This is to certify that the
thesis entitled

EXTENDING AMALIA TO GENERATE ANALYSIS
COMPONENTS FOR STATECHARTS

presented by

Chad R. Meiners

has been accepted towards fulfillment
of the requirements for the

MASTER degree in COMPUTER SCIENCE

[illegible]
Major Professor's Signature

5/7/2004
Date

Master's Thesis

EXTENDING AMALIA TO GENERATE ANALYSIS COMPONENTS FOR STATECHARTS

By

Chad R. Meiners

A THESIS

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

Department of Computer Science and Engineering

2004

ABSTRACT

EXTENDING AMALIA TO GENERATE ANALYSIS COMPONENTS FOR STATECHARTS

By

Chad R. Meiners

The Amalia framework generates components that automate the analysis of operational specifications and designs. The original Amalia framework was designed to use Plotkin-style structural operational semantics rules as an input language for generating lightweight components and was suitable for generating analysis components for languages such as LOTOS and linear temporal logic. However, these Plotkin-style rules were not sufficient for more semantically complex formal languages like StateCharts. This paper describes the extensions to Amalia's input language that allow the Amalia framework to generate analysis components for StateCharts. These extensions include the ability to specify multiple semantics relations and the ability the define semantics rules with variable numbers of premises and with negated premises. These extensions are validated by using the semantics for StateCharts to generate an analysis component for ArgoUML.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Automated software-engineering environments (ASEs) create and manipulate representations of specifications, designs, and programs (hereafter called *system descriptions*). The Amalia framework uses generative programming techniques (C.f., [9]) to generate software to analyze the behavior of system descriptions that are written in a language with an operational semantics [13]. Amalia-generated analyzers are not stand-alone tools, but rather are *software components*, which can be customized and tightly integrated into an ASE [31]. Consequently, this software must satisfy design concerns that do not arise in stand-alone analysis/verification tools. Since the publication of [13], I have refined and extended Amalia to handle a larger class of operational specification languages and to generate components that can be tightly integrated into a larger class of environments. This thesis describes my extensions and reports my experience in an integration study.

Amalia was originally designed to allow tool developers to systematically attend to difficulties that arise when integrating analysis and verification capability into an ASE. Specific difficulties include the need to customize analysis algorithms to incorporate problem or domain-specific optimizations and the need to manage translations among multiple (slightly different) representations of a system description under analysis. To support customization, analysis components are assembled from more primitive components using layered composition, in the style of GenVoca component assemblies [2]. Moreover, every assembly includes a component called a *step analyzer*, which is automatically generated from an operational-semantic specification of the language being analyzed. To minimize the need to translate internal representations, step analyzers are implemented as *metaprograms*, which must be *instantiated* to operate over a particular (ASE-specific) internal representation.

Unfortunately, the version of Amalia reported in [13] could not generate analyzers for many popular modeling notations, such as StateCharts, or popular ASEs, such as ArgoUML. This deficiency derives from two limitations of the original tool suite. First, the language used for generating step analyzers was based on Plotkin's structural operational semantics [26], which lacks the power to express the semantics of StateCharts.[1] Second, the Amalia tool suite lacked the powerful metaprogramming facilities required to instantiate these generated step analyzers over the variety of ASE-specific internal representations that occur in practice. Lacking these facilities, our analysis components had to assume representations conformed to certain design conventions, such as supporting traversals via the *visitor pattern* [14]. I addressed these limitations by (1) developing a new step-analyzer generator, which uses a more expressive operational-semantics language, and (2) removing the conventional dependencies on representations by introducing explicit metaprogramming capability into the Amalia tools.

This thesis describes the new and improved step-analyzer generator with emphasis on its input language, which provides the features required to specify the semantics of visual modeling notations, such as StateCharts. Briefly, these extensions provide the ability to reason about multiple semantic relations; to define these relations using semantic rules that can verify or refute assertions about the contents of these relations; and to concisely define the semantics of language features whose term structure involves an unbounded number of sub-expressions. Using these extensions, I was able to generate an analyzer of UML State Diagrams, using the operational StateCharts semantics of [23], and I was able to tightly integrate this analyzer into a third-party UML modeling tool, ArgoUML [29].

The results described in this thesis lend further credence to the hypothesis, first presented in [31], that generative programming techniques enable the tight integration of powerful analysis capability into existing tools and environments. Specifically, these results extend the class of representations that can be analyzed by an Amalia-generated

[1] In fairness, a compositional operational semantics of StateCharts appeared only recently (see [23, 19]).

analyzer. The remainder of this thesis is structured as follows. The thesis first provides a brief introduction to the Amalia framework, operational semantics, and the StateCharts notation, which is the subject of our study (**Chapter 2**). The thesis then describes the functional architecture of the new step analyzer generator and describes how this generator produces components that are amenable to tight integration (**Chapter 3**). The generator processes semantics specifications that are written using a new language, which extends Plotkin-style operational semantics in several ways (**Chapter 4**). I validated this new language and the step analyzer generator itself on a case study that involved adding the analysis of StateCharts to the ArgoUML environment (**Chapter 5**). The thesis concludes with a discussion of related and future work (**Chapter 6**).

Chapter 2

Background

2.1 StateCharts

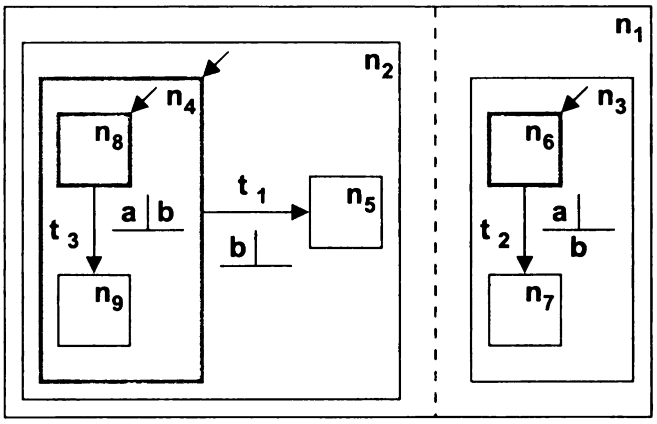


Figure 2.1: Example statechart from [23]

StateCharts is a popular visual language for specifying behaviors of *reactive systems.* I use the term 'system' to mean a computational unit with identity, persistence, and behavior—similar to Milner's notion of an *agent* [24]. A system is reactive if it is *event-driven*—that is, if the computations that it performs are in response to events produced by its *environment.* These computations typically broadcast other events back to the environment, which may respond by offering further events. Thus, the system and its environment engage in an on-going interaction. In this thesis, I consider a simple dialect of StateCharts, treated in [23], which models sequential and parallel composition of reactive systems.

A statechart is a hierarchical state machine, i.e., a machine whose states may themselves be statecharts[1] (**Figure 2.1**). In essence, the notation identifies the notion of a state with that of a statechart. A rectangle denotes a state (statechart); each has a unique name, which we show in the state's upper right corner (e.g., n_1, n_2). The notation distinguishes three types of states: A *basic* state (e.g., n_5, n_6) has no internal structure; an *or-state* (e.g., n_2, n_3) is a sequential composition of two or more sub-states; and an *and-state* (e.g., n_1) is a parallel composition of two or more sub-states.

An or-state comprises *transitions*, as well as sub-states; it also distinguishes one of the sub-states as the *current* sub-state and one as the *initial* sub-state. When an or-state is executing, the current sub-state indicates the locus of control within the or-state; we mark it by a thick shaded outline. Control starts in the initial sub-state, which we mark by a short incoming arrow. In **Figure 2.1**, control resides in n_4, n_6, and n_8, which are the initial states of, respectively, n_2, n_3, and n_4. A statechart, such as the one labeled n_1, in which the initial state of every or-state is the current state is said to be in its *default configuration.*

A transition is drawn as an arrow from a *source* sub-state to a *target* sub-state. Every transition has a unique name and three classes of associated events, *triggers*, *guards*, and *actions*. The transition is *enabled* if control resides in the source sub-state and if the environment offers all of the associated triggers and none of the associated guards.[2] When enabled, a transition may *fire*; in doing so, it broadcasts the associated actions to the environment and transfers control of the or-state to the target sub-state. I show transition names (e.g., t_1, t_2) on one side of the transition and the associated events (e.g., a, b) on the other, with triggers on the top left, guards on the top right, and actions below a perpendicular ($\perp$). Thus, in **Figure 2.1**, both t_1 and t_2 can fire if the environment offers both a and b; whereas t_3 cannot fire in this case because it is guarded by b. Firing t_1 causes control to leave n_4 (also, n_8) and enter n_5, and firing t_2 transfers control from n_6 to n_7 and

[1] We write "StateCharts" when referring to the language, and "statechart" when referring to an expression in StateCharts that denotes a hierarchical state machine.

[2] Most treatments of StateCharts distinguish only triggers and actions, but allow for positive and negative triggers. Since negative "triggers" disable a transition (as opposed to triggering the transition), I prefer to refer to them as "guards."

generates the event *b*.

An and-state comprises sub-states all of which execute in parallel. Hence, when executing an and-state, each sub-state has its own locus of control. A dotted line separates states that are composed in parallel.

StateCharts belongs to the family of *synchronous* specification languages. A number of formal semantics exist [17, 18, 22, 27, 23]. Most define the notion of a *micro step*, in which an enabled transition fires, possibly enabling previously disabled transitions and disabling other previously enabled transitions; they then define a *macro step* as a maximal sequence of micro steps (subject to certain conditions that are discussed in the sequel). Intuitively, the maximality requirement expresses that a complete macro step occurs only when there are no enabled transitions left to fire. Once a macro step occurs, the system is ready to respond to a new set of events offered by its environment. An observer of the system observes only a sequence of macro steps. That is, to an observer, the micro steps that make up a macro step appear to take place concurrently.

The Amalia step analyzer generator operates on a *compositional* semantics for a specification notation. Compositional semantics are specified using *semantic rules* that define the semantics of composite expressions in terms of the semantics of their parts. Here, we are interested in the semantics of expressions that denote states. Thus, we require rules that define the semantics of composite states in terms of the semantics of their sub-states and other parts (transitions, events, and so on). Unfortunately, the macro step relation is not compositional [23]. Lüttgen et al. therefore provide a compositional semantics on the micro step level: they provide a compositional semantics for the micro step relation and for an auxiliary relation, called the *clock relation*. They then describe how to recover the macro step relation from these two compositional relations.

For defining micro steps and clock steps, they introduce two new types of states, *inner-states* and *outer-states*. We refer to a statechart that does not contain any inner-states or outer-states as a *pure* statechart. Intuitively, a statechart that contains inner-states and/or outer-states represents an intermediate stage in the construction of a macro step. Starting with a pure statechart, a sequence of micro steps collects transitions that can be taken in a single macro step, provided the environment offers appropriate events. The micro steps transform sub-states of the statechart into outer-states and inner-states, each of which represents a choice of some transition to be taken as part of the macro step being constructed. An outer-state represents a statechart that is committed to taking one of its own transitions, called an *outer transition*, and an inner-state represents a statechart that is committed to taking a transition, called an *inner transition*, that is in one of its sub-states. A clock step marks the start and end of a macro step. The semantics makes these notions more precise.

StateCharts is a graphical language; however, for defining semantics it is more convenient to use textual expressions to denote statecharts. **Figure 2.2** shows the textual expressions that I use to represent the graphical elements of the example statechart in **Figure 2.1**. To save space, I introduce abbreviations (tr_3, s_8, etc.). In general, I use s_j to stand for a statechart with name n_j in its default configuration and tr_i to stand for the transition with name t_i. Additionally, I use s_j', s_j'', etc. to stand for statecharts with the name n_j that are the result of executing a micro step, and s_j^o, s_j^{oo}, etc. to stand for statecharts with the name n_j that are the result of executing a macro step.

As shown in **Figure 2.2**, the textual expression denoting a transition is a sextuple composed of the transition name, the name of the source state, the set of trigger events, the set of guard events, the set of action events, and the name of the target state. Statechart expressions are denoted by a statechart operator (e.g. Basic) followed by the components of the statechart, which are delimited by parentheses. The components of or-states and inner-states are, in order: the state name, the set of sub-states, the current sub-state, the initial sub-state, and the set of transitions. The components of outer-states are similar except that the current sub-state is replaced with the name of the transition that is fired. Graphically, I distinguish inner and outer-states from others using roundtangles; furthermore, the fired transitions of outer-states are highlighted since outer-states lack current states.

2.2 Structural operational semantics

A detailed treatment of structural operational semantics (SOS) is beyond the scope of this thesis. Instead, I describe only the key ideas of SOS that are needed to understand the

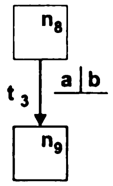
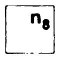
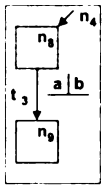
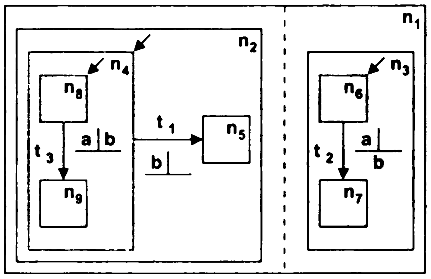
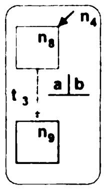
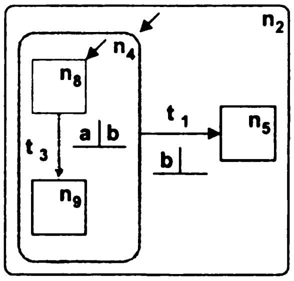
Type	Graphic	Text
trans- ition		$tr_3 \triangleq \langle t_3, n_8, \{a\}, \{b\}, \varnothing, n_9 \rangle$
basic state		$s_8 \triangleq \mathrm{Basic}(n_8)$
or state		$s_4 \triangleq \mathrm{Or}(n_4, \{s_8, s_9\}, s_8, s_8, \{tr_3\})$
and state	 n1 n2 n4 n8 t3 a | b n9 t1 b | n5 n3 n6 t2 a | b n7	$s_1 \triangleq \mathrm{And}(n_1, \{s_2, s_3\})$
outer state		$s'_4 \triangleq \mathrm{Outer}\begin{pmatrix} n_4, \{s_8, s_9\}, \\ tr_3, s_8, \{tr_3\} \end{pmatrix}$
inner state		$s'_2 \triangleq \mathrm{Inner}\begin{pmatrix} n_2, \{s'_4, s_5\}, \\ s'_4, s_4, \{tr_1\} \end{pmatrix}$

Figure 2.2: The graphical and textual representations for the StateCharts syntax

rules for the compositional semantics of StateCharts contained within [23]. These rules illustrate the limitations of the previous Amalia generator and motivate the extensions described in later sections of the paper.

In an operationally-defined specification language, a *system expression* represents a system in some state of execution, called a *configuration*.[3] For example, in the StateCharts language, system expressions are statecharts, which represent configurations via current states. A key concept in modeling the behavior of a system is the notion of a *step*, which is a computational action that causes the system to transition to a new configuration. Whereas the configuration of a system determines the steps in which the system may engage, we formalize behavior as a relation that associates a configuration with the steps that a system may engage in when in that configuration.

More formally, we represent a step as a tuple whose last component, a system expression, represents the system in its new configuration. If a step contains more than one component, those that precede the last component designate the details of the computational action. A *step relation* is then a relation that associates system expressions to steps. For example, the behavior of a statechart is modeled using two such relations, called the micro-step and clock-step relations:[4]

$$\begin{aligned} \text{micro step relation: } \rightarrow \; &\subseteq \; \mathcal{SC} \times (2^{\mathcal{E}} \times 2^{\mathcal{E}} \times 2^{\mathcal{E}} \times \mathcal{SC}) \\ \text{clock step relation: } \twoheadrightarrow \; &\subseteq \; \mathcal{SC} \times \mathcal{SC} \end{aligned}$$

where $\mathcal{SC}$ and $\mathcal{E}$ denote the sets of, respectively, statecharts and events.

As is customary, if s and s' are statecharts and R, P, and A are sets of events, then we write $s \xrightarrow[A]{R,P} s'$ instead of $\langle s, \langle R, P, A, s' \rangle \rangle \in \rightarrow$, and $s \twoheadrightarrow s'$ instead of $\langle s, s' \rangle \in \twoheadrightarrow$; moreover, we refer to these statements as *step assertions*. Intuitively, the former asserts that, if the environment offers all of the events in R and none of the events in P, then s can perform a micro step, transforming itself into s' and producing the events in A. Similarly, the latter asserts that s can execute a clock step, transforming itself into s'.

[3]to prevent confusion with the specific notion of "state" in StateCharts.

[4]Here and in the remainder of this section, the notation is in the spirit of that used in [23], but modified for readability and to correspond more closely to that used for Amalia.

Given a step assertion, we refer to the first expression as the *subject* of the assertion, and the step in which the subject may engage as the *target*. Moreover, we refer to the expression that represents the new configuration of the system as a *derivative* of the subject expression. By convention, if a step is a tuple of multiple components, the derivative expression will always be the last component in the tuple, and the tuple attained by aggregating all components except the derivative is called the *label*. For example, in $s \xrightarrow[A]{R,P} s'$, s is the subject; the step $\langle R, P, A, s' \rangle$ is the target; s' is a derivative of s; and the triple $\langle R, P, A \rangle$ is the label. Similarly, in $s \twoheadrightarrow s'$, s is the subject, $\langle s' \rangle$ is the target, and s' is a derivative of s; this latter step assertion has no label.

A structural operational semantics describes how to compute, for a given step relation, the step assertions that pertain to a given subject expression by analyzing the syntactic structure of that expression. Specifically, these semantics comprise *semantic rules*, which prescribe how to compute step assertions for a composite expression from the step assertions engendered by its sub-expressions. A semantic rule has the form:

$$\mathit{name}\ \frac{\mathit{premise}}{\mathit{conclusion}}\ \mathit{side\text{-}condition}$$

It is to be read as saying that, if all assertions in *premise* and *side-condition* hold, then you can infer that *conclusion* also holds; *name* provides a name by which to refer to the rule. The conclusion and the premise (if any) contain step assertions, and a side-condition is a boolean expression. The subjects of step assertions in a premise must be sub-expressions of the subject of the conclusion. A premise typically introduces one or more variables, which may be referenced in the conclusion and in a side condition. The expressive power of a notation for writing semantic rules is determined by the forms of assertions that may appear in *premise*, *conclusion*, and *side-condition*.

2.2.1 Micro step relation rules

Table 2.1 depicts the set of micro step rules for Statecharts. A micro step describes a transformation that is available for an individual statechart as part of a bigger macro step. In most cases the premises of micro step rules require that a sub-state is able to perform a micro step. Thus, the micro steps available to a statechart depend on the micro steps

$$\text{mOr1}\ \frac{}{\mathrm{Or}(n, S, s_c, s_i, T) \xrightarrow[\mathrm{actions}(t)]{\mathrm{triggers}(t),\mathrm{guards}(t)} \mathrm{Outer}(n, S, t, s_i, T)}$$

$$\text{where } \exists\, t \in T \bullet \mathrm{source}(t) = \mathrm{name}(s_c)$$

$$\text{mOr2}\ \frac{s_c \xrightarrow[A]{R,P} s'_c}{\mathrm{Or}(n, S, s_c, s_i, T) \xrightarrow[A]{R,P} \mathrm{Inner}(n, S', s'_c, s_i, T)}$$

$$\text{where } S' = (S \setminus \{s_c\}) \cup \{s'_c\}$$

$$\text{mInner}\ \frac{s_c \xrightarrow[A]{R,P} s'_c}{\mathrm{Inner}(n, S, s_c, s_i, T) \xrightarrow[A]{R,P} \mathrm{Inner}(n, S', s'_c, s_i, T)}$$

$$\text{where } S' = (S \setminus \{s_c\}) \cup \{s'_c\}$$

$$\text{mAnd}\ \frac{\exists\, s \in S \bullet s \xrightarrow[A]{R,P} s'}{\mathrm{And}(n, S) \xrightarrow[A']{R',P'} \mathrm{And}(n, S')}\ \mathrm{NoConflicts}(S \setminus \{s\}, R, P, A)$$

$$\text{where}\ \begin{array}{l} R' = R \setminus \mathrm{EnvironmentEvents}(S \setminus \{s\}) \\ A' = A \setminus \mathrm{EnvironmentEvents}(S \setminus \{s\}) \\ P' = P \setminus \mathrm{EnvironmentProhibits}(S \setminus \{s\} \\ S' = (S \setminus \{s\}) \cup \{s'\} \end{array}$$

Table 2.1: Micro step rules for StateCharts

available to its sub-states. The rules assume that a number of functions are available. The glossary contains a listing of all functions and their definitions.

mOr1 The first rule, mOr1, allows an or-state to engage in a micro step with a label $\langle \text{triggers}(t), \text{guards}(t), \text{actions}(t) \rangle$, which is the label extracted from the transition t, and a derivative $\text{Outer}(n, S, t, s_i, T)$ for each transition t in T such that the source state of t is equal to s_c, the current state of the or-state. This rule provides the semantics for determining when an or-state may fire an outer transition. Intuitively, it says that, for every transition leaving an or-state's current state, if the environment of the or-state offers all of the transition's triggers and none its guards, then the or-state can produce the transition's actions and transform itself into an outer-state. An outer-state is essentially an or-state, but one for which a commitment has been made to fire a specific transition. An example application of this rule is found in **Table 2.2**. **Figure 2.3** depicts the conclusion inferred from this application. In figures, a labeled dashed arrow denotes the micro step relation. This application justifies the assertion $s_4 \xrightarrow[\varnothing]{\{a\},\{b\}} s_4'$. That is, if the environment offers the event a and does not offer the event b, then s_4 can transition into s_4' and, in doing so, it does not produce any actions. The outer-state s_4' is committed to performing the transition t_3.

$$\text{mOr1}\ \frac{}{s_4 \xrightarrow[\varnothing]{\{a\},\{b\}} s_4'}$$

where

$$tr_3 \triangleq \langle t_3, n_8, \{a\}, \{b\}, \varnothing, n_9 \rangle$$
$$s_4 \triangleq \text{Or}(n_4, \{s_8, s_9\}, s_8, s_8, \{tr_3\})$$
$$s_4' \triangleq \text{Outer}(n_4, \{s_8, s_9\}, tr_3, s_8, \{tr_3\})$$

Table 2.2: Application of mOr1

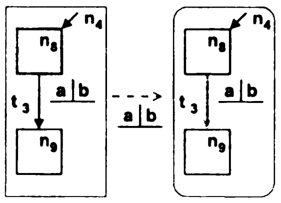


Figure 2.3: Graphical representation of the conclusion in **Table 2.2**

mOr2 and mInner The second rule, mOr2, states that an or-state may engage in a micro step with label $\langle R, P, A\rangle$ and derivative expression Inner(n, $(S \setminus \{s_c\}) \cup \{s_c'\}$, s_c', s_i, T), if s_c may engage in a micro step with label $\langle R, P, A\rangle$ and derivative expression s_c'. Likewise, the third rule, mInner, states that an inner-state may engage in a micro step, if s_c may engage in a micro step. These two rules allow micro steps from sub-states to update the configuration of and percolate through or-states and inner-states. An inner-state is essentially an or-state in which a commitment has been made to fire an inner transition. **Table 2.3** shows an application of mOr1 followed by an application of mOr2. The conclusion of the application of mOr1 serves as the premise for the application of mOr2, indicated by stacking the applications in the fashion shown. A sequence of stacked rule applications is called a *derivation*. The derivation in **Table 2.3** justifies the assertion $s_2 \xrightarrow[\varnothing]{\{a\},\{b\}} s_2'$. In other words, if the environment offers an a-event and does not offer a b-event, then s_2 can transition into s_2' and, in doing so, it does not produce any actions. The inner-state s_2' is essentially an or-state, but one that is committed to firing the inner transition t_3. **Figure 2.4** depicts the conclusion of the derivation in **Table 2.3**.

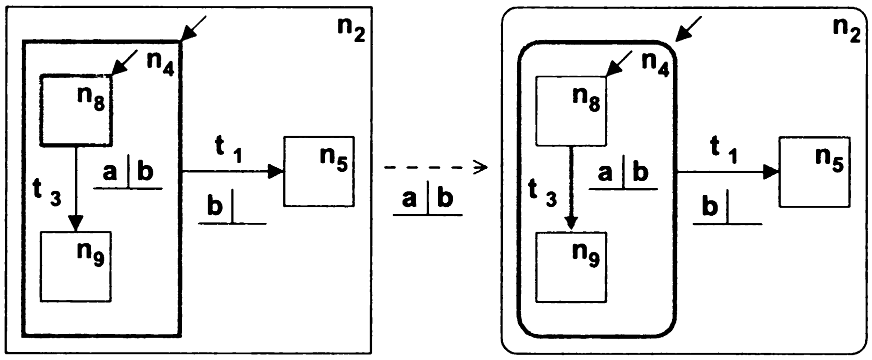


Figure 2.4: Graphical representation of the conclusion in **Table 2.3**

$$\text{mOr2}\ \dfrac{\text{mOr1}\ \dfrac{}{s_4 \xrightarrow[\varnothing]{\{a\},\{b\}} s_4'}}{s_2 \xrightarrow[\varnothing]{\{a\},\{b\}} s_2'}$$

where

$$\begin{aligned}
tr_1 &\triangleq \langle t_1, n_4, \{b\}, \varnothing, \varnothing, n_5 \rangle \\
s_2 &\triangleq \mathrm{Or}(n_2, \{s_4, s_5\}, s_4, s_4, \{tr_1\}) \\
s_2' &\triangleq \mathrm{Inner}(n_2, \{s_4, s_5\}, s_4', s_4, \{tr_1\})
\end{aligned}$$

Table 2.3: Application of mOr2

mAnd The last rule, mAnd, provides the semantics for an and-state. Unlike the previous rules, mAnd has a side condition. A micro step, $\phi\colon s \xrightarrow[A]{R,P} s'$, of sub-state, $s \in S$, of an and-state, $\mathrm{And}(n, S)$, can contribute to a micro step of $\mathrm{And}(n, S)$ only if the rule's side condition is true. Intuitively, $\mathrm{And}(n, S)$ represents an intermediate stage in the construction of a macro step. It is produced from a pure statechart by a series of micro steps, each indicating a transition in one of the sub-states in S that will be fired as part of the macro step. The side condition of mAnd, therefore, checks that the transition within s that s' commits to firing does not conflict with transitions that other sub-states in S have already committed to firing. Provided that no conflict exists, ϕ can contribute to the macro step that is being constructed, transforming $\mathrm{And}(n, S)$ into $\mathrm{And}(n, S')$. The label of the micro step for $\mathrm{And}(n, S)$ is updated to contain only the "new" events that affect the environment. That is required or action events within ϕ that are already present in the environment due to previously committed transitions within $S \setminus \{s\}$ are not present in the micro step's label. Likewise, events that are prohibited by the committed transitions within $S \setminus \{s\}$ are removed from the current micro step's label[5]. Intuitively, if some earlier transition that the macro step will take requires, prohibits, or generates an event, then the environment of $\mathrm{And}(n, S)$ does not need to require, prohibit or generate that event. **Table 2.4** shows

[5]This rule differs from the corresponding rule in [23] because their rule neglects to adjust the label's action and prohibited events. I believe their rule to be in error because two rules for the clock step relation rely on the adjustment of these event sets.

a derivation that contains an application of the mAnd rule. A graphical representation of the conclusion is found in **Figure 2.5**.

$$\text{mAnd}\ \dfrac{\text{mOr2}\ \dfrac{\text{mOr1}\ \dfrac{}{s_4 \xrightarrow[\varnothing]{\{a\},\{b\}} s_4'}}{s_2 \xrightarrow[\varnothing]{\{a\},\{b\}} s_2'}}{s_1 \xrightarrow[\varnothing]{\{a\},\{b\}} s_1'}$$

where

$$s_1 \triangleq \text{And}(n_1, \{s_2, s_3\})$$
$$s_1' \triangleq \text{And}(n_1, \{s_2', s_3\})$$

Table 2.4: Application of mAnd

2.2.2 Clock step relation rules

Table 2.5 lists the clock rules used to define the operational semantics of StateCharts [23]. Intuitively, a clock steps marks the completion of a macro step. Thus, a clock step can be viewed as actually taking all transitions that a statechart has committed to take in constructing a macro step. The only difficulty in defining clock steps comes from needing to ensure that a clock step occurs only when all micro steps that are enabled by events offered by the environment have occurred. This requirement is expressed using negated premises. A negated premise has the form

$$s \overset{R,P}{\underset{A}{\not\longrightarrow}} s'$$

which is an abbreviation for the assertion

$$\nexists s' \bullet s \xrightarrow[A]{R,P} s'$$

cBas The first rule, cBas, asserts that a clock step leaves a basic statechart unchanged. An application of this axiom is found in **Table 2.6**, and its conclusion is depicted graphi-

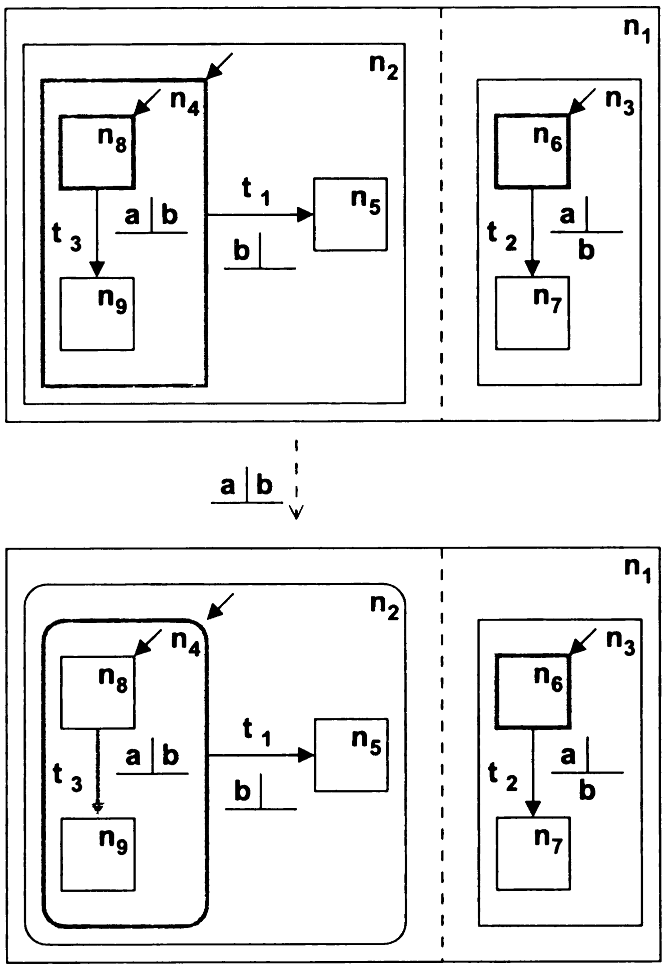


Figure 2.5: Graphical representation of the conclusion in **Table 2.4**

$\text{cBas}\ \dfrac{}{\text{Basic}(n) \twoheadrightarrow \text{Basic}(n)}$
$\text{cOuter}\ \dfrac{}{\text{Outer}(n,S,t,s_i,T) \twoheadrightarrow \text{Or}(n,S',s_t,s_i,T)}$ where $s_t = \text{default}(S,t)$ $S' = (S \setminus \{\text{target}(S,t)\}) \cup \{s_t\}$
$\text{cInner}\ \dfrac{s_c \twoheadrightarrow s'_c}{\text{Inner}(n,S,s_c,s_i,T) \twoheadrightarrow \text{Or}(n,S',s'_c,s_i,T)}$ where $S' = (S \setminus \{s_c\}) \cup \{s'_c\}$
$\text{cOr}\ \dfrac{\text{Or}(n,S,s_c,s_i,T) \overset{\emptyset,\emptyset}{\underset{\emptyset}{\not\longrightarrow}}}{\text{Or}(n,S,s_c,s_i,T) \twoheadrightarrow \text{Or}(n,S,s_c,s_i,T)}$
$\text{cAnd}\ \dfrac{(\forall s_k \in S \bullet s_k \twoheadrightarrow s'_k),\ \ \text{And}(n,S) \overset{\emptyset,\emptyset}{\underset{\emptyset}{\not\longrightarrow}}}{\text{And}(n,S) \twoheadrightarrow \text{And}(n,S')}$ where $S' = \bigcup s'_k$

Table 2.5: Clock step rules for StateCharts

cally in **Figure 2.6**, where the unlabeled dashed arrow represents the clock step relation.

$$\text{cBas}\ \frac{}{s_8 \twoheadrightarrow s_8}$$

where

$$s_8 \triangleq \text{Basic}(n_8)$$

Table 2.6: Application of cBas

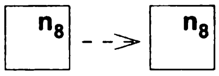


Figure 2.6: Graphical representation of the conclusion in **Table 2.6**

cOuter The second rule, cOuter, asserts that a clock step transforms an outer-state into the or-state produced by taking the designated outer transition. The function, default(S, t), returns the target state of t in its default configuration. This state becomes the current state of the or-state. Thus, this rule represents the firing of the transition designated by the outer-state. An application of cOuter is found in **Table 2.7**, and its conclusion is shown in **Figure 2.7**.

$$\text{cOuter}\ \frac{}{s_4' \twoheadrightarrow s_4^o}$$

where

$$s_4^o \triangleq \text{Or}(n_4, \{s_8, s_9\}, s_9, s_8, \{tr_3\})$$

Table 2.7: Application of cOuter

cInner The third rule, cInner, states that every clock step that can be performed by the current state of an inner-state produces a clock step of the inner-state. Like mInner, cInner

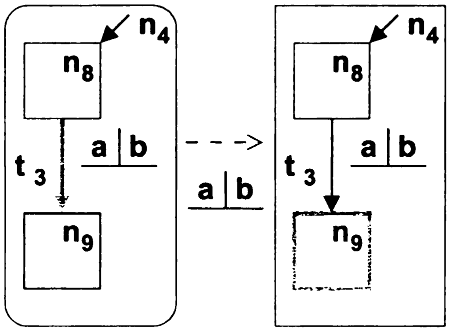


Figure 2.7: Graphical representation of the conclusion in **Table 2.7**

allows clock steps engender by the current state to percolate up through the inner-state. An example of how to apply this rule is found in **Table 2.8**, and its conclusion is shown in **Figure 2.8**.

$$\text{cInner}\ \dfrac{\text{cOuter}\ \dfrac{}{s_4^{\prime} \twoheadrightarrow s_4^{o}}}{s_2^{\prime} \twoheadrightarrow s_2^{o}}$$

where

$$s_2^{\prime} \triangleq \text{Inner}(n_2, \{s_4, s_5\}, s_4^{\prime}, s_4, \{tr_1\})$$
$$s_2^{o} \triangleq \text{Inner}(n_2, \{s_4, s_5\}, s_4^{o}, s_4, \{tr_1\})$$

Table 2.8: Application of cInner

cOr The fourth rule, cOr, contains a negated premise. It says that an or-state may engage in a stuttering clock step if it cannot engage in any micro steps with the label $\langle\varnothing,\varnothing,\varnothing\rangle$. A micro step whose label consists solely of empty sets is called a *free step*. This rule forces an or-state to commit to firing a free transition rather than take a clock step. An application of cOr is found in **Table 2.9**, and its conclusion is shown in **Figure 2.9**.

cAnd The fifth rule, cAnd, says that an and-state can perform a clock step when every sub-state can, provided that the and-state cannot perform a free step. This second requirement prevents a clock step from occurring until all the sub-states of the and-state have committed to firing any transitions that are enabled by the events offered by the environ-

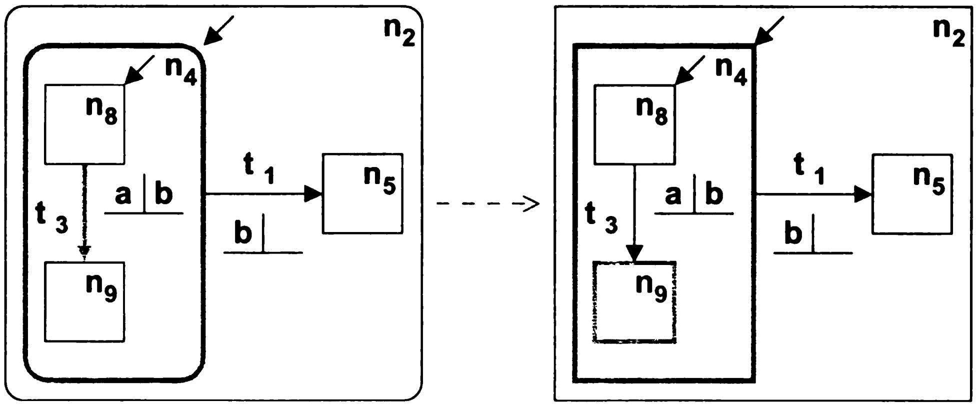


Figure 2.8: Graphical representation of the conclusion in **Table 2.8**

$$\text{cOr}\ \frac{}{s_3 \twoheadrightarrow s_3}$$

where

$$tr_2 \triangleq \langle t_2, n_6, \{a\}, \varnothing, \{b\}, n_7 \rangle$$
$$s_3 \triangleq \text{Or}(n_3, \{s_6, s_7\}, s_6, s_6, \{tr_2\})$$

Table 2.9: Application of cOr

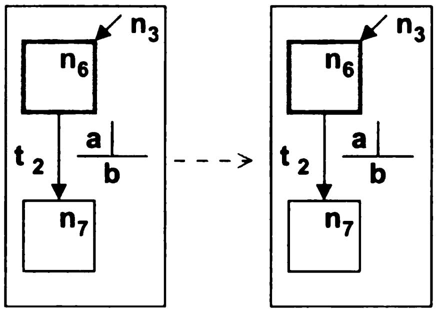


Figure 2.9: Graphical representation of the conclusion in **Table 2.9**

ment or generated by previous micro steps. This rule combined with cOr ensures that enabled transitions will be fired in parallel from a single macro step[6]. An application of this rule is found in **Table 2.10**, and its conclusion is shown in **Figure 2.10**.

$$\text{cAnd}\ \dfrac{\text{cInner}\ \dfrac{\text{cOuter}\ \dfrac{}{s'_4 \twoheadrightarrow s^o_4}}{s'_2 \twoheadrightarrow s^o_2}\ ,\ \text{cOr}\ \dfrac{}{s_3 \twoheadrightarrow s_3}}{s'_1 \twoheadrightarrow s^o_1}$$

where

$$\begin{aligned} s'_1 &\triangleq \text{And}(n_1, \{s'_2, s_3\}) \\ s_3 &\triangleq \text{Or}(n_3, \{s_6, s_7\}, s_6, s_6, \{tr_2\}) \\ s^o_1 &\triangleq \text{And}(n_1, \{s^o_2, s_3\}) \end{aligned}$$

Table 2.10: Application of cAnd

2.2.3 Labeled transition systems

The step relation for an operationally defined specification notation is used in deriving a *labeled transition system* (LTS) from an expression. This LTS models all possible behaviors of the expression and is the basis for various kinds of analyses (e.g., model checking, simulation). For instance, **Figure 2.11** contains a partial LTS for the statechart expression, s_1. Deriving an LTS from a language expression is called *step computation*.

[6]For a more detailed explanation, see [23].

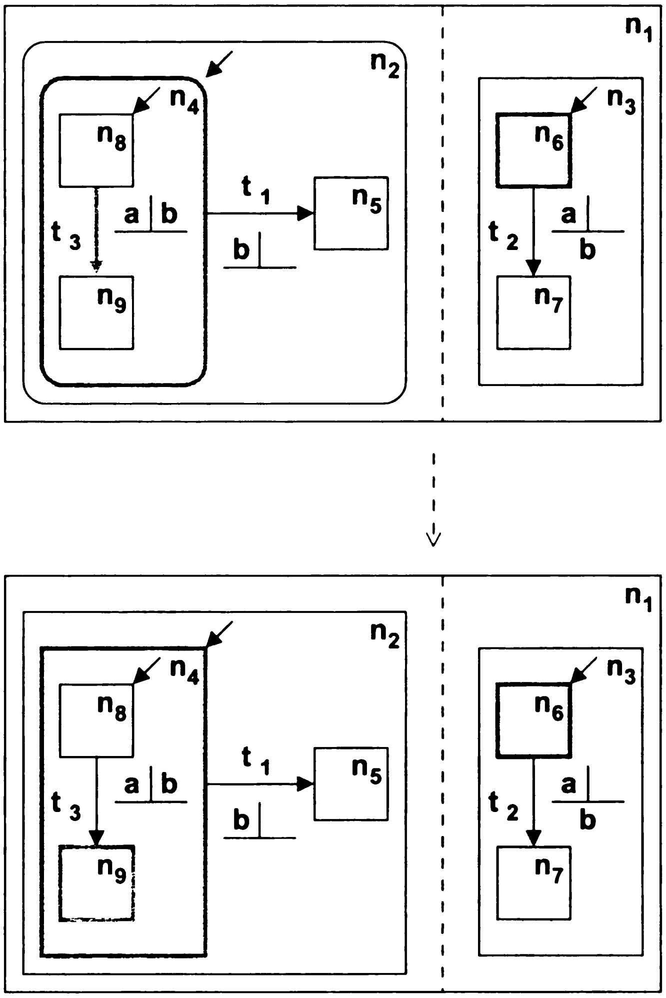


Figure 2.10: Graphical representation of the conclusion in **Table 2.10**

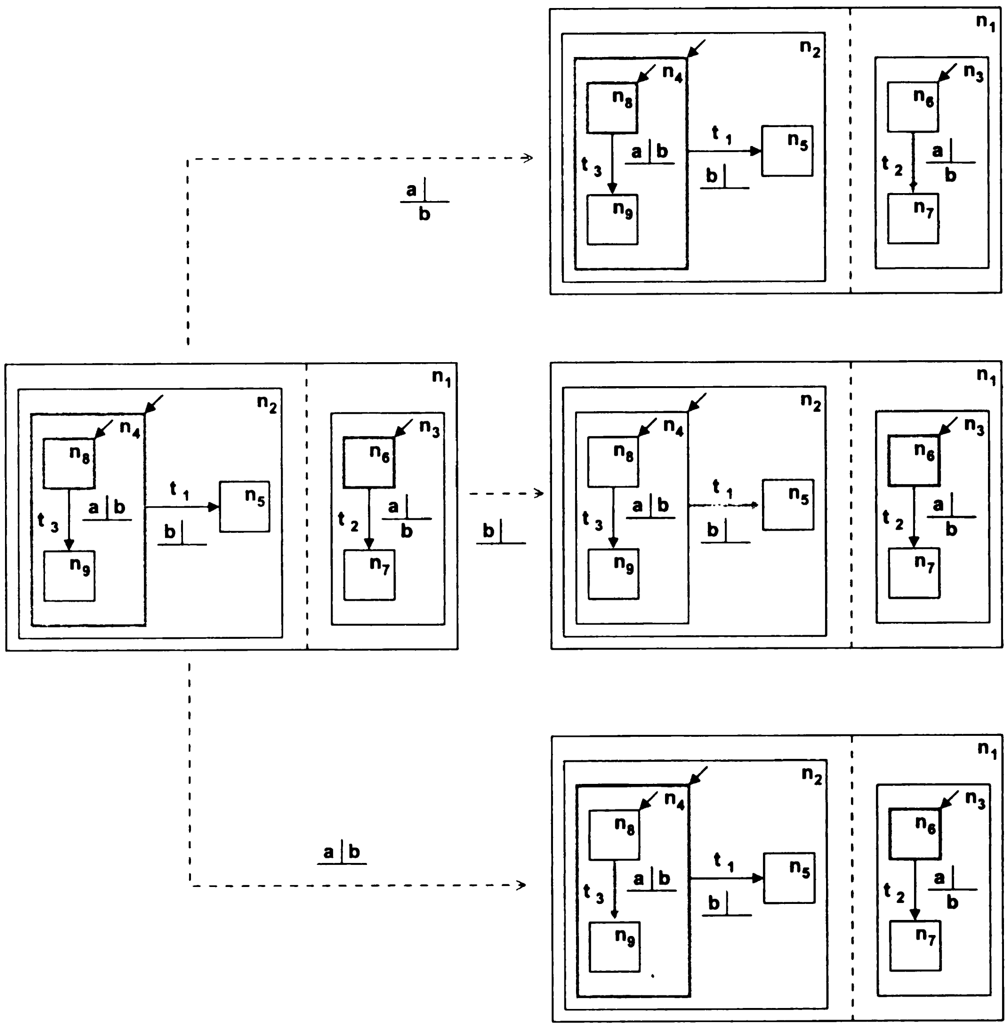


Figure 2.11: Partial LTS for s_1

Level	Realms and contents

3	LTS = { lazyLTS[SA], eagerLTS[LTS], minLTS[LTS] }
	SIM = { sim[SA] }

2	SA = { scSA[SC], lotosSA[LOTOS], ltlSA[LTL] }

1	SC = { scInterp[UMLSD] }
	LTL = { ltlTerm, ltlDNF }
	LOTOS = { lotosTerm }

0	UMLSD = { nsUML }

Figure 2.12: Layered design of Amalia analyzers

2.3 Amalia analyzers

Step computation is the basis for a large variety of analysis and verification tools, from interactive behavioral simulators to exhaustive model checkers. A *step analyzer* is a software component that computes steps from a language expression. Since these expressions do not have standard data structure representations, Amalia uses generative programming techniques, specifically hierarchical components in the GenVoca style [2], to standardize the interface of a step analyzer over a large class of target languages and ASE-specific representations. In [31], the authors show how such a standardized interface enables a range of powerful analysis capability to be customized to operate over a given (ASE-specific) internal representation, thereby enabling tight integration.

The key idea is to decompose behavioral analysis and simulation capability into a layered hierarchy of components, one of which is a step analyzer that has been customized for integration into a given environment. Because each step analyzer implements a standard interface, a component, such as a simulator, can be developed to use this standard interface, thereby making the component portable to a new environment by simply plugging

in an appropriate step analyzer. Components are grouped into plug-compatible libraries called *realms*, and a component may be *parameterized* by one or more (lower-level) realms. A parameterized component is *instantiated* by plugging it into (i.e., stacking it on top of) components of from the required realms.

Figure 2.12 depicts some of the realms and components in the Amalia library. The library comprises four different levels, which indicate the logical position of a component in the context of a layered hierarchy. A particular analyzer in this model is an assembly of components from the various levels and is specified using a GenVoca *type equation*. For example, the type equation:

```
ltsLtl = lazyLTS[ltlSA[ltlTerm]]
```

declares a component `ltsLtl`, which implements the services of a labeled transition system whose states and transitions are computed on demand by analyzing a formula in linear temporal logic (ltl). Specifically: `ltlTerm` is a component that encapsulates the construction and manipulation of ltl expressions; `ltlSA` is a step analyzer for ltl expressions; and `lazyLTS` is an LTS component that derives its states and transitions by invoking the services of a step analyzer, which in this assembly happens to be `ltlSA[ltlTerm]`.

As another example, the type equation:

```
argosim = sim[scSA[scInterp[nsUML]]]
```

declares a component `argosim`, which implements an interactive behavioral simulator of a UML state diagram (in the ArgoUML environment) under the StateCharts semantics of [23]. Specifically: `sim` is a generic simulator, which can be instantiated with an arbitrary step analyzer; `scSA` is a step analyzer for StateCharts using the semantics of [23]; `scInterp` is a StateCharts representation component that implements its services by invoking the services of a UML state-diagram component; and `nsUML` is the ArgoUML state-diagram representation.

To appreciate what is actually happening in the assemblies specified by a type equation, it is instructive to examine what kind of code is "in" these components. For example, the `nsUML` component is a collection of classes, specifically the actual Java classes used to represent state diagrams in ArgoUML. Likewise, the `scInterp` component is a collection of classes that represent the structural features of StateCharts. Moreover, these `scInterp` classes are *facades*[7] whose instances encapsulate instances of the analogous classes in `nsUML`. Thus, instances of `scInterp` classes *interpret* requests for traversing the structure of a statechart by invoking analogous services on instances of the corresponding `nsUML` classes. This interpretation is required because the UML state-diagram syntax differs from that of statecharts; for example, UML does not distinguish inner and outer states.

Finally, I should note that changing Amalia's toolset to target Java instead of C++ removed some of the automation benefits reported in [31]. Most notably, type equations cannot be "compiled", as was possible in the prototype implementation. Briefly, the original Amalia prototype implemented parameterized components in C++ using a technology, called *mixin layers* [30], that implemented component instantiation via C++ template instantiation. Because Java does not provide templates, assemblies are constructed at runtime by the code that invokes the Amalia assembly.

[7]in the sense of the facade pattern [14].

Chapter 3

Step-Analyzer Generator

The step-analyzer component is the key to portability, reuse, and tight integration in an Amalia-generated analyzer. A *step-analyzer generator*, produces step analyzers automatically from a specification of the operational semantics of an abstract representation for *subject language* expressions. The subject language is the formal language whose expressions we wish to analyze.

The generator described in [31],[1] generates step analyzers from a semantics specification written using a limited SOS. The key contribution of this thesis is an improved generator, which uses a more expressive dialect of SOS and which generates code that is more flexible and thus able to integrate into a larger class of ASEs.

Figure 3.1 depicts the functional architecture of our improved step-analyzer generator. In this figure, tool inputs are depicted using crimped-page icons and labeled by text in the typewriter font; tools (and functional components within a tool) are depicted using roundtangles; and the single tool output is depicted using a component icon. The step-analyzer generator takes two inputs, a specification for a metalanguage, which provides a target-ASE neutral representation of subject expressions, and a target-ASE specific mapping of metalanguage constructs. The step-analyzer generator then produces a step-analyzer component, which can be tightly integrated into a specific target ASE.

The metalanguage specification comprises two parts: 1) declarations of the meta data types that are used to represent source expressions and declarations of the functions and

[1]which was called the LWA (or lightweight analyzer) generator

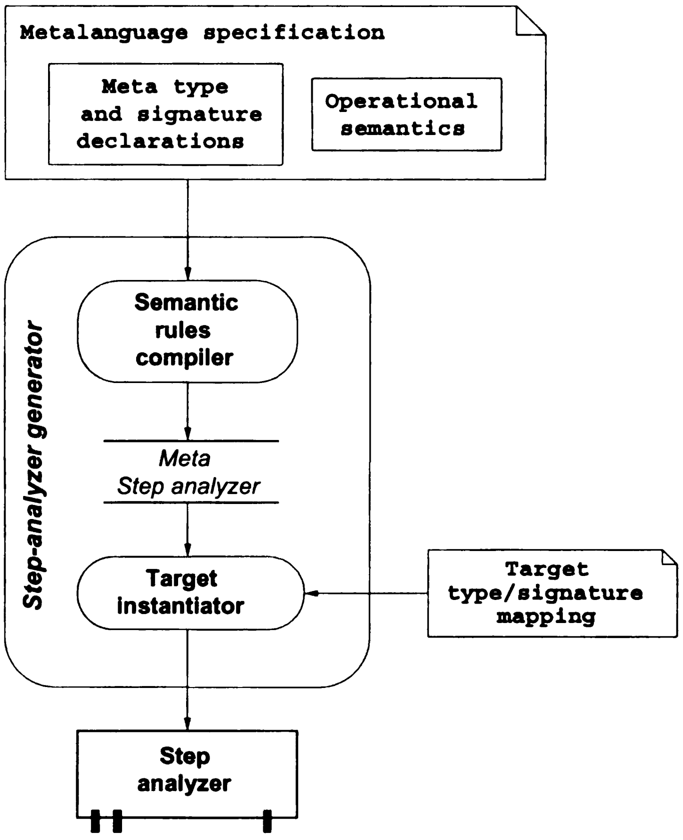


Figure 3.1: Step-analyzer generator: Functional architecture

predicates that are used to manipulate and query these representations. 2) a specification of the formal semantics of subject expressions. The formal semantics are specified in our extended SOS notation and may refer to these types and functions. The language for declaring types and signatures (**Chapter 4**) minimizes assumptions about the target environment within which the generated step analyzer will be integrated. Thus, the box in **Figure 3.1** that refers to these declarations is labeled "Meta type and signature declarations". The target type/signature mapping describes how to instantiate the meta types and functions, which are declared in the metalanguage specification, to the concrete types and operations of a given target ASE. Separating the meta type and signature declarations from the target type/signature mapping allows the language specification to be retargeted for use with different tools.

The step-analyzer generator comprises two smaller generators, the *semantic rules compiler* and the *target instantiator*. The semantic rules compiler inputs a metalanguage specification and produces an *meta step analyzer*, which refers to the meta types and functions defined in the metalanguage specification, as opposed to the concrete types and functions provided by a target environment. The meta step analyzer exists only internally in the generator; therefore, it is depicted in the diagram as a private data store. The target instantiator uses information provided by the target type/signature mapping to transform the meta step analyzer into a component that uses the concrete types and functions provided by the target environment. The resulting component becomes part of the SA realm in **Figure 2.12**, which means that it can be used to instantiate higher-level analysis and verification tools.

By way of comparison with the tool suite described in [31], the semantic rules compiler in **Figure 3.1** corresponds to the old LWA generator, which produced step analyzer components directly, without an explicit target instantiation. In addition, the semantic rules compiler handles more expressive rules than the LWA generator. Finally, the step analyzer component depicted in **Figure 3.1** does not require target representations to implement the visitor pattern, as was required in previous incarnations of Amalia.

Although I built a complete step-analyzer generator, the key contribution of this thesis is the metalanguage specification. Therefore, the details of target integration and the target type/signature mapping are beyond the scope of this thesis.

Chapter 4

Amalia Input Language

Metalanguage specifications and target type/signature mappings are written in the *Amalia Input Language*, whose high-level syntax appears in **Figure 4.1**[1]. The metalanguage specification is divided into four sections: the *type section*, the *signature section*, the *relation section*, and the *rule section*. This chapter describes these four specification languages in detail.

4.1 Type section

The type section defines the meta syntax of expressions in a specification language and also of the steps used in defining the semantics of these expressions. The language provides four categories of types: *atomic types*, *union types*, *set types*, and *tuple types*. Atomic types describe expressions that cannot be further decomposed. An example of an atomic type is the name of a statechart, which we declare:

```
type State_Name is atomic;
```

Union types describe disjoint unions of types and allow us to group related types. For instance, we can declare a type, `State`, to represent all of the different kinds of statechart expressions:

[1]The entire BNF for the Amalia Input Language is found in **Appendix A**

```
language name is
  type_section is
    typedef*
  end type_section;
  signature_section is
    signaturespec*
  end signature_section;
  relation_section is
    relationspec*
  end relation_section;
  rule_section is
    {rule | mrule}*
  end rule_section;
end name ;
```

Figure 4.1: Syntax of the Amalia Input Language system

```
type State is union Basic, And, Or,
                    Inner, Outer;
```

where types `Basic`, `And`, `Or`, `Inner`, and `Outer` have been previously declared[2]. Set types describe sets whose elements must be of a given type. For instance the type for sets of statechart expressions is

```
type State_Set is set State;
```

Finally, tuple types describe aggregate structures with named members. `Or` expressions are a good example of a tuple type.

```
type Or is tuple
  ( name :  State_Name,
    children :  State_Set,
    trans :  Transition_Set,
    n_init :  State_Name,
    n_curr :  State_Name );
```

Each `Or` expression comprises five parts: its name (`name`), its set of sub-states (`children`), its set of transitions (`trans`), its initial sub-state (`n_init`), and its current sub-state (`n_curr`). Notice that two of these parts, `children` and `trans`, are sets, which contain a variable number of elements. The ability of parts to have a variable number of elements motivates two of the SOS extensions that we describe in **Chapter 4.4**.

Tuple types are also used to represent steps in the analysis of a language. Statecharts requires two such types:

[2]The type section allows forward declarations.

```
type MicroStep is tuple
  ( trigger :  Event_Set,
    guard :  Event_Set,
    action :  Event_Set,
    s :  State );

type ClockStep is tuple (s :  State);
```

Notice that the last component in both of these tuples is a `State`, which is used to represent derivative expressions. From this we can see that a `MicroStep` has three labels, each denoting a set of events. Moreover, a `ClockStep` is has no label.

4.2 Signature section

The signature specification declares *function* and *predicate* signatures, which can be used in the rule specification section. Functions differ from predicates in that the former may return values of an arbitrary type, whereas the latter has an implicit boolean return type. The implementations of both functions and predicates are required to be free of side effects.

Signatures may reference any type defined in the type specification. For example, the function signature:

```
function to_Outer (o :  Or,
  t :Transition)
  return Microstep;
```

takes two parameters, the first of type `Or` and the second of type `Transition`, and returns a `MicroStep`. This declaration defines the signature for a function that will return a micro step whose `trigger`, `guard`, and `action` members are obtained from event sets aggregated by the second parameter (`t`); and whose `o` member is an outer-state constructed using the `name`, `children`, `trans`, and `n_init` members of the first parameter (the o expression) and the second parameter. (In Lüttgen et al.'s extended StateCharts notation, an outer-state aggregates a transition instead of a current state.) The actual target implementation will be specified by the target mapping.

Predicate declarations are similar to function declarations, but they do not have a return type. For example:

```
predicate free_step (s :  Microstep);
```

declares a predicate with one parameter of type `MicroStep`. For the validation case study, this predicate is defined to be true when the parameter's `trigger`, `guard`, and `action` members are all empty.

4.3 Relation section

The relation section declares one or more step relations, each of which relates system expressions to steps. Step relations are declared as follows:

`step` *relname* *P* `to` *Q* `;`

where *relname* is the name of the relation and *P* and *Q* are the names of types declared in the type specification. For example, the StateCharts language specification, specifies two step relations, `Micro` and `Clock`, as follows:

```
step Micro is State to MicroStep;
step Clock is State to ClockStep;
```

In the sequel, we refer to the type of elements in the domain of a step relation as its *subject type* and the type of elements in its co-domain as its *target type*. Thus, `State` is the subject type for both `Micro` and `Clock`; whereas `MicroStep` is the target type for `Micro`, and `ClockStep` is the target type for `Clock`. As illustrated by these examples, the target type must be a tuple, and the types of the subject and of the last member of the target type must be the same.

With the type, signature, and relation sections, we have defined the metalanguage syntax for which the rule section provides the semantics. As described in **Chapter 3**, the metalanguage syntax frees the rule section from any ASE-specific representations.

```
rule name ( var :  type )
   [where pred⁺]
 is
   ...
   [with {premise [where pred⁺]}⁺]
   ...
   [without {premise [where pred⁺]}⁺]
   ...
   conclude concl
end name ;
```

Figure 4.2: Elided syntax of extended SOS rules

4.4 Rule specification

This thesis's key contribution is an extended SOS rule notation, which generalizes traditional SOS with new features that increase its expressive power.

4.4.1 Rules

Extended SOS rules are written as depicted in **Figure 4.2**. Each rule is required to have a name (*name*), a parameter (*var*), and a conclusion (*concl*), which is a step assertion of the form:

relname var `to` $f(\overline{X})$

Here, *relname* is the name of a step relation; *var* is the variable declared as the rule's parameter; *f* is a function whose return type conforms to the target type of *relname*; and $\overline{X}$ is a sequence of variables and function applications. In addition, the type of *var* must conform to the subject type of *relname*. A simple example of a rule derived from the SOS rule, cBas, is:

```
rule cBas ( b :  Basic ) is
  conclude Clock b to ClockStep'(b);
end cBas;
```

This rule is named `cBas`; its parameter is `b`; and its conclusion is a clock step assertion. The subject of this conclusion is the rule parameter, and the target is the application of the `ClockStep` constructor to the variable `b`.[3]

A rule is applied by instantiating its subject parameter and its free variables, which may be introduced in the optional parts of the rule, with expressions and steps so as to satisfy all conditions that are specified in the rule. A condition implicitly specified by a rule is that the parameter, when instantiated, must conform to the subject type of the step relation named in the conclusion. Other conditions may be specified using *where clauses*. A where clause names one or more predicates that must be true. Predicates in a where clause appearing in the preamble of a rule are called *pre-conditions* of the rule; whereas predicates in a where clause appearing the body of a rule are called side-conditions.

When a rule can be instantiated in this manner, it is said to justify the instantiated conclusion. The rule `cBas` does not contain a pre-condition or any side-conditions. Therefore, it can be instantiated with any expression of type `Basic`. Once instantiated, `cBas` justifies the assertion that the `Basic` expression can engender a stuttering clock step.

In addition to a conclusion, a rule may contain zero or more premises, each of which is specified using a step assertion of the form:

relname v `to` *s*

where *relname* names a step relation; *v* is a bound variable of the subject type of *relname*; and *s* is a free variable. The introduction of *s* in the premise assertion declares it to be of the target type of *relname*. The keyword `with` opens a block of premises.

An example of an SOS rule with a premise is `mInner`, which may be written:

[3]Notationally, we denote a tuple constructor by appending the name of the type with a tick symbol.

```
rule mInner ( i :  Inner ) is
  let
    s = referent(i.children, i.n_curr);
  with
    Micro s to s1;
  conclude Micro
    i to new_Inner(i,s,s1);
end mInner;
```

This rule contains a single premise, which declares `s1` to be a `MicroStep`. The subject of the premise refers to a variable `s`, which is introduced using a `let` binding[4]. The target of the premise's step assertion introduces a new variable `s1`. The scopes of all variable declarations extend to the end of the rule.

The features of the Amalia Input Language described thus far provide the expressive power of traditional SOS rules. In addition, they add the ability for the subject of premises to be defined by arbitrary computable functions. However, the most significant increase in expressive power is obtained from the following new features, which I now discuss in turn.

4.4.2 Negated premises

In addition to premises, an extended SOS rule may declare negated premises. Syntactically, negated premises are identical to premises with the exception that they are listed in a block that is opened by the keyword `without`. For example, the rule `cOr` uses a negated premise:

[4]To simplify **Figure 4.2**, I have elided `let` bindings, which introduce new variables and bind them to values of functions or variable names.

```
rule cOr ( o :  Or ) is
 without
   Micro o to f1;
   where
     free_step(f1);
 conclude Clock
     o to ClockStep'(o);
end cOr;
```

This rule states that an Or expression may take a stuttering macro step when it cannot engage in a so-called *free* micro step. A micro step is free if it contains no events. We check free-ness using a side condition, which references the predicate `free_step`.

Allowing negated premises in general is non-trivial because they can lead to rules that cannot be used to derive an LTS[16]. Therefore, Amalia only decides that a negated premise is justified when it is impossible to justify the premise that is being negated. Since Amalia rules are compositional, and each expression is finite, all possible steps for the subject of the negated premise may be enumerated to verify that none the subject's steps contradict the negated premise.

4.4.3 Enumerated premises

Some languages contain features that comprise a variable number of parts. In StateCharts, for example, the `And` state comprises a set of sub-states, called its children. If the rule for such a feature requires a premise (or negated premise) for each of its parts, then such a rule cannot be expressed using the facilities described thus far. Enumerated premises solve this problem.

An *enumerated premise* is a premise (or negated premise) that contains a step assertion of the form:

relname E `to` *S*

where *relname* names a step relation; E is a bound variable of type "set of *relname*-subject type"; and S is a free variable. The use of S in this context declares its type to be "set of *relname*-target type." Operationally, a premise assertion of this form requires that the set used to instantiate S has the same size as E. Additionally, the premise requires that the ith element of E can engage in the *relname* step that is the ith step of S.[5] The rule `cAnd` uses this feature:

```
rule cAnd (a :  And) is
  let
    S = a.children;
  with
    Clock S to S1;
  without
    Micro a to f1;
    where
      free_step(f1);
  conclude
    Clock a to
      ClockStep'(
        And'(a.name,all_states(S1)));
  end cAnd;
```

In this rule, the type of the variable `S` is `State_Set`, i.e., set of `State`. Because `S` is the subject of a premise step assertion, that premise is interpreted as an enumerated premise. Consequently, the type of the new variable `S1` is inferred to be of `ClockStep_Set`. To justify the conclusion, this rule must: (1) instantiate `S1` with a set of `Clock` steps of the same size as `a.children` such that each sub-state of a (element of `a.children`) can engage in the corresponding `Clock` step of `S1`, and (2) verify

[5]The target mapping defines a method to iterate over the implementation of the set E and defines a method to construct S.

that a cannot engage in a free Micro step. The target of the conclusion creates a new ClockStep whose derivative is an And state with the same name as that of a, and with a set of sub-states that are computed by extracting the derivative state of each step in S1.

4.4.4 Metarules and pre-conditions

Enumerated premises enable the specification of rules that must verify a set of proof obligations prior to justifying a conclusion. Unfortunately, enumerated premises by themselves cannot always be used to concisely define the semantics of language features with a variable number of parts. Consider, for example, the micro step semantics of an Or expression. An Or expression can engage in a micro step for each transition out of its current state. Depending on the topology of child states and transitions and depending on the current state, these semantics could require the generation of a different step for each element of an Or state's transition set. There is no way to specify a rule with this interpretation using enumerated premises or Plotkin style SOS rules. What is needed is a facility for concisely specifying a variable number of rules, one for each element of a set.

Two language features are used to concisely specify rules of this nature. First, a rule may declare a pre-condition, which constrains the expressions that can be used to instantiate the rule. Using pre-conditions, the notion of a *metarule* is defined as a template for generating an arbitrary number of rules, each of which contains a pre-condition that excludes candidate expressions that lack a particular configuration of parts. To make these ideas concrete, I describe them by elaborating the example of the micro step semantics of Or expressions.

A pre-condition is specified by a where clause that follows the declaration of a rule parameter (**Figure 4.2**). Consider, for example, the following rule:

```
meta ( var :  type )
  {for each var in setexpr;
    [such that pred⁺]}⁺
rule name
  ...
  [where pred⁺]
is
  ...
  [with {premise [where pred⁺]}⁺]
  ...
  [without {premise [where pred⁺]}⁺]
  ...
  conclude conclusion
end name ;
```

Figure 4.3: Elided syntax of metarules

```
rule mOr1-3 ( o :  Or )
 where
   low_bound(o.trans, 3);
   choose(o.trans, 3).source = o.n_curr;
is
 let
   t = choose(o.trans, 3);
 conclude Micro
   o to to_Outer(o,t);
end mOr1-3;
```

According to this rule, a candidate subject must be an `Or` expression that contains at least three transitions, as indicated by the `low_bound` predicate; moreover, the subject's current state (o.n_curr) must also be the source state of the "third" transition. Logically, the pre-condition could have been expressed as a side condition. The Amalia rule language specifies them separately for operational reasons; side conditions are evaluated after having checked the premises.

Clearly, one could develop a library of rules with suitable pre-conditions (e.g., `mOr1-1`, `mOr1-2`, ... `mOr1-k`) in order to analyze specifications whose `Or` states contain up to *k* transitions. However, doing so would be tedious, and would limit the size of the StateCharts that could be analyzed. Fortunately, these libraries of similar rules can be generated automatically from a construct called a metarule, which is now described.

Metarules are written according to the syntax in **Figure 4.3**. Each metarule declares a parameter (*var*), a *selector predicate*, which introduces *metavariables* and is used to govern the generation of rules, and a *body*. The selector predicate is specified using a `for each` clause, which introduces metavariables and specifies the sets over which the variables will range. The clause may also contain an optional `such that` clause, which contains a predicate that may reference the parameter and the metavariables. The body is a rule specification whose parameter is the same as the metarule parameter and within which any metavariables introduced in the selector predicate may appear. When a metarule is applied to an expression, a new rule is generated for each valid metavariable assignment.

`mOr1` is an example of a metarule[6].

```
meta (o : Or)
 for each t in o.trans;
   such that t.source = o.n_curr;
rule mOr1 is
 conclude Micro
   o to to_Outer(o,t);
end mOr1;
```

In `mOr1`, there is one meta-variable, `t`, which ranges over the set of `o`'s transitions. The predicate `t.source = o.n_curr` further constrains `t` to be a transition that emanates from the current state of `o`.

Additionally, metarules may also have premises and guards.

```
meta (a : And)
 for each s in a.children;
rule mAnd is
 with
   Micro s to s1;
   where
     no_conflicting_transitions(a,s,s1);
 conclude
   Micro a to microstep_and(a,s,s1);
end mAnd;
```

Here, `mAnd` is a metarule with a premise.

[6]More specifically, `mOr1` is a metaaxiom.

4.4.5 Target Type/Signature Mapping

A brief overview of an example target type/signature mapping will demonstrate the important role that the type and signature sections play as key enablers for the instantiation of meta step analyzers for a variety of ASE-specific representations. The type and signature section provide a contract of expected operations and obligations that a target implementation is expected to fulfill. For example, all type declarations contain the expectation that the target implementation will have a corresponding type or class.

Type and Signature Operations and Obligations

	Type / Class Name	Constructor	Iteration	Field Accessors
atomic	√			
union	√			
set	√	√	√	
tuple	√	√		√

Table 4.1: Summary of type operations and obligations.

Table 4.1 contains a summary of the requirements for each type in the meta syntax. All type declarations require a corresponding type or class name. Both set and tuple types require an operation that constructs a new instance of their type because enumerated premises construct set instances, and tuple constructors construct tuple types. Set types have an additional operation that iterates over the elements of the set. Likewise, tuple types have operations that provide access to the tuple's components. Since signature declarations define a user specified operation, the target must provide an implementation for that operation.

Mapping Obligations to Implementations

Currently, the Amalia step analyzer generator uses a *parameterized substitution text* technique to map type and signature operations into implementation code. Parameterize substitution text is source code text that is used by the target instantiator to fulfill the operations required by the meta syntax. This source code text may contain specially delimited parameters that indicate how source code from subexpressions should be substituted into the source code text. Since this method is not the the focus of thesis, I will not provide a full description of the process, but I will give a short description of how expressions in rules are mapped into implementation code.

```
for tuple And'(n,s)
  use org.argouml.ui.amalia.ExtendedStateCharts.uAnd ;
  with ##Factory##.And(#n#,#s#) ;
  n : #And#.Name() ;
  s : #And#.States() ;
end And;


for signature Replace(In_Set,Old,New)
  use #In_Set#.Replace(#Old#, #New#) ;
end Replace;
```

Figure 4.4: Portions of the target type/signature mapping description for ArgoUML.

Figure 4.4 contains a small excerpt of the target mapping for StateCharts to ArgoUML. This excerpt contains two mapping declarations: the first is for the `And` type, and the second is for the `Replace` function. The declaration for the `And` tuple type begins with, `And'(n,s)`, which positionally renames the components of the `And` type to `n` and `s`[7]. These new names allow the components to be parameters within the substitution text of the mapping declaration. The use clause provides the implementation

[7] `n` corresponds to name and `s` corresponds to children.

type name for the type being mapped[8]. The with clause defines how to construct an instance of an And type given instances of n and s. Tuple types also have a clause for each component, which defines how to acquire an instance of the type's component. n : #And#.Name() ; and s : #And#.States() ; are the clauses that contain the parameterized substitution text for accessing the n and s components respectively.

Each parameter in the substitution text is delimited by either pound symbols(#) or double pound symbols(##). Single pound parameters denote parameters that apply directly to the type instance being manipulated. For example, the parameter #And# in n : #And#.Name() ; denotes the instance of the And type whose name component is being accessed. Double pound parameters denote parameters of a global scope[9]. The parameter ##Factory##, for example, denotes an instance of a factory class that is used to construct the instance of the And type.

```
And'(And_Term.name,
     Replace(And_Term.children,s,s1.s)


m_Factory.And(m_Instance.Name(),
                 m_Instance.States().Replace
                  (v2,v3.result())
                 )
```

Figure 4.5: Portion of a rule and the corresponding ArgoUML implementation.

Figure 4.5 contains an example of how the rule expression in the top of the figure is transformed into the implementation code below it by using the target type/signature mapping. The rule expression contains three variables: AndTerm of type And, s of type State, and s1 of type MicroStep[10]. During code generation these variables become the variable instances m_Instance, v2, and v3 respectively. The first transformation

[8]The use clause for functions and predicates defines how to transform a function or predicate into its implementation.

[9]I excluded ##Factory##'s definition because it is beyond the scope of this thesis.

[10]I excluded the MicroStep mapping because it is similar to the mapping for And.

maps `s1.s` into `v3.result()` and `And_Term.children` into `m_Instance.States()`, which is then used to transform `Replace(And_Term.children,s,s1.s)` into its corresponding expression `m_Instance.States().Replace(v2,v3.result()-)`. Finally, `And_Term.name` is transformed into `m_Instance.Name()`, which is used to complete the transformation of the expression.

Type and Signature Sections as Enablers

As demonstrated in the previous section, the Amalia step analyzer generator uses a parameterized text substitution technique to map required meta operations into implementation specific operations. This capability is possible due to the explicit type and signature section; thus these sections enable the target instantiator to customize a single meta language specification for use with multiple target implementations. Furthermore, these sections provide a clean interface between the semantic rules compiler and the target instantiator.

Chapter 5

Case study

For validation I built a step analyzer generator that supports the new input language and then generated a step analyzer, which supplies StateCharts semantics to UML state diagrams, for use with ArgoUML. With this step analyzer, I was able to write a plugin for ArgoUML that provides two new user functions for UML state diagrams: LTS enumeration and UML state diagram simulation. The LTS enumeration function prints out all the possible configurations that a state diagram can enter and lists the transitions for each configuration. The simulation function allows the user to interactively explore a statechart's execution paths for a selected state diagram. Such execution is animated by highlighting its active states within the user interface.

The creation of the plugin was divided into three steps: creating the `scInterp[UMLSD]` component listed in **Figure 2.12** and described in **Chapter 2.3**, instantiating the analysis functions, and creating the user interaction code. Creating the `scInterp[UMLSD]` component involved creating interfaces that correspond to the syntactic expressions of StateCharts. With these interfaces I was able to write the wrapper classes, which have the appropriate StateCharts interface, for the UML state diagram classes used by ArgoUML. The next step involved writing the language specification for StateCharts and writing the target type/signature mapping that instantiates a step analyzer that is parameterized by `scInterp[UMLSD]`. I composed the step analyzer with two generic analyzer components: `eagerLTS[SA]` to create the LTS enumerator and `sim[SA]` to create the StateCharts simulator. The final step was to create the plugin code for ArgoUML, which adds these two analysis functions to ArgoUML's user interface. This code consists of a plugin class, which is a class that implements the ArgoUML plugin interface and defines menu items that invoke each analysis function.

code section	lines of code	classification	%
`scInterp[UMLSD]`	1314	dependent	55.8
StateCharts language specification	75	reusable	3.2
target mapping for `scInterp[UMLSD]`	84	dependent	3.5
step analyzer	514	generated	21.8
analysis functions from libraries	107	reusable	4.5
ArgoUML menu plugin	262	dependent	11.1

Table 5.1: Code metrics for the ArgoUML plugin.

Table 5.1 contains the lines of code for each artifact produced by creating the StateCharts plugin for ArgoUML. The code is classified as dependent, reusable, or generated. Dependent code had to be written specifically for ArgoUML. Reusable code can be reused to generate step analyzers for a different target tool. Generated code is generated by the Amalia step analyzer generator.

The case study demonstrates Amalia's ability to generate analysis components for languages with complex operational semantics and the ability of these components to integrate tightly into an existing environment. Additionally, **Table 5.1** shows that the Amalia step analyzer generator was able to generate approximately twenty-two percent of the plugin's code. However, **Table 5.1** also shows that the `scInterp[UMLSD]` component makes up the bulk of the ArgoUML plugin. This is to be expected since we had to provide a syntactic mapping between UML state diagrams and StateCharts. ASE's that provide the necessary syntactic data structures for a particular specification language should have a higher percentage of reusable and generated code. For example, if ArgoUML had classes that supported the syntax of StateCharts, I would not have had to create the `scInterp[UMLSD]`. Under such a conditions, the generated code would make up approximately forty-nine percent of the ArgoUML plugin.

Furthermore, this case study validates that the metalanguage specification language is both effective and feasible. This new language is effective at (1 abstracting away implementation details for the SOS rules and (2 expressing the semantics of the complexity of the StateCharts. The presence of a working step-analyze generator and the ArgoUML case study demonstrate the feasibility of the new Amalia input language.

Chapter 6

Discussion

The new additions to Amalia's SOS rule format have considerably extended its expressibility. The original version of Amalia's SOS rule format could only express a subset of the De Simone format rules [12]. Specifically, the original rule format assumed that syntactic types were either atomic or binary and that the format allowed only one relation type. Furthermore, negated premises were not allowed and rules were limited to a maximum of two premises. The new Amalia SOS rule format allows a bounded number of premises and negated premises making our format as expressive as GSOS [3, 1]. Furthermore, the new format allows the source expression of the rule to be the subject of either premises or negated premises[1], which is not allowed by GSOS. Additionally, our rule format supports the extensions to StateCharts found in [23], which add semantics for state references, state history, and transition priority.

6.1 Related Work

As **Chapter 2.2** illustrates, Amalia relies upon SOS. SOS originated from Plotkin's approach to describe the formal semantics of programming and specification languages[26]. SOS has received many extensions and has been applied to a wide variety of applications[1]. The extensions to the Amalia rule language is an application of previous extensions to SOS (GSOS [3], existential and universal quantifiers [32], and negated premises [16]) to

[1]The relation type of the conclusion and the premise cannot match in order to avoid circularity.

the Amalia framework with the goal of providing analysis components for a richer set of languages to ASEs.

The intrinsic ability to generate LTSs from SOS rules has proven useful for other formal language analysis projects. The Process Algebra Compiler(PAC) [7] generates analyzers from SOS rules. The PAC generates code modules that are similar to Amalia's step analyzers. The PAC has been used in two formal analysis tools: Concurrency Workbench of the New Century [8] and MAUTO [5]. However, the code modules that the PAC produces are specialized; both Concurrency Workbench and MAUTO had to be designed around the interface provided by the code modules. Therefore, the PAC is useful for adding support for new formal languages to Concurrency Workbench and MAUTO, but it does not produce analyzers that are easily integrated into third party tools. Furthermore, the PAC's rule format does not support rules with negative premises or enumerated premises, and the PAC does not support metarules.

Other tools like CENTAUR [4], RML [25], and LATOS [20] generate stand-alone analyzers for languages that are described by natural semantics [21]. None of these tools support notations that allow enumerated premises or metarules. Furthermore, many general-purpose theorem provers, e.g. Isabelle [15] and HOL [6], allow SOS rules to be represented declaratively. In particular, a framework developed by Day and Joyce [11] uses an embedding of a notation's semantics in HOL to define how a specification determines a "model," a HOL type by which formalizes a step relation. Of course, theorem provers are themselves stand-alone tools. Their framework, however, uses symbolic functional evaluation to allow certain analyses (e.g., consistency and completeness checking and model checking) without using a theorem prover [10].

In general these projects inhibit tight integration with ASEs and these inhibitions lead to complications to both the ASEs' developers and users. For instance, projects that provide stand-alone tools create distribution problems for the ASEs. Developers must distribute both their ASE and the stand-alone analyzer. Such inconveniences deter the development of well designed and user-friendly ASEs that include formal analysis capabilities.

6.2 Future Work

For future work, I expect to add two features to the Amalia SOS rule format. The first feature is *premise lookahead*, which allows the subject of a premise to be a step from a preceding premise. This is a trivial addition to the syntax and semantics of the rule language; however, this feature places additional requirements upon the target instantiation, which must be investigated. The second feature adds syntax that denotes collecting all the steps from a premise into a single set. We discovered the need for this feature when developing a set of SOS rules for future interval logic [28]. This feature will require further investigation to determine how to elegantly add it to the Amalia SOS rule syntax.

Additionally, work is required to further generalize the format of meta step-analyzers. By further generalizing the format, I hope to standardize conventions for developing target mapping languages and their corresponding target instantiators. Furthermore, I would like improve the format to improve the target instantiators' optimization of step-analyzers.

Appendix A

BNF for the Amalia Specifcation Language

```
Spec     ::= language name is
             TypeSection
             SignatureSection
             RelationSection
             RuleSection
             end name ;
Comment ::= --texttoendofline
```

A.1 Type Section

TypeSection ::= `type_section is`

*TypeDef**

`end type_section ;`

TypeDef ::= `type` *name* [`is` *TypeRefinement*] `;`

TypeRefinement ::= `atomic`

| `set` *name*

| `union` *NameList*

| `tuple (` *Tuple* `)`

NameList ::= *name* | *name* `,` *NameList*

Tuple ::= *Pair* | *Pair* `,` *Tuple*

Pair ::= *name* `:` *name*

A.2 Signature Section

SignatureSection ::= `signature_section is`

*SigDef**

`end signature_section ;`

SigDef ::= *FunctionDef* | *PredicateDef*

FunctionDef ::= `function` *name* `(` *Tuple* `) return` *name* `;`

PredicateDef ::= `predicate` *name* `(` *Tuple* `) ;`

A.3 Relation Section

RelationSection ::= `relation_section is`

*RelationDef**

`end relation_section ;`

RelationDef ::= `step` *name* `is` *name* `to` *name* `;`

A.4 Rule Section

RuleSection ::= `rule_section is`

*RuleDef**

`end rule_section ;`

```
RuleDef        ::= Rule | Metarule

Rule           ::= rule name (Pair ) RuleBody

Metarule       ::= meta (Pair )

                   ForClause+

                   rule name RuleBody

ForClause      ::= for each name in name; [SuchClause]

SuchClause     ::= such that PredicateList

RuleBody       ::= [WhereClause] is

                   [LetClause]

                   [WithClause]

                   [GuardClause]

                   ConcludeClause

                   end name ;

LetClause         ::= let AssignmentList

WithClause        ::= with PremiseClause+

GuardClause       ::= without PremiseClause+

ConcludeClause ::= conclude name name to Reference


Assignment      ::= Variable = Reference ;

AssignmentList ::= Assignment | Assignment AssignmentList
```

```
Reference     ::= Variable | Literal | Function

Variable      ::= name | name . Variable

Literal       ::= name' ( ReferenceList )

Function      ::= name ( ReferenceList )

ReferenceList ::= Reference | Reference , ReferenceList

Predicate     ::= [not]{Equality | Method}

Equality      ::= Reference = Reference

Method        ::= name ( ReferenceList )

PredicateList ::= Predicate ; | Predicate , PredicateList


PremiseClause ::= StepClause [WhereClause] [LetClause]

WhereClause   ::= where PredicateList
```

Appendix B

Specification for the StateCharts Language

```
language Sc is
  – This section declares the meta syntax.
  type_section is


    type Event is Atomic;
    type Event_Set is set Event;


    type State_Name is Atomic;


    type Transition is
      tuple (
      source : State_Name,
      required : Event_Set,
      prohibited : Event_Set,
      action : Event_Set,
      destination : State_Name
      );
```

```
type Transition_Set is set Transition;

type Basic is atomic;
type Or;
type Inner;
type Outer;
type And;

type State is union Basic, Or, Inner, Outer, And;
type State_Set is set State;

type Or is
   tuple (
   name : State_Name,
   children : State_Set,
   trans : Transition_Set,
   n_init : State_Name,
   n_curr : State_Name
   );

type Inner is
   tuple (
   name : State_Name,
   children : State_Set,
   trans : Transition_Set,
   n_init : State_Name,
   n_curr : State_Name
   );

type Outer is
```

```
        tuple (
        name : State_Name,
        children : State_Set,
        Trans : Transition_Set,
        n_init : State_Name,
        taken : Transition
        );

    type And is
        tuple (
        name : State_Name,
        children : State_Set
        );

    type MicroStep is
        tuple (
        required : Event_Set,
        prohibited : Event_Set,
        action : Event_Set,
        s : State
        );

    type ClockStep is tuple (s : State);
    type ClockStep_Set is set ClockStep;
end type_section;

signature_section is

    function Referent (From : State_Set,
                       Item : State_Name)
```

```
                    return State;

    function Default (Item :State) return State;

    function Actions (Item : State_Set) return Event_Set;
    function Prohibits (Item : State_Set) return Event_Set;

    predicate No_Conflicts (Left : Event_Set, Right : Event_Set);
    predicate No_Events (Item : Event_Set);

    function Replace (In_Set : State_Set, Old : State, New : State)
                    return State_Set;

    function Remove_State (From : State_Set, Item : State)
                            return State_Set;

    function Remove_Events (From : Event_Set, Items :Event_Set)
                              return Event_Set;

    function All_States (Item : ClockStep_Set)
                              return State_Set;

end signature_section;

relation_section is
    Step Micro is State to MicroStep;

    Step Clock is State to ClockStep;

end relation_section;
```

```
rule_section is
   meta (Or_Term : Or)
      for each t in Or_Term.trans;
         such that t.source = Or_Term.n_curr;
   rule Or1 is Conclude Micro
      Or_Term to
         MicroStep'(
            t.required,
            t.prohibited,
            t.action,
            Outer'(Or_term.name,
               Or_term.children,
               Or_term.trans,
               Or_term.n_init,
               t
               )
      );
   end Or1;

rule Or2 (Or_Term : Or) is
   let
      s = Referent(Or_Term.children,Or_Term.n_curr);
   with
      Micro s to s1;
   conclude Micro
      Or_Term to MicroStep'(
         s1.required,
         s1.prohibited,
         s1.action,
      Inner'(Or_Term.name,
         Replace(Or_Term.children,s, s1.s),
         Or_Term.trans,
         Or_Term.n_init,
         Or_Term.n_curr
         )
      );
   end Or2;

rule Inner1 (Inner_Term : Inner) is
   let
      s = Referent(Inner_Term.children,Inner_Term.n_curr);
   with
      Micro s to s1;
   conclude Micro
      Inner_Term to
         MicroStep'(
            s1.required,
            s1.prohibited,
            s1.action,
            Inner'(Inner_Term.name,
             Replace(Inner_Term.children,s,s1.s),
             Inner_Term.trans,
             Inner_Term.n_init,
             Inner_Term.n_curr
            )
      );
end Inner1;

meta (And_Term : And)
```

```
  for each s in And_Term.children;
rule And1 is
  with
    Micro s to s1;
  where
    No_Conflicts(s1.prohibited,
     Actions(Remove_State(And_Term.children, s))),
    No_Conflicts(s1.action,
     Prohibits(Remove_State(And_Term.children, s))),
    No_Conflicts(s1.required,
     Prohibits(Remove_State(And_Term.children, s)));
  conclude Micro
    And_Term to MicroStep'(
     Remove_Events(s1.required,
       Actions(Remove_State(And_Term.children, s))
       ),
     s1.prohibited,
     s1.action,
     And'(And_Term.name,
       Replace(And_Term.children,s,s1.s))
       );
end And1;

rule cBas (Basic_Term : Basic) is
  conclude Clock
    Basic_Term to ClockStep'(Basic_Term);
end cBas;

rule cOuter (Outer_Term : Outer) is
  let
```

```
   r = Referent(Outer_Term.children,
      Outer_Term.taken.Destination);
conclude Clock
   Outer_Term to ClockStep'(
      Or'(Outer_Term.name,
         Replace(Outer_Term.children, r, Default(r)),
         Outer_Term.trans,
         Outer_Term.n_init,
         Outer_Term.taken.Destination
         );
end cOuter;

rule cOr (Or_Term : Or) is
   without
      Micro Or_Term to f1;
   where
      No_Events(f1.required),
      No_Events(f1.prohibited),
      No_Events(f1.action);
   conclude Clock
      Or_Term to ClockStep'(
         Or'(Or_Term.name,
         Or_Term.children,
         Or_Term.trans,
         Or_Term.n_init,
         Or_Term.n_curr
         ));
end cOr;

rule cInner (Inner_Term : Inner) is
```

```
            let
              s = Referent(Inner_Term.children,Inner_Term.n_curr);
            with
              Clock s to s1;
            conclude Clock
              Inner_Term to MacrStep'(
                Or'(Inner_Term.name,
                Replace(Inner_Term.children, s, s1.s),
                Inner_Term.trans,
                Inner_Term.n_init,
                Inner_Term.n_curr
                ));
          end cInner;
          rule cAnd (And_Term : And) is
            let
              s = And_Term.children;
            with
              Clock s to s1;
            without
              Micro And_Term to f1;
            where
              No_Events(f1.required),
              No_Events(f1.prohibited),
              No_Events(f1.action);
          conclude Clock
            And_Term to ClockStep'(And'(And_Term.name,s1));
        end cAnd;
      end rule_section;
    end SC;
```

Appendix C

Glossary

actions(t) Function that denotes the set of action events for transition, t.

`all_states(s1)` Function that takes the set of macro steps, `s1`, and returns the set of extracted `State` expressions from each member of `s1`.

default(S, t) Function that denotes statechart, target(S, t), in its default configuration.

EnvironmentEvents(S) Function that denotes the events within the environment that are a result of fired transitions within the set of sub-states, S.

EnvironmentProhibits(S) Function that denotes the events prohibited from being within the environment that is the result of the fired transitions within the set of sub-states, S.

`free_steps(mStep)` Predicate that asserts that all the event sets in the micro step `mStep` are empty.

guards(t) Function that denotes the set of guard events for transition, t.

`microstep_and(a,s,s1)` Function that returns a micro step with the event sets of the micro step, `s1`, and it contains an and-state identical to `a` with the exception that `s` in `a.children` is replaced with `s1`.

name(s) Function that denotes the name of the statechart, s.

`new_Inner(i,s,s1)` Function that returns an inner-state identical to `i` with the exception that `s` in `i.children` is replaced with `s1`.

NoConflicts(S, R, P, A) Predicate that asserts that the set of sub-states, S, does not contain any fired transitions that disable events within $R \cup A$; or require or produce events within P.

`no_conflicting_transitions(a,s,s1)` Predicate that asserts that `s1` does not contain any events that do not already exist in the environment due to transitions fired from outer-states within

$$\texttt{a.children} \setminus \texttt{s}$$

`referent(children,name)` Function that returns the state in `children` with the name, `name`.

source(t) Function that denotes the name of the source statechart for transition, t.

target(S, t) Function that denotes the statechart s in the set, S, such that the name of s matches the name of the target of transition, t.

triggers(t) Function that denotes the set of required events for transition, t.

`to_Outer(o,t)` Function that returns a micro step with the same trigger, guard, and action event sets as the transition, `trans`, and it contains an outer-state with the same name, children, transitions, and initial state as `o`. Additionally in the place of a current state the outer-state has the transition, `trans`.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] L. Aceto, W. Fokkink, and C. Verhoef. Structural operational semantics, 1999.

[2] D. Batory and S. OMalley. The design and implementation of heirarchical software systems with reusable components. *ACM Trans. Softw. Eng. Meth.*, 1(4):355–398, October 1992.

[3] B. Bloom, S. Istrail, and A. R. Meyer. Bisimulation can't be traced. *J. Assoc. Comput. Mach.*, 42:232–268, 1995.

[4] P. Borras, D. Clement, Th. Despeyrouz, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: The system. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (PSDE)*, volume 24, pages 14–24, New York, NY, 1989. ACM Press.

[5] G. Boudol, V. Roy, R. de Simone, and D. Vergamini. Automatic verification methods for finite state systems. In *CAV Grenoble : Process Calculi, from Theory to Practice: Verification Tools*, volume LNCS 407, 1989.

[6] A. J. Camilleri. Mechanizing csp trace theory in higher order logic. *IEEE Trans. Softw. Eng.*, 16(9):993–1004, September 1990.

[7] R. Cleaveland, E. Madelaine, and S. Sims. A front-end generator for verification tools. In *Proc. of the Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'95)*, volume 1019, pages 153–173, 1995.

[8] Rance Cleaveland and Steven T. Sims. Generic tools for verifying concurrent systems. *Science of Computer Programming*, 42(1):39–47, 2002.

[9] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[10] Nancy A. Day and Jeffrey J. Joyce. Symbolic functional evaluation. In *12th Internatl. Conf. Theorem Proving in Higher Order Logics*, pages 341–358. Springer, 1999. LNCS 1690.

[11] Nancy A. Day and Jeffrey J. Joyce. A framework for multi-notation specification and analysis. In *Fourth IEEE Internatl. Conf. Requirements Engineering*, June 2000.

[12] Robert de Simone. Higher-level synchronising devices in meije–sccs. *Theoretical Computer Science*, 35:245–267, 1985.

[13] L. K. Dillon and R. E. K. Stirewalt. Inference graphs: A computational structure supporting generation of customizable and correct analysis components. *IEEE Transactions on Software Engineering*, 29(2), February 2003.

[14] Eric Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[15] B. Gramlich and F. Pfenning. Strategic principles in the design of isabelle. In Work. Strategies in Automated Deduction, July 1998.

[16] J. F. Groote. Transition system specifications with negative premises. *Theoretical Computer Science*, 118:263–299, 1993.

[17] D. Harel, A. Pnueli, J. P. Schmidt, and R. Sherman. On the formal semantics of statecharts. In *Second Symposium on Logic in Computer Science*, 1987.

[18] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[19] David Harel and Orna Kupferman. On object systems and behavioral inheritance.

[20] P. Hartel. Latos – a lightweight animation tool for operational semantics, 1997.

[21] G. Kahn. Natural semantics. In K. Fuchi and M. Nivat, editors, *Programming of Future Generation Computers*, pages 237–258. Elsevier Science Publishers B.V. North Holland, 1998.

[22] Diego Latella, Istvan Majzik, and Mieke Massink. Towards a formal operational semantics of UML statechart diagrams. In *Proc. FMOODS'99, IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems, Florence, Italy, February 15-18, 1999*. Kluwer, 1999.

[23] G. Lüttgen, M. von der Beeck , and R. Cleaveland. A compositional approach to statecharts semantics. In *Proc. of ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE'2000)*, 2000.

[24] Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[25] Mikael Pettersson. A compiler for natural semantics. In *Computational Complexity*, pages 177–191, 1996.

[26] G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[27] A. Pnueli and M. Shalev. What is in a step: On the semantics of Statecharts. In *TACS'91*, volume 526 of *LNCS*, pages 244–264. Springer-Verlag, 1991.

[28] Y. S. Ramakrishna, P. Michael Melliar-Smith, Louise E. Moser, Laura K. Dillon, and George Kutty. Interval logics and their decision procedures, part I: An interval logic. *Theoretical Computer Science*, 166(1–2):1–47, October 1996.

[29] Jason Elliot Robbins. *Cognitive Support Features for Software Development Tools.* PhD thesis, University of California, Irvine, 1999, 1999.

[30] Y. Smaragdakis and D. Batory. Implementing layered designs with mixin layers. In *Proc. of the 12th European Conference on Object-oriented Programming*, 1998.

[31] R. E. K. Stirewalt and L. K. Dillon. A component-based approach to building formal analysis tools. In *Proc. of the 2001 International Conference on Software Engineering (ICSE'2001)*, 2001.

[32] Chris Verhoef. A congruence theorem for structured operational semantics with predicates and negative premises. In *International Conference on Concurrency Theory*, pages 433–448, 1994.