Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Radnai Bálint

INTEGRATION OF SCXML STATE MACHINES TO THE GAMMA FRAMEWORK

Bachelor's thesis

ADVISOR

Graics Bence

BUDAPEST, 2022

Contents

HALLGATÓI NYILATKOZAT

Alulírott **Radnai Bálint**, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2022. 12. 09.

...…….
Radnai Bálint

Összefoglaló

Napjainkban egyre fontosabb szerepet kapnak a modellalapú szoftverfejlesztési paradigmák, mivel a reaktív rendszerek és szakterület-specifikus modellek komplexitása az utóbbi években nagymértékben megnőtt. A modellvezérelt fejlesztőeszközök lehetővé teszik, hogy a rendszertervező mérnökök modellekben gondolkozhassanak. Az elkészült modellekhez a keretrendszer automatikus verifikációs, illetve forráskód generáló eszközöket biztosít, ezáltal felgyorsítva a fejlesztési folyamatot. A rendszerek és komponenseik belső viselkedésének leírására gyakran alkalmazunk állapot alapú modelleket. Az állapot alapú fejlesztést támogató eszközök – mint az UML vagy az SCXML – hiányossága, hogy jellemzően nem definiálnak precíz végrehajtási szemantikát, így nem teszik lehetővé a modellek formális verifikációját, illetve forráskód generálását sem.

Munkámban egy olyan modelltranszformációt mutatok be, amely lehetővé teszi modellvezérelt fejlesztőeszközök alkalmazását SCXML modelleken. A transzformáció SCXML állapotgépeket alakít a Gamma Állapotgép Kompozíciós Keretrendszer köztes állapotgép nyelvére. E transzformáció eredményeképp a keretrendszer formális modellellenőrzési, illetve automatikus teszt- és kódgenerálási funkciói elérhetővé válnak az SCXML modellekhez is. A transzformáció az Eclipse EMF és az Xtext keretrendszereken alapul, és az SCXML állapotgépek nyelvi elemeinek többségét képes a Gamma keretrendszer megfelelő modellelemeire leképezni. Azért, hogy a transzformáció megőrizze az SCXML modellek szemantikai sajátosságait, összehasonlítom egymással a két állapot alapú formalizmus szintaktikáját és szemantikáját. A modelltranszformáció képességeit a Gamma útmutatójában szereplő projekten mutatom be, amely egy útkereszteződés közlekedési lámpáinak vezérlőit modellezi.

A modelltranszformációt jelen állapotában vagy önálló SCXML állapotgépekre, vagy több együttműködő SCXML állapotgépből álló, összetett SCXML modellen lehet végrehajtani. A későbbiekben kiegészíthető az SCXML fennmaradó vagy opcionális nyelvi elemeinek leképezésével.

Abstract

Model-driven software development approaches are getting more and more important as the complexity of component-based reactive systems and domain-specific models has been increasing in the last years. Model-driven development tools speed up the development process by allowing system architects thinking in models, and provide verification and automatic code generation capabilities based on these models. The internal behavior of systems and their components is usually represented using state-based formalisms. Several of the widely used tools which support state-based modeling like UML and SCXML provide execution semantics, but intentionally leave parts of their dynamic semantics un- or underspecified. To ensure the compatibility and traceability of models and generated code, rigorous model semantics is needed.

To help modelers use model-driven features with precise semantics on SCXML models, a model transformation is presented that translates SCXML state machines to the intermediate statechart language of the Gamma Statechart Composition Framework. By this transformation, the automatic model checking, code generation and model-based test generation features can be used on transformed SCXML models. The transformation builds on Eclipse EMF and the Xtext framework. The transformer is capable of mapping most elements of an individual SCXML state machine to Gamma model elements. To ensure semantic equivalence of transformed elements, the two formalisms are compared both syntactically and semantically. The capabilities of the transformer are demonstrated on a crossroad model included in the tutorial project of Gamma.

This model transformer can translate individual SCXML state machines as well as composite components containing state machines. It can later be extended with the transformation of remaining language elements of SCXML.

1. Introduction

1.1. Model-driven development

Model-driven engineering[1] is a software development methodology that focuses on creating and exploiting domain models, which are conceptual models of all the topics related to a specific problem. It highlights abstract representation of knowledge of a particular application domain rather than computing concepts.

Model-driven software development tools make use of models. A *model* is the simplified image of an element of the real or a hypothetical world (the system), that replaces the system in certain considerations. Models are abstractions that focus only on the important features of the modeled system from the point of the purpose of modeling.

An advantage of model-driven software design is automatic source code generation from models, which makes the complexity of the modeled system manageable, and ensures consistency between models and generated code as well as reducing human errors in the system implementation. This is the reason why this development paradigm is usually applied in the development of safety-critical systems, since validation and verification steps have to be taken at each phase of system design [1].

1.2. Overview

Different modeling languages aim to serve different modeling purposes, and this is the case for statechart modeling languages, too. One formalism may provide more rigorous rules on how modeles state machines have to be executed. This can be done using formal semantics that define with mathematical precision how a model has to be interpreted. Other formalisms may lack of such rigorous semantics, but provide a more open modeling formalism. Implementations of these languages are free to implement these intentionally underspecified or unspecified modeling features for their purposes (possibly with optimizations).

Modelers may want to create a model that they wish to describe models that can be easily validated and verified, but the modeling framework they use do not provide tools for doing this. In this cases, a model-to-model transformation can help to transform

[1] Model-driven engineering, https://en.wikipedia.org/wiki/Model-driven_engineering

their model to a formalism in the background that they do not need to learn, just use its automatic features.

In this work, I created a model-to-model transformation that maps the metamodel elements of a general purpose state-based modeling language to the metamodel elements of another formalism that help modelers with such automatic model-driven tasks, that I presented above. The transformation aims at keeping the semantics of the transformed modeling languages the same or similar. This enables verifying a simple or composite statechart model with mathematical precision.

Chapter 2 is a brief introduction of the modeling languages that provide state-based modeling features, and which the model transformation builds on.

In Chapter 3, I investigate and summarize the main similarities and differencies of the languages presented in Chapter 2. I present the mapping rules of the transformation that has to be implemented.

The tools I used in the implementation of the transformation as well as some aspects of the transformer is presented in Chapter 4.

In Chapter 5, I show the capabilities of the transformer and the structure of the components generated from SCXML models of an example Crossroads model.

Chapter 6 summarizes the work, and provides tips for future work.

2. Background

2.1. State machine formalisms

State machine formalisms are mathematical models of computation, which describe the behavior of the modeled system or component in an event-driven way. A state machine is a quintuple (I, O, S, s_0, T) where

- I is a finite set of input events or signals
- O is a finite set of output events or signals
- S is a finite non-empty set of states
- s_0 is an initial state, an element of S
- T is a finite set of transitions, which represent changes of states in response to input events, and generate output events

This basic state machine formalism can be extended to statecharts, which support state refinement, concurrent states and extend the broadly understood state space by variables. I introduce three statechart formalisms in this section. [1, pp. 6-7]

2.1.1. UML state machines

The **Unified Modeling Language** (**UML**) [2] is a standard visual modeling language. It is intended to be used for the analysis, design and implementation of software-based systems and to model business processes[2]. UML behavior diagrams represent the dynamic aspects of the system. State-based modeling is supported by state machine diagrams.

There are two types of UML state machines. Behavioral state machine specifies discrete behavior of a part of a designed system through finite state transitions. The state machine formalism is an extended object-based variant of Harel statecharts[3]. Protocol state machine is a specialization of behavioral state machine used to express the usage protocols or the lifecycle of a UML classifier.

---

[2] The Unified Modeling Language, https://www.uml-diagrams.org/

[3] Modeling Reactive Systems with Statecharts: The STATEMATE Approach, (with M. Politi), McGraw-Hill, 1998., http://www.wisdom.weizmann.ac.il/~dharel/reactive_systems.html

2.1.2. SCXML

SCXML (**S**tate**C**hart **XML**) [3] is a W3C[4] technical report used to describe general-purpose event-based state machines. The language is a generalized extension of the state and event notation of CCXML[5], an event-based state machine language designed to support call control features in Voice Applications. The language has XML-like textual syntax and semantics based on Harel statecharts and UML.

SCXML models are defined by SCXML documents. SCXML documents are loaded, parsed and processed by SCXML processors. Execution of a state machine is called an SCXML session. An SCXML processor is a pure event processor that can send and receives events with the help of SCXML event processors. Processors keep track of the internal and external event queue of asynchronous messages, trigger the execution of transitions and maintain the data model of the statechart. SCXML is extendable by custom action elements that can be taken as part of taking transitions. SCXML is a widely extendable language, but also specifies some features that help interoperability. Such features are the ECMAScript datamodel based on the third edition of ECMAScript[6], and the Basic HTTP Event I/O Processor to implement messaging between any kind of web-based service supporting HTTP POST requests. Support for these language features is optional. SCXML processors may support other platform-specific data model languages, or event processors that implement platform-specific message sending.

SCXML is used in Cameo Simulation Toolkit, the simulation toolset of MagicDraw, as part of its extendable model execution framework[7]. There are several SCXML implementations like Apache Commons SCXML[8], uSCXML[9] and Qt SCXML Engine[10]. Because SCXML is a widely used state machine execution platform in

---

[4] World Wide Web Consortium, https://www.w3.org/

[5] Voice Browser Call Control: CCXML Version 1.0, https://www.w3.org/TR/2011/REC-ccxml-20110705/

[6] http://www.ecma-international.org/publications/standards/Ecma-262.htm

[7] https://docs.nomagic.com/display/CST2021x/State+Machine+simulation

[8] Apache Commons SCXML, https://commons.apache.org/proper/commons-scxml/

[9] uSCXML, https://github.com/tklab-tud/uscxml

[10] Qt SCXML Engine, https://doc.qt.io/qt-6/qtscxml-index.html

engineering practice, we should also focus on SCXML modeling toolsets that support semantically correct validation, analysis and code generation of such models.

2.1.3. Gamma Statechart Composition Framework

The **Gamma Statechart Composition Framework** (**Gamma**)[11] [4] is a toolset to model, verify and generate code for component-based reactive systems. Individual statecharts, as well as composite statechart networks can be validated and verified by an automated translation to different backend model checkers. Designers can use the code generation functionality of the framework, which can derive, for example, Java code for the whole system.

Gamma has rigorous semantics concerning composition, communication and execution of components, which is necessary for the validity and conformation of models, generated code, generated tests and the result of formal analysis. It provides strongly-typed model and domain-specific languages for different aspects of modeling composite reactive systems.

2.1.3.1. Gamma Statechart Language

The Gamma Statechart Language (GSL) is an intermediate state machine formalism for the Gamma framework which is the starting point for model-based test generation, source code generation and formal verification of individual statechart components. GSL is configurable thus the developer can set e.g. the priority of different hierarchy levels (top-down or bottom-up scheduling of contained regions), as well as setting the execution order of parallel regions in a statechart. It contains the building blocks of a statechart, so it also provides the basis of modeling composite reactive systems.

2.1.3.2. Gamma Composition Language

The Gamma Composition Language (GCL) supports the definition of communicating composite systems from individual components. Components can behave as communication endpoints by defining ports. Communication of components on the same hierarchy level is made through channels connecting compatible port instances of

[11] The Gamma Statechart Composition Framework is open-source project, whose source code is available at https://github.com/ftsrg/gamma/tree/dev

the connected components. Ports and interface realizations are also strongly-typed. For two ports to be compatible, one has to provide the same or a more specialized interface that the other port requires. Communication hierarchy is established through port binding, where events occurring on a composite system port are forwarded to an appropriate port of a subcomponent of the composite system. The hierarchical composition of components is supported by various component types implementing various execution models, e.g. synchronous or asynchronous composition. [1]

3. Theoretical results

In this section I summarize the main theoretical observations I made considering the syntax and semantics of SCXML and Gamma statecharts. To get more familiar with SCXML, I first compare it with UML state machines, as UML is a very widely known formalism based on which the main concepts of SCXML can be introduced.

After summarizing the similarities and differences between UML and SCXML, I compare SCXML and UML with the Gamma Statechart Language, focusing more on semantic differences. Then I give an overview on how the similar elements of the two languages can be mapped to each other, and how their differences can be resolved if possible.

3.1. Comparing UML and SCXML state machines

SCXML state machines are based on Harel State Tables just like UML and thus share some basic concepts like states, transitions, parallel regions and actions. Generally, SCXML syntax and semantics of these basic concepts are a subset of those found in UML state machines, even if their execution environments are different. [5]

3.1.1. Semantic similarities

3.1.1.1. Transition selection and execution

Some of the most important semantic similarities of them is that transitions are selected and executed the same way in both formalisms. [3] [2]

1. When an event E occurs, the state machine interpreter that executes the state machine, observes the transitions enabled by that event in atomic descendant states of the active state configuration C. A transition T is *enabled* by event E in atomic state S if
    a. T's source state is S or an ancestor of S
    b. T matches E
    c. T lacks a guard expression or its guard evaluates to true.
2. The interpreter next determines the *optimal transition set* of event E in configuration C. This set will contain the largest set of enabled transitions so that no two transitions conflict in the set, and there are no enabled transitions with higher priority outside this optimal transition set. Two transitions T1 and T2

conflict[12] in state configuration C if the intersection of their exit sets is not empty. The *exit set* of a transition in configuration C is consists of all active states in C that are proper descendants of the *least common compound ancestor (LCCA)* of its source and target states[13]. When resolving conflicts, transitions are prioritized: transitions residing in descendant states or preceding other transitions in document order are selected in the optimal transition set.

3. The state machine interpreter executes the transitions in the optimal transition set of event E in configuration C. This is called a *microstep*.

When executing a microstep, the state machine first exits all source states in exit order, and executes exit actions accordingly. Exit order or reverse document order means that descendant states are exited earlier than ancestor states, with reverse document order used to break ties. The statechart processor then executes the effects of the selected transitions themselves in document order, as transitions appear in the document. Finally, the state machine enters the target states in entry order (or top-down order, from parents to children, which is the same as document order) and executes entry actions of these states.

3.1.1.2. Determinism

An SCXML state machine runs deterministically if it does not invoke any external event processor, and if the actions it executes are not programmed to introduce any non-deterministic behavior. Determinism in this context means that the state machine takes the exact same transitions, goes through the same state configuration changes and emits the same sequence of output events every time it is executed with the same sequence of input events.

The way how transitions are selected when an event occurs is designed to resolve situations when more transitions could be selected. They are prioritized. If the source state

[12] Two enabled transitions conflict when they reside in different orthogonal regions of an SCXML <parallel> element, but at least one has a target state that is outside of that region. In this case, executing both transitions might take the statechart in an invalid active state configuration. To prevent this, at most one such transition can be taken. Such conflicting transitions have a non-null intersection of exit state sets. Thus this requirement is caught by examining the intersection of the exit sets of transitions.

[13] This is the case when the type of the transition is 'external'. If a transition is internal and all target states are descendants of the source state S, the exit set of the transition contains only the active proper descendants of S, i.e. S is not part of the exit set in this case.

of two transitions are the same or these states are not descendants of each other, document order specifies which one is selected. When the source states of two conflicting transitions are not the same, but one is descendant of the other, the transition in the descendant state is selected. When there is no matching transition when an event is removed from an event queue, the state machines take no action, and wait for the next event in the queue.

3.1.1.3. Run to completion and termination

Another semantic similarity is the *run-to-completion* execution behavior of microsteps in both formalisms. This means that the state machine does not process an event occurrence until the previous event is processed, its effects are completed and a stable state configuration is reached. Besides that, completion events are generated automatically. A completion event is generated when a simple state has finished executing its entry and do behaviors. Also, in the case of composite or submachine states, a completion event is generated when the given state has reached a final state.

3.1.2. Differences

UML state machines have an explicit structural context in which they are defined. In contrast to this, a single SCXML state machine (denoted by an `<scxml>` element) is the top-level language element. Hence, SCXML state machines have an execution environment that is independent from other SCXML statecharts. They can interact with each other and external services via URIs, for example, in a web-based environment. An SCXML state machine therefore cannot contain any submachine states, but they can instantiate other (external) child SCXML sessions by using the `<invoke>` element. Protocol state machines are not supported either.

SCXML has its own event model. It does not define a typed event model, but uses specific event matching descriptor strings. SCXML's data model is also different from that of UML, and it is customizable: the SCXML state machine and all of its states can define parts of global or local sets of data. An SCXML implementation can support different data models, all of which define the format in which data is stored, the datatypes and the legal operations on the data elements. An implementation can define its own data model and realize it in a platform-specific way.

SCXML supports only a subset of UML pseudo states, namely initial states and shallow or deep history states. There is no '*do*' event in SCXML, but states can have an arbitrary number of `<onentry>` and `<onexit>` handlers, which can contain executable

content, and are executed upon entering or exiting the state, respectively. SCXML also has some specific action elements, but they mostly have general programming equivalents (like the `<foreach>` element).

As mentioned before, completion events are also present in SCXML similarly to UML, but they are also identified by unique names. For example, `‘done.state.<stateId>’` represents the completion event of a `<state>` or a `<parallel>` element with identifier `‘stateId’`. The modeler can use them to explicitly trigger transitions in the state machine, so SCXML gives greater control over handling completion-related constructs than UML.

3.2. Comparing SCXML and Gamma statecharts

As a general observation, SCXML offers some basic state machine elements which aim at state machine interoperability. However, the language also contains highly customizable constructs like the chosen data model, event content structure and external communication methods. These characteristics are in contrast to the precise design languages of Gamma with formal syntax and semantics. Here I compare the syntax and semantics of the most important SCXML and Gamma statechart elements.

3.2.1. Similarities

Both formalisms describe statecharts based on Harel State Tables, therefore the core constructs of SCXML and Gamma are also similar. Gamma statecharts can be configured in several aspects, like how enabled transitions are scheduled and selected, how orthogonal regions are scheduled and executed, and when transition guard expressions are evaluated. There are appropriate configurations that cause the statechart to be executed with UML and SCXML-like semantics.

When transitions are selected for firing in SCXML, transitions from descendant states and ones appearing in the same state, but earlier in document order are selected. In Gamma, these semantics can be enforced by setting the statechart’s scheduling order to bottom-up and setting the transition priority to order-based.

Selected transitions are executed in a microstep, or cycle in both formalisms. First, states in the exit set of all transitions are exited in exit order. Then transitions effects are executed as blocks of actions. As the last step, states are entered in the transitions’ entry set in entry order (from parent states to atomic child states, or otherwise in document order).

Gamma also supports parallel region execution in a compound state, just like UML and SCXML. Orthogonal regions and parallel execution do not introduce real parallelism in SCXML, just denotes all children states of a parallel element are simultaneously active. This affects which transitions are enabled in a state configuration, but the selected ones are executed sequentially in document order (possibly on one execution thread). In Gamma, setting the orthogonal region scheduling order to sequential meets this semantics. In Gamma, it is also possible to set the order of orthogonal regions as unordered, or parallel.

3.2.2. Differences

3.2.2.1. Differences in composition and communication

In SCXML, statecharts are run individually, and communicate with each other by URIs as addresses. In SCXML, external communication is handled by event processors. The main event processor is the SCXML Event I/O Processor, which any SCXML implementation platform must support, and which handles SCXML events for the state machine. These event I/O processors define only the transportation mode and the mapping of event metadata between SCXML sessions and other services. The implementation details of event transmission and the mapping of event content between the sender and the receiver SCXML session are platform-dependent.

Communication between SCXML statecharts is implemented by `<send>` elements, which define the target URI, SCXML event processor, the optional time delay, and the event content sent to the receiving SCXML session or other web service as payload. The event is sent by the ‘fire and forget’ concept. The format of the payload depends on the data model, and the delivery depends on the selected event processor, both of which can have platform-dependent parts. The content sent in the event is not strongly-typed.

In Gamma, multiple components can be integrated in a composite model using the semantically well-defined Gamma Composition Language. Hierarchic nesting of components is also supported.

Gamma establishes communication between components in a different manner. In a Gamma composite model, each component has its own ports, which serve as communication endpoints, and communication can only happen through these ports. Each port realizes an interface, which interface in turn defines the receivable and transmittable

events. The events are directed (by ‘in’, ‘out’, or ‘inout’ directions), and can have typed payload parameters, e.g. a string or an array of integers.

Communication between ports is done by channels connecting two compatible ports. Port compatibility means that one of the ports provides an interface with in and out events, and the other one implements an interface with the subset of the same events but with opposite directions. Also, in Gamma, a subcomponent’s port can only be connected to another port with a channel, and only if they are at the same hierarchy level. Nested components can only communicate through their ancestors’ matching ports by port binding.

In SCXML, when events are sent or are set as triggers of transitions, they are referenced by event descriptor strings. Such examples are ‘*’, ‘count’, ‘count.add’ or ‘count.add.zero’. Event matching on selecting transitions is done by splitting the trigger’s event descriptors into tokens by dot characters. An event descriptor matches such an event name, if it is the same or its prefix (e.g. incoming event name ‘count.add.one’ would match descriptors ‘*’, ‘count’ or ‘count.add’, and the transition would be enabled, however, it would not match ‘count.add.zero’).

3.2.2.2. Differences in the data model, actions and expressions

In SCXML, the state machine has a data model, and it can have <data> elements like variables. The interpretation of those data all depend on the data model set in the state machine. Examples are

- boolean expressions of transition guards
- data model’s value expressions
- data manipulating actions executed in entry, exit actions and when taking transitions.

Data elements can be complex objects with a possible XML, JSON, plain text or a platform-specific notation.

Gamma provides constants and variables that a statechart can handle. Constants are set at instantiation time, but the values of variables can be changed. The variables and constants all have a type, which comes from a limited, but complex enough datatype set (array and record objects are supported, and can be nested to create complex typed

datatypes). Mapping simple SCXML data elements to Gamma variables can even be done by the Gamma Expression Language's automatic type recognizer feature.

An SCXML `<raise>` action corresponds to a Raise Event Action in Gamma, but we have to create an Event object and specify the interface and port on which we send out that event. `<send>` actions represent sending an event wrapped in an asynchronously sent message, which can also be mapped to Raise Event Actions. The SCXML event name is mapped to a Gamma Event object. The target of the send action can be mapped to a Gamma Instance Port Reference, representing a port of an instance of a component type. The name of the interface and the port has to be inferred from the target.

3.2.2.3. Differences in execution-related elements

An SCXML transition defines the target state specification to which the transition brings the statechart after it is executed. This target state specification can contain multiple `<state>` and `<parallel>` element identifiers, and allows for greater control over specifying the exact target state configuration. The next active state configuration will be deduced by expanding this state specification by initial and history states of compound states, parallel regions and their successive subregions. In Gamma, the state configuration specified by the transition target is determined the same way as in SCXML, but only a single target state can be specified as a target state specification.

A syntactical difference is that SCXML transition targets are given by state identifier strings (such as event descriptors). Another syntactical difference is that in SCXML, the initial state of a state can be given by an `<initial>` element, or an initial attribute. If neither is specified then the first state in (top-down) document order is the initial state in that compound state.

Execution is different considering final states and completion events. SCXML offers referencing named completion events (and also error events) explicitly. Transitions implicitly triggered by such completion events are prioritized over explicit event triggers, therefore event execution order depends on execution time.

Gamma does not support completion-related constructs, because the timed behavior and completion events make the state space unmanageable and hard to formally analyze.

3.3. Mapping between SCXML and Gamma elements

Many aspects of the basic state machine concepts of the two languages can be mapped easily, though they have semantically quite different constructs as well. Where mapping the different syntactical and semantical aspects was not trivial, I chose a solution that can fit in the Gamma framework and which aims to keep the semantics of the original language. When I could not resolve a difference, I gave the SCXML modeler customization possibilities for the model as well as set some restrictions that the model transformation will expect. These custom SCXML modeling constructs still can be translated into a Gamma statechart, but partially support those SCXML features they are created with.

3.3.1. Core statechart elements

Table 3-1 contains the core SCXML statechart elements and their corresponding Gamma model elements. The name of the <scxml> element is mapped to the name of the Gamma statechart. The identifiers (*id* attributes) of `<state>`, `<parallel>` and `<history>` elements become the name of the corresponding Gamma statechart element. These identifiers have to be unique in the statechart.

Element	SCXML element	Gamma element
Statechart	<scxml>	*Synchronous Statechart Definition*
State	<state> -id	*State* - contains a *Region* with an *Initial State* if it is compound
Parallel regions	<parallel>	*State* with more regions containing the mapped children of the <parallel> element
Initial state	- <initial> element - initial attribute of <scxml> or <state> - first child state in document order of <scxml> or <state>	*Initial State* - *Transition from the initial pseudo state if not an SCXML <initial> element*
History (shallow / deep)	<history> with *type* = 'shallow' / 'deep'	*Shallow History State* / *Deep History State*
Entry behavior	<onentry>	*Action* block containing mapped individual child actions of <onentry>
Exit behavior	<onexit>	*Action* block containing mapped individual child actions of <onexit>
Transition	<transition> - *source* - *target* - *trigger* - *guard* - effects (children actions)	*Transition* • *containing State Node* • *target State Node, name = target id* • *Port-Event Trigger* • *(Boolean)Expression* • *Action block containing mapped <assign>, <raise>* actions etc.
Variable	<data> with implicit type of boolean, integer, string or an array of one of these types	*Variable Declaration* • resolving its *type* automatically if 'gamma' data model is used
Assign	<assign>	*Assignment Statement*
Raise event	<raise>	*Raise Event Action*

Table 3-1: Corresponding SCXML and Gamma statechart elements

3.3.2. Action and data model elements

Element	SCXML element	Gamma element
Variable	`<data>` with implicit type of boolean, integer, string, array, record or other valid Gamma type	*Variable Declaration* • resolving its *type* automatically if `'gamma'` data model is used
Assign	`<assign>`	*Assignment Statement*
Raise event	`<raise>`	*Raise Event Action*

Table 3-2: Corresponding SCXML and Gamma action and data model elements

The datamodel of the SCXML statecharts have to be set to `'gamma'` to use the platform-specific Gamma action and expression elements. Other datamodel specifications are not supported by the transformation project. Besides history states, an SCXML statechart has inner memory by comprising a data model. When using the Gamma datamodel, SCXML `<data>` elements with expressions of supported Gamma types can be specified. These `<data>` elements are mapped to variables with the name of the unique identifier of `<data>` and the value of the parsed expression of `<data>`.

Like transition guards, variables can be initialized and updated with the evaluated values of Gamma expressions. The expression transformer parses the textual Gamma expression tree from the `expr` attribute. It also links all variable names to their corresponding variable declarations, so that evaluation of the parsed expression will use the actual value of the variable instead of its name.

3.3.3. Composition and communication elements

Element	SCXML element	Gamma element
Composition / invoking services	SCXML model with `<invoke>` elements	*Scheduled Asynchronous Composite Component*
Message sending	`<send>` -*target* attribute	*Raise Event Action* `-InstancePortReference`
Event	-implicitly defined by transitions, `<raise>` or `<send>` actions, or `<data>` with implicit type of boolean, integer, string or an array of one of these types	*Event* *EventDeclaration* with the proper event direction: *internal* or *out*
Interface	Implicitly defined by event string or by a top-level <data> element representing a port	*Interface* and *EventDeclaration*s
Port	Implicitly defined by event string or top-level `<data>` element with id starting with `'pro_port_'` or `'req_port_'`	*Port* and *InterfaceRealization*
Port binding	top-level `<data>` element with id starting with `'binding_'`	*Port Binding*
Channel	top-level `<data>` element with id starting with `'channel_'`	*Channel*

Table 3-3: Corresponding SCXML and Gamma composition and communication elements

In SCXML, the `<invoke>` element creates an external service when executed. It allows for more coupled communication between a parent SCXML session and a child service than `<send>`, but less tightly coupled than the composition and communication in Gamma. In this work, I only handle the cases when the child service is also an SCXML statechart, to stay in the domain of the SCXML-Gamma transformation.

I transform the individual statecharts of the composite SCXML model, then compose them into a composite Gamma component. For the composition semantics I chose the scheduled asynchronous model of computation. The corresponding Gamma component type is also called *Scheduled Asynchronous Composite Component*.

Individual statecharts process events asynchronously. An SCXML interpreter uses an internal message queue for events raised and targeted inside that SCXML session, and an external message queue for storing messages coming from external services.

An event in the SCXML model is defined implicitly by its name. During transformation, a Gamma Event object is created with this name. Because there are no elements in SCXML that directly define the interfaces, ports and contracts that define communication between components, I changed the semantics of the structure of event descriptors for the purpose of modeling a transformable SCXML statechart. Interface names and port names can be defined in event descriptors with a special syntax, with the option of omitting one or both (See Section 3.3.4.2). The names of events and interfaces are considered to be global in the context of the transformation. Where the same event or interface name is found, they are mapped to the same Gamma Event or Interface object, respectively when transforming any SCXML model.

Event declarations on the Gamma interface must have an event direction specified. The special direction internal is used for events that are sent to the same component as the sender. The default interfaces of transformed statecharts contain only this kind of events, the default interfaces and ports becoming internal automatically. All other events are directed as out events. This is not a problem, because in events are received by ports that realize interfaces in required mode, so these out events become in events on them. This way SCXML events on can be mapped to events and event declarations on a strongly-typed Gamma interface, and they can be sent and received through ports.

3.3.4. Resolving differences between SCXML and Gamma statecharts

There are some constructs in SCXML that cannot be easily mapped into Gamma. Completion events and event descriptors are two examples.

Handling completion events in an environment that does not support them is nbot trivial. It would require additional states, transitions, and another message queue with higher priority than the one queueing normal event occurrences. Implementing this feature and mapping corresponding SCXML states and events would result in a complex statechart model with a state space too complex for an SCXML model, so I did not focus on resolving it in this work.

3.3.4.1. Choice of data model and expression language

Parsing expressions from different data models (such as ECMAScript, Lua or XPath), would need support for their data representation, expressions and actions manipulating the data. This was out of the scope of the core transformation. Instead of

supporting these languages and creating a transformation just for the optional datamodel language, I made a restriction for the input SCXML models. They have to use a custom data model named `'gamma'`, which uses the Gamma Expression Language for its conditional expressions and value expressions. This is a reasonable choice because the Gamma Expression Language provides the common simple and complex datatypes, expressions, variables, logic operators, arithmetic operators and selectors that can conveniently model a broad set of expressions occurring in an expression modeling language, while enabling the formal analysis of these expressions. Besides that, the framework used in the implementation support the parsing of these expressions out-of-the-box (See Chapter 4).

3.3.4.2. Handling event descriptors

In the case of event descriptors, SCXML uses event strings and prefix matching at transition triggers. Because of this prefix matching strategy, there can be infinitely many possible matching events for an SCXML event descriptor in an SCXML transition list, which Gamma cannot handle. Gamma does not support trigger name prefix matching. Because of that, the model transformer handles only exact event names, not event name prefixes.

To allow SCXML developers to specify a convenient and more elaborated model of events, I decided to support three kinds of event description modes. Event name tokens are separated by single dots:

- `'port.interface.event'`: Port, interface and event names are all specified. In the transformed Gamma model, the event will be transmitted on the specified port realizing the interface. This allows for differentiation of multiple port instances on a statechart providing the same interface.
- `'interface.event'`: Events with the specified event name will be transmitted through the interface specified in the name as its first token. The transformer automatically creates a default port for this interface, which implements this interface as provided. This default port of the named interface is different from named ports specified with the `'port.interface.event'` description mode.
- `'event'`: Specifies only the event name. To handle this event description mode in Gamma, the transformer automatically creates an interface, which will serve as the default interface for the Gamma statechart, and also a port providing this default interface, which will be its default port. The default interface and port are different from all other interfaces and ports of the same statechart. For the transition triggers and raise event actions that receive or send this type of event (i.e. events with exactly

this name), the event will be sent out and received through this default port and interface.

3.3.4.3. Composition with <invoke> elements

To model the composition of SCXML statecharts, I included transforming an SCXML statechart with <invoke> elements to a composite Gamma component. The `<invoke>` element represents a service that is invoked when the SCXML session has reached a stable state, and cancelled when the state containing the `<invoke>` is deactivated. The structure of composite models in Gamma cannot be changed dynamically in this sense. To model similar execution semantics in SCXML that Gamma composite models support, the SCXML modeler should place all invocations in a state that is always active while the SCXML statechart is being executed. The transformer currently supports hierarchical invocations for one level depth, with at most one composite statechart. Statechart-like semantics can only be specified in atomic statechart models, which do not contain any `<invoke>` elements.

The `src` attribute of an `<invoke>` element defines the SCXML file that contains the markup of the invoked statechart. Thus, the transformer tries to find and load the contents of that file when transforming `<invoke>`. Multiple invocations may refer to the same SCXML statechart file, so these SCXML sources can be mapped to statechart definitions in Gamma. Invocations are then mapped to instances of this statechart component type.

The transformer composes the subcomponents based on the scheduled asynchronous composite model of computation [1, p. 9]. I chose this model because individual SCXML sessions are executed independently, just as components in an asynchronous-reactive model of computation.

To support this computation model, I wrapped Synchronous Statechart Definition instances into Asynchronous Adapters, that control the wrapped statecharts asynchronously. The adapters communicate using message queues to store messages from other services. They are responsible for storing incoming messages in message queues. When a message arrives, they extract the event from it, and make their wrapped state machine execute it. The queues follow first-in-first-out semantics. While the external event queue is empty, it is blocking, but when a message arrives in that queue, a new execution cycle – a macrostep – starts in the SCXML session. The internal event

queue has greater priority than the external. While there are internal events generated inside the statechart, external events are not removed or executed from the external event queue. The schedulable asynchronous component is used because its semantics is similar, but more general than an asynchronous composite component, since the execution list of its subcomponents can be specified or refined.

3.3.4.4. Handling top-level <data> elements describing communication

Since SCXML lacks the precise notion of interfaces, ports, port bindings and communication channels, a modeler should be able to specify these elements in an SCXML model, too. To achieve this, the transformer investigates a composite statecharts' top-level `<data>` elements with the special identifiers of the SCXML-Gamma transformation platform. The transformer assumes that some top-level SCXML `<data>` elements will be responsible to represent these communication elements.

- *Composite system ports:* To make it possible to explicitly model ports communicating with the environment, composite system ports can be defined by a top-level <data> element with id that starts with the string `'pro_port_'` or `'req_port_'`. With these identifiers, the modeler can specify a provided or required port, respectively. The expr attribute of the port <data> has to be set to `[<port_name>].<interface_name>`. The name of the port can be omitted, in this case it will be the same as the interface name. Interface names are considered global in the modeled composite component and its subcomponents.

- *Port bindings:* Composite system ports can be bound to a port of a subcomponent instance by port binding. A binding can be specified by a top-level <data> element with id starting with the string 'binding_'. The expr should be in this format: `'<system_port_name> - <instance_name><port_name>'`. This will tell the transformer to bind the system port to the specified port of a component instance. The ports' interfaces have to be compatible for binding.

- *Channels:* A top-level `<data>` element represents a channel if its id starts with 'channel_'. The expr attribute shall contain a channel descriptor with the format *'<source_port>.<source_interface> - <target_port>.<target_interface>'*. A channel will be created during transformation that connects the compatible source and target ports.

4. Implementation

4.1. Technologies

I used the Eclipse[14] platform as the development environment for the SCXML-Gamma statechart transformation projects. For handling the abstract syntax of the language models, I used Eclipse Modeling Framework (EMF)[15]. This is a reasonable choice because the Gamma framework itself is written as a set of Eclipse plugin projects, and other model transformers in Gamma also use this Eclipse-based modeling framework.

I also introduce Xtext technology because the languages in Gamma are built upon it, and I use the Gamma Expression Language for parsing transition guard and variable value expressions. Xtend helped me write easy-to-read transformation source code.

4.1.1. Eclipse Modeling Framework

The EMF project is a modeling framework and code generation facility for building tools and other applications based on a structured data model. From a model specification described in XMI, EMF provides tools and runtime support to produce a set of Java classes for the model, along with a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor.

The framework includes Ecore meta models which describe the abstract syntax of a modeled language. In an Ecore meta model, EClass, EAttribute, EReference, EEnum, ELiteral, EOperation metamodeling elements can be used to capture the abstract conceptual model and logical connections of the concrete language elements. The framework also uses Genmodel files which hold code generation settings for a model, and by which the framework provides automatic code generation features (e.g. Java code).

4.1.2. Xtext

Xtext[16] is a language engineering framework. Programming languages and domain-specific languages can be developed with Xtext by using a powerful grammar language. The meta models of defined languages are Ecore models, as Xtext is based on

[14] Eclipse homepage: https://www.eclipse.org/

[15] Eclipse Modeling Framework (EMF), https://www.eclipse.org/modeling/emf/

[16] Xtext homepage, https://www.eclipse.org/Xtext/

EMF. With Xtext, we define the concrete syntax of our language as well as how it is mapped to its semantic Ecore model. As a result, Xtext provides a full infrastructure, including parser, linker, type checker, compiler and editing support for Eclipse.

4.1.3. Xtend

Xtend[17] is a high level statically-typed programming language, which translates to comprehensible Java source code. Xtend is built in Xtext, and has its syntactic and semantic roots in the Java programming language. Xtend extends and improves Java in many aspects. Extension methods, lambda expressions, dispatch methods (for polymorphic function invocation) and type inference are some of the most important features of the language that improve readability, maintainability and convenient development of the source code.

4.2. Implementation of the SCXML-Gamma transformation

Both SCXML and Gamma has an Ecore meta model. This makes the transformation possible in EMF. I found a suitable SCXML meta model[18] on GitHub, which provides the language elements in an EMF Ecore model and contains SCXML editor features. The models and languages of Gamma also provide their Ecore meta models (as the framework was built upon EMF).

To implement the model transformation, I created two Eclipse plugin projects. The task of the first plugin is executing the actual statechart transformation. The other plugin is an Eclipse command handler, which is used to process Eclipse UI interactions in an Eclipse instance and start the SCXML-Gamma transformation project code for a selected file containing an SCXML model. I included the meta model packages of both languages in the transformation project. I traverse the whole SCXML Ecore model in multiple phases, create and set corresponding Gamma expression, action, interface, statechart and composition language elements, and store mapped model elements in traceability objects. I defined a traceability type for holding data for an individual statechart component, and another type to handle composite statechart models, (containing instance type traceabilities as well). Execution of the statechart transformer returns the appropriate traceability object to the caller. The Eclipse command handler then

[17] Xtend homepage, https://www.eclipse.org/xtend/

[18] SCXML Ecore meta-model in EMF: https://github.com/eventB-Soton/SCXML_EMF

serializes the Gamma statechart and its interface into two separate Gamma files with the textual syntax of Gamma.

The transformation ends with this step. The generated Gamma models can be used to generate statechart and composite component Java code, test suites, as well as create a formal composite model and execution traces using a verification backend that Gamma provides.

4.2.1. Traversing an SCXML statechart

I transformed SCXML models by first transforming the state containment hierarchy, then transforming other cross references in the model between states. Such cross references are transitions and initial states in this transformation algorithm. While I transform states and parallel regions, I also transform events and their interfaces and ports, if needed. I did the transformation of interfaces and communication elements in the manner of 'get or transform', while storing already transformed elements in the statechart traceability object.

The root `<scxml>` element with `<state>`, `<parallel>` and `<final>` elements form a tree of states, because their containment graph is an SCXML document tree, which cannot contain cycles. Therefore, this state hierarchy forms the framework of the state machine. This is the first phase of statechart transformation. After I transform these state nodes, SCXML transition source and target states are already transformed, so the transformed Gamma transition source and target state node references can be set to already existing Gamma states. SCXML `initial` attributes of the root element or compound states can also be transformed then, since the target Gamma state of the transition from this initial element is already known.

While traversing states, I also transform entry and exit actions. While transforming transitions, I also transform its executable content.

4.2.2. Traceability classes

The `StatechartTraceability` Xtend class of the transformation project is responsible for storing an SCXML statechart's elements and their mapped Gamma model elements which are important to be stored while the transformation runs. One can set the SCXML root element in such a traceability object during creation, and get its corresponding Gamma synchronous statechart definition and its generated interfaces after statechart transformation is executed.

`StatechartTraceability` contains maps for statecharts, states, transitions, ports and interfaces – including default ports and interfaces for the statechart –, and variable mappings. There are put~, get~ and contains~ methods for the proper element types to change the content of the traceability object. States can also be searched by their string identifiers, because we need such a function when transforming an SCXML transition's target state. I store ports and interfaces by their given or generated name tokens, because there are no matching pairs for them in the SCXML state machine.

`CompositeTraceability` holds mapping data for a composite SCXML model with invoked statechart types. It traces the sources, definitions and transformed Gamma statechart definitions of the types of instances. It also traces the instantiated scheduled asynchronous composite component, and its subcomponents, which are the transformed statechart wrapper instances. Interfaces, constant declarations and events are collected in composite component-scoped hashed maps, then serialized into one declarations file.

4.2.3. Transformer classes

I applied the principle of 'separation of concerns' on the transformation project. I put the transformation functions of basic SCXML state machine elements in the `ScxmlToGammaStatechartTransformer` Xtend class, and the transformation of the composite components is put into the `ScxmlToGammaCompositeTransformer` class. I extracted and encapsulated other aspects of transformation into another classes, which hold their set of transformation functions for that given set of state machine concepts. Such classes are Action~, Data~, Event~, Interface~, Port~ and TriggerTransformer. For example, transformation of events occurs both when transforming transition triggers (by creating a reference to a pair of port and event) and when transforming raise event actions. Therefore, PortTransformer acts as an API to the statechart transformer and the action transformer classes.

I also had to resolve referenced variables in parsed Gamma expressions. These expressions appear as transition guards or the value expression in `<assign>` elements. In the Scxml-Gamma expression language parser class I iterate over the variable declarations parsed by the Gamma Expression Language parser, and set the corresponding variable declarations in those parsed Xtext reference expressions.

5. Evaluation

I tested the implemented transformer with an SCXML model of the composite Crossroads component from the Gamma tutorial[19].

5.1. Example: Crossroads

The Gamma tutorial presents a traffic lights controller of a crossroad, where there are two roads intersecting each other (Figure 5-1). Traffic lights are standard 3-phase lights looping through the red-green-yellow-red sequence. The police can trigger an interrupted mode in which the traffic lights are blinking yellow.

The controller is divided into three submodules. Each submodule's behavior is described by statecharts. Two traffic light controller submodules are controlling the traffic lights in each direction, and a single crossroad controller component is responsible for the coordination of the flow of traffic.

The transformer I implemented is capable of transforming either submodule or the whole system to an asynchronous Gamma component, or transforming the composite component as a whole. I show some relevant parts of the original and translated models in this documentation. I use the Traffic Light Control component as an example for a transformation of an individual statechart. Then I show how I transformed the Crossroads composite component to Gamma.

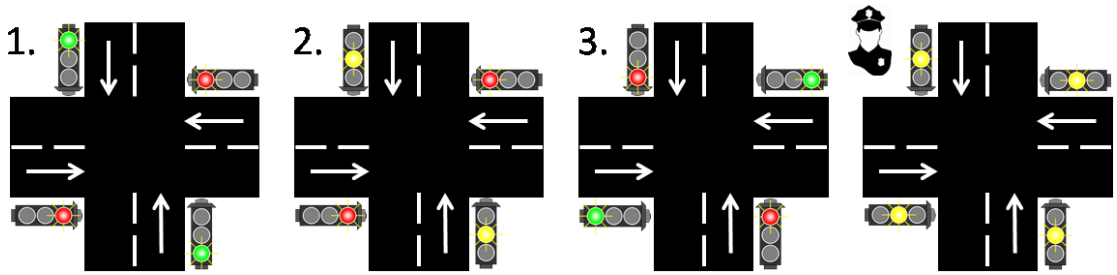

Figure 5-1: Traffic lights of a crossroad presented in the Gamma tutorial

5.1.1. Transforming a statechart component

I first translated the statecharts of the subcomponents I present this phase of the transformation on the traffic light controller.

[19] Gamma tutorial, https://github.com/ftsrg/gamma/tree/master/tutorial

5.1.1.1. SCXML model of a traffic light controller

I translated the Gamma statechart of the traffic light controller submodule to an SCXML state machine manually. The only difference is that the transformer cannot handle timeouts used in the interrupted mode of the lights (the `<send>` element is needed to simulate timed behavior by delaying events). Instead of timeouts, I put explicit timeout events in the model to trigger light change when blinking. I did not specify the port on which these events would be received, so the transformation should transmit them on the default port of the Gamma statechart.

Some relevant parts of the SCXML model of the crossroad (parts not included here are replaced with triple dots):

```
<?xml version="1.0" encoding="UTF-8"?>
<scxml:scxml xmlns:scxml=http://www.w3.org/2005/07/scxml
      initial="Normal" name="TrafficLightCtrl">
  <scxml:state id="Normal" initial="Entry2">
    …
  </scxml:state>
  <scxml:state id="Interrupted">
    <scxml:initial>
      <scxml:transition target="BlinkingYellow" type="external"/>
    </scxml:initial>
    <scxml:state id="BlinkingYellow">
      <scxml:onentry>
        <scxml:raise event="LightCommands.displayYellow"/>
      </scxml:onentry>
      <scxml:transition        event="BlinkingYellowTimeout"        target="Black"
type="external"/>
    </scxml:state>
      ...
    <scxml:transition        event="PoliceInterrupt.police"        target="Normal"
type="external"/>
  </scxml:state>
</scxml:scxml>
```

Figure 5-2: Parts of the SCXML document of a Traffic Light Control component

5.1.1.2. Transformed Gamma model of the Traffic Light Control component

Execution of the SCXML-Gamma statechart transformation produced the two Gamma files required. `TrafficLightCtrlDeclarations.gcd` contains the `LightCommands` interface to handle traffic lights, the Control interface to receive light toggle commands from the crossroad controller, the `PoliceInterrupt` interface to switch between normal and interrupted operation modes, and the default component interface to receive timeout events when blinking.

Here are the transformed interfaces in the generated `TrafficLightCtrlDeclarations.gcd`. Note that the default interface contains only

internal events, and all other interfaces contain only out directed events. This declarations package also contains a constant that is used when setting default message queue capacities in asynchronous adapters.

```
package trafficlightctrl_interfaces
const QUEUE_CAPACITY : integer := 4
interface LightCommands {
	out event displayRed
	out event displayGreen
	out event displayYellow
	out event displayNone
}
interface Control {
	out event toggle
}
interface PoliceInterrupt {
	out event police
}
interface TrafficLightCtrl_DefaultInterface {
	internal event BlinkingYellowTimeout
	internal event BlackTimeout
}
```

Figure 5-3: The interfaces and constant declaration of the transformed Traffic Light Control component

`TrafficLightCtrl.gcd` contains the transformed Gamma statechart definition and its asynchronous wrapper component. The transformer put in a Gamma package, which package imports another package that contains the common interfaces and other declarations needed for the component.

The statechart maps the core SCXML execution elements of the original model, like states, transitions and event raising actions. Note the necessary execution-related annotations on the statechart that configure aspects of the statechart execution semantically the same as the SCXML model.

The asynchronous adapter defines the internal and external message queues that correspond to the message queues of an SCXML session. Appropriate incoming events of ports are set to trigger execution of the wrapped statechart when the adapter receives a message. Event references tell how the events are selected by the queues of different priorities.

```
package trafficlightctrladapter
import "/hu.bme.mit.gamma.scxml.examples.crossroads/model/
    TrafficLightControl/TrafficLightCtrlDeclarations.gcd"
adapter TrafficLightCtrlAdapter of
     component TrafficLightCtrl : TrafficLightCtrl
{
```

```
        when Control.toggle / run
        when PoliceInterrupt.police / run
        queue TrafficLightCtrlInternalEventQueue
              (priority = 2, capacity = QUEUE_CAPACITY) {
              TrafficLightCtrl_DefaultPort.BlinkingYellowTimeout,
              TrafficLightCtrl_DefaultPort.BlackTimeout
        }
        queue TrafficLightCtrlExternalEventQueue
              (priority = 1, capacity = QUEUE_CAPACITY) {
              Control.toggle, PoliceInterrupt.police
        }
}
```

Figure 5-4: Asynchronous adapter of TrafficLightCtrl that enables using the statechart model in an asynchronous composite component. It has a message queue for external events, and one for the internal statechart events with higher priority.

```
@RegionSchedule = bottom-up
@TransitionPriority = order-based
@GuardEvaluation = beginning-of-step statechart TrafficLightCtrl [
      port          TrafficLightCtrl_DefaultPort          :          provides
TrafficLightCtrl_DefaultInterface
      port LightCommands : provides LightCommands
      port Control : requires Control
      port PoliceInterrupt : requires PoliceInterrupt
] {
      region TrafficLightCtrlRegion {
            state Normal {
                  region NormalRegion {
                        state Red {
                              entry / {
                                    raise LightCommands.displayRed;
                              }
                        }
                        ...
                        shallow history Entry2ShallowHistory
                        initial NormalInitial
                  }
            }
            state Interrupted {
                  ...
            }
            initial TrafficLightCtrlInitial
      }
      transition from Entry2ShallowHistory to Red
      transition from Red to Green when Control.toggle
      ...
      transition from Interrupted to Normal when PoliceInterrupt.police
      transition from TrafficLightCtrlInitial to Normal
      transition from NormalInitial to Entry2ShallowHistory
}
```

Figure 5-5: Relevant parts of the transformed Traffic Light Control Gamma statechart. Note the execution-related annotations on top of the TrafficLightCtrl statechart definition.

5.1.2. Transforming the composite model

5.1.2.1. Presenting the composite SCXML statechart model

The composite model contains an SCXML statechart with the <invoke> elements that represent subcomponent instantiation. They are placed in state Invocation under a Working <parallel> element to be run always while the statechart runs. Three composite system ports are defined on this composite component with appropriate bindings to the instantiated Controller and Traffic Light Control components. The two Traffic Light control components share the same source URI, so their corresponding Gamma component instances are instances of the same transformed asynchronous adapter component type. The three submodules are at the same hierarchy levels, and their appropriate ports are connected with channels.

```
<scxml:scxml
     xmlns:scxml=http://www.w3.org/2005/07/scxml
     datamodel="gamma" initial="Working" name="Crossroads">
  <scxml:datamodel>
    <scxml:data expr="police.PoliceInterrupt" id="req_port_1"/>
    <scxml:data expr="priorityOutput.LightCommands" id="pro_port_1"/>
    <scxml:data expr="secondaryOutput.LightCommands" id="pro_port_2"/>
    <scxml:data expr="police - controller.PoliceInterrupt" id="binding_1"/>
    <scxml:data expr="priorityOutput - prior.LightCommands" />
    <scxml:data      id="binding_2"
          expr="secondaryOutput - secondary.LightCommands" id="binding_3"/>
    <scxml:data      id="channel_1"
          expr="controller.PriorityControl - prior.Control" />
    <scxml:data      id="channel_2"
          expr="controller.SecondaryControl - secondary.Control" />
    <scxml:data      id="channel_3"
          expr="controller.PriorityPolice - prior.PoliceInterrupt" />
    <scxml:data      id="channel_4"
          expr="controller.SecondaryPolice - secondary.PoliceInterrupt" />
  </scxml:datamodel>
  <scxml:parallel id="Working">
    <scxml:state id="Invocation">
      <scxml:invoke  id="controller"
          src="hu.bme.mit.gamma.scxml.examples.crossroads/model/
                    Controller/Controller.scxml"/>
      <scxml:invoke  id="prior"
          src="hu.bme.mit.gamma.scxml.examples.crossroads/model/
                    TrafficLightControl/TrafficLightCtrl.scxml"/>
      <scxml:invoke  id="secondary"
          src="hu.bme.mit.gamma.scxml.examples.crossroads/model/
                    TrafficLightControl/TrafficLightCtrl.scxml"/>
    </scxml:state>
    <scxml:state id="Operating"/>
  </scxml:parallel>
</scxml:scxml>
```

Figure 5-6: The SCXML model of the Crossroads component. It composes Controller and TrafficLightCtrl components by instantiating and connecting them.

5.1.2.2. The transformed Crossroads component

Two files are generated when applying the SCXML-Gamma transformation to the composite component, too. CrossroadsDeclarations.gcd contains the global interfaces and declarations of the components, basically the same as the ones contained in TrafficLightCtrlDeclarations.gcd, completed with the default interfaces of the Controller module. The transformed scheduled asynchronous composite component resides in Crossroads.gcd.

```
package crossroads
import
"/hu.bme.mit.gamma.scxml.examples.crossroads/model/CrossroadsDeclarations.
gcd"
scheduled-async Crossroads [
	port police : requires PoliceInterrupt
	port priorityOutput : provides LightCommands
	port secondaryOutput : provides LightCommands
] {
	component controller : ControllerAdapter
	component prior : TrafficLightCtrlAdapter
	component secondary : TrafficLightCtrlAdapter
	bind police -> controller.PoliceInterrupt
	bind priorityOutput -> prior.LightCommands
	bind secondaryOutput -> secondary.LightCommands
	channel [ controller.PriorityControl ] -o)- [ prior.Control ]
	channel [ controller.SecondaryControl ] -o)- [ secondary.Control ]
	channel [ controller.PriorityPolice ] -o)- [ prior.PoliceInterrupt ]
	channel [ controller.SecondaryPolice ] -o)- [secondary.PoliceInterrupt
]
}
```

Figure 5-7: The transformed Crossroads component with its ports, component instances, port bindings and channels.

5.1.3. Verification of the transformed Crossroads component

After transformation, I initiated the Gamma framework to create the formal analysis model of the composite component. I also specified a Gamma generator file in which I asked Gamma to generate state coverage traces and generate test cases from these. The verification process resulted in traces that show the reachability of the states of the Crossroads component by raising events and taking transitions.

```
import "/hu.bme.mit.gamma.scxml.examples.crossroads/model/.Crossroads.gsm"
trace CrossroadsTrace of Crossroads
step {
	act {
		reset
	}
	assert {
		raise priorityOutput.displayRed();
		raise secondaryOutput.displayRed();
		state controller_Controller.ControllerRegion.Operating;
		state controller_Controller.OperatingRegion.Init;
		state prior_TrafficLightCtrl.TrafficLightCtrlRegion.Normal;
		state prior_TrafficLightCtrl.NormalRegion.Red;
		state secondary_TrafficLightCtrl.TrafficLightCtrlRegion.Normal;
		state secondary_TrafficLightCtrl.NormalRegion.Red;
	}
}
step {
	...
}
```

Figure 5-8: Beginning of a trace file generated as part of the state coverage verification process on the generated Crossroads component

6. Conclusion

Model-driven development tools speed up the development process by allowing system architects thinking in models, and provide verification and automatic code generation capabilities based on these models. The internal behavior of systems and their components is usually represented using state-based formalisms.

To help modelers use model-driven features with precise semantics on SCXML models, I designed and implemented a model transformer that translates SCXML state machines to the Gamma Statechart Composition Framework. By this transformation, the automatic model checking, code generation and model-based test generation features can be used on transformed SCXML models. To ensure semantic equivalence of transformed elements, I compared the two formalisms to each other both syntactically and semantically.

I used Eclipse EMF and the Xtext framework to catch the abstract syntax elements for transforming models and parsing languages (e.g. the Gamma Expression Language). I developed statechart traceability and transformation classes using the Xtend language. The transformer is capable of mapping most elements of an individual SCXML state machine to Gamma model elements. I tested the capabilities of the transformer on a crossroad controller included in the tutorial project of Gamma.

This model transformer in itself can translate individual SCXML state machines or composite SCXML system models with one level of subcomponents. For future work, it should be extended with the validation of the model to be transformed, clearing and extending the options how non-trivial mappings can be modeled, and adding support for the transformation of more SCXML elements, including elements which are more challenging to map to Gamma in a semantically correct way.

Bibliography

[1] B. Graics, "*Model-Driven Design and Verification of Component-Based Reactive Systems*" 2016. [Online]. Available: https://inf.mit.bme.hu/sites/default/files/gamma/documents/BSc2016_Graics.pdf. [Accessed 09-12-2022].

[2] Object Management Group, "*Unified Modeling Language*" Object Management Group, [Online]. Available: https://www.omg.org/spec/UML. [Accessed 09-12-2022].

[3] World Wide Web Consortium, "*State Chart XML (SCXML): State Machine Notation for Control Abstraction*" World Wide Web Consortium, 01-09-2015. [Online]. Available: https://www.w3.org/TR/scxml/. [Accessed 09-12-2022].

[4] B. Graics, V. Molnár, A. Vörös, I. Majzik and D. Varró, "*Mixed-semantics composition of statecharts for the component-based design of reactive systems*" *Software and System Modeling,* vol. 2020, no. 19, pp. 1483-1517, 01-07-2020.

[5] B. Selic, "*General Differences Between SCXML and UML*" 16-01-2015.