knowledge Engineering, SEKE 2019, Portugal, July 10-12, 2019* (A. Perkusich, ed.), pp. 259–342, KSI Research Inc. and knowledge systems institute graduate school, 2019.

[8] Z. Khai, A. Nadeem, and G.-s. Lee, "A Prolog based approach to consistency checking of UML class and sequence diagrams," in *Software engineering, business continuity, and education* (T.-h. Kim, H. Adeli, H.-k. Kim, H.-j. Kang, K. J. Kim, A. Kiumi, and B.-H. Kang, eds.), pp. 85–96, Springer Berlin Heidelberg, 2011.

[9] N. L. Hashim and Y. S. Dawood, "A review on test case generation methods using UML statechart," in *2019 4th International conference and workshops on recent advances and innovations in engineering (ICRAIE)*, pp. 1–5, 2019.

[10] C. Chen and W. Lin, "Research of software testing technology based on statechart diagram," in *Advances in intelligent information hiding and multimedia signal processing* (J.-S. Pan, J. Li, O.-E. Namsrai, Z. Meng, and M. Savić, eds.), pp. 314–322, Springer Singapore, 2021.

[11] M. Aktaş and T. Ovatman, "UML statechart anti-patterns," in *2022 IEEE 46th annual computers, software, and applications conference (COMPSAC)*, pp. 413–414, 2022.

[12] T. Mens, A. Decan, and N. I. Spanoudakis, "A method for testing and validating executable statechart models," *Software and Systems Modeling*, vol. 18, p. 837–863, apr 2019.

[13] S. Van Mierlo and H. Vangheluwe, "Introduction to statecharts modeling, simulation, testing, and deployment," in *2019 Winter simulation conference (WSC)*, pp. 1504–1518, 2019.

[14] D. Balasubramanian, C. S. Păsăreanu, G. Karsai, and M. R. Lowry, "Polyglot: systematic analysis for multiple statechart formalisms," in *Tools and algorithms for the construction and analysis of systems* (N. Piterman and S. A. Smolka, eds.), pp. 523–529, Springer Berlin Heidelberg, 2013.

[15] D. Harel and A. Naamad, "The STATEMATE semantics of statecharts," *ACM Transactions on Software Engineering and Methodology*, vol. 5, no. 4, p. 293–333, 1996.

[16] D. Harel and H. Kugler, *The rhapsody semantics of statecharts (or, on the executable core of the UML)*, pp. 325–354. Springer Berlin Heidelberg, 2004.

[17] S. E. V. and P. Samuel, "Automatic code generation from UML state chart diagrams," *IEEE Access*, vol. 7, pp. 8591–8608, 2019.

[18] N. I. S. Tom Mens, Alexandre Decan, "A method for testing and validating executable statechart models," *SoftwareX*, vol. 12, p. 100590, 2020.

[19] K. Jin and K. Lano, "Generation of test cases from UML diagrams - a systematic literature review," in *14th Innovations in software engineering conference (formerly known as India software engineering conference)*, ISEC 2021, (New York, NY, USA), Association for Computing Machinery, 2021.

[20] P. V. Murthy, P. C. Anitha, M. Mahesh, and R. Subramanyan, "Test ready UML statechart models," in *Proceedings of the 2006 international workshop on scenarios and state machines: models, algorithms, and tools*, SCESM '06, (New York, NY, USA), p. 75–82, Association for Computing Machinery, 2006.

[21] E. Sekerinski, "Verifying statecharts with state invariants," in *13th IEEE international conference on engineering of complex computer systems (ICECCS 2008)*, pp. 7–14, 2008.

[22] E. S. S. Freire, G. C. Oliveira, and M. E. de Sousa Gomes, "Analysis of open-source case tools for supporting software modeling process with UML," in *Proceedings of the XVII brazilian symposium on software quality*, SBQS '18, (New York, NY, USA), p. 51–60, Association for Computing Machinery, 2018.

[23] "JPL, an API between SWI-Prolog and the Java virtual machine." `https://jpl7.org/DevelopmentJPL-Devel`. 2003 - 2018, Fred Dushin, Paul Singleton, Jan Wielemaker and JPL contributors.

[24] "GraalVM." `https://www.graalvm.org/`.