Mapping
Template Semantics to SMV

by

Yun Lu

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2004

©Yun Lu 2004

AUTHOR'S DECLARATION FOR ELECTRONIC SUBMISSION OF A THESIS

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Template semantics is a template-based approach to describing the semantics of model-based notations, where a pre-defined template captures the notations' common semantics, and parameters specify the notations' distinct semantics. In this thesis, we investigate using template semantics to parameterize the translation from a model-based notation to the input language of the SMV family of model checkers. We describe a fully automated translator that takes as input a specification written in template semantics syntax, and a set of template parameters, encoding the specification's semantics, and generates an SMV model of the specification. The result is a parameterized technique for model checking specifications written in a variety of notations. Our work also shows how to represent complex composition operators, such as rendezvous synchronization, in the SMV language, in which there is no matching language construct.

Acknowledgements

I would like to express my most sincere gratitude to my supervisor, professor Nancy A. Day, without whose insightful guidance and invaluable help this thesis would not be possible. I really appreciate the time and energy she had dedicated to my thesis work. I feel fortunate to be under her supervision.

I would like to give great thanks to professor Joanne M. Atlee, who also played an important role in my research work; I am grateful for her guidance throughout this project. As my thesis reader, her insightful suggestions were invaluable in the production of my thesis.

I would like to thank Professor John Thistle, for taking time to read my thesis; his valuable feedback has improved my thesis greatly.

Many thanks to Jianwei Niu, for helping me to understand the background of my thesis, and for helping me with the development of the case studies.

Thanks also go to Davor Svetinovic and Vlad Ciubotariu for their valuable discussions on my research work.

Thanks to my friends, Yaya Yang, Jei Zhang, Yuan Peng, and Lixiao Wang, for making my two years at UW enjoyable.

Finally, I would like to thank my parents, my sister, and my husband, for their endless support and encouragement for my studies at Waterloo.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this thesis, we describe a method for translating model-based specification notations into the input language of the SMV family of model checkers using template semantics descriptions of a notation's meaning. In this chapter, we provide the motivation for creating our translator, give an overview of our method, and describe how we evaluate our work. Finally, we list the contributions of this thesis. A summary of this thesis also appears in [25].

1.1 Motivation

Reactive systems, in which the inputs may arrive in an endless and unexpected sequence, are complex and difficult to specify, and the errors in the specifications of these systems are usually difficult to discover. **Model checking** [10] is a technique that involves building a finite model of a system and checking that a desired property holds of that model. Given a finite model of a reactive system, a list of properties of the system, and sufficient resources, a model checker can automatically verify whether these properties hold in the model. If a property holds, the model checker returns true, otherwise, it returns a counterexample showing a trace of the model in which the property does not hold. The counterexample can be used to analyze errors in the model, and the specification can be revised accordingly. Model checking is a promising technique for software requirements engineers to use to verify the specification of a reactive system rigorously, and to detect subtle errors that might be difficult and time-consuming to find in the implementation.

When verifying a model, model checking tools exhaustively explore the state space of the whole model. When the model has many components that interact with each other, or the model has data structures that assume many different values, this exhaustive exploration of the state space can cause the **state space explosion problem**. Model checkers usually have their own state-space representation, reduction, and exploration algorithms to deal with the state space explosion problem. For example, SMV [28] uses BDDs to represent the state space symbolically, and Spin [20] use partial order reduction to reduce the state space. Since each model checker emphasizes different aspects of the model checking process or has a particular domain of applicability, sometimes it is useful to model a specification in different model checkers' input languages to verify different aspects of a system.

Since each model checker has its own input language, verification of a software model using a model checker requires software requirements engineers to write the model in the model checker's input language. Most model checkers have relatively simple and restricted input languages to facilitate automatic verification. The process of modelling a system directly using the input languages of model checkers is time consuming and not straightforward when compared to modelling a system using a model-based requirements notation.

Model-based requirements notations, which allow a user to specify an abstract model that describes the possible execution steps (next-state relation) that the system can take, are attractive to software requirements engineers for modelling the dynamic behaviours of reactive systems. Model-based notations include process algebras (e.g., CSP [19], CCS [29], basic LOTOS [21]), and statecharts variants (e.g., statecharts [17] [1], RSML [24], and STATEMATE [18]). They provide composition operators, which enable requirements engineers to decompose large problems into modules.

To facilitate requirements engineers' ability to model check their specifications, work has been done on constructing translators from model-based requirements notations to the input languages of model checkers. Examples of existing translators for specific notations include: SCR to EMC model checker [2], RSML to SMV [8], and SCR to SPIN and SMV [4]. But if we want to translate specifications of m model-based notations to the input languages of n model checkers, in order to take advantage of their different optimization techniques, we need to build $m \times n$ different translators. Furthermore, a notation may change slightly over time, so the translator

[1] In this thesis, the word "statecharts" always refers to Harel's original statecharts defined in [17].

associated with the evolved notation also needs to change.

To reduce the number of translators and to facilitate analysis using multiple tools, intermediate languages have been introduced, such as SAL [3], IF [6], Action Language [7], Bandera Intermediate Representation (BIR) [11], and CDL [23]. In most of these cases, there are translators from several notations to the intermediate language, and from the intermediate language to the input languages of verification tools. Using this method, we need $m + n$ translators: m translators translate specifications of these m notations to the intermediate language, and n translators translate the intermediate language to the input languages of n model checkers.

Another approach is to generate a model or analysis tool from a description of a notation's semantics. Day and Joyce [14] embedded the semantics of a notation in higher-order logic and produced a next-state relation in logic for a specification. Pezzè and Young [33] embedded the semantics of notations into hypergraph rules. Dillon and Stirewalt defined operational semantics for process-algebra and temporal-logic notations, from which an inference graph for the specification can be generated [15]. The problem with these approaches is that it is very difficult to define the semantics of a notation.

Template semantics [31] is a new technique aimed at easing the construction of notation-specific analysis tools. It provides a template that captures the common semantics of model-based notations, and a list of parameter predicates that represent the distinct semantics of these notations. It offers a rich set of composition operators that describe how components execute and share information. By choosing values for the parameter predicates, template semantics can be used to describe succinctly the meanings of many model-based notations. Furthermore, template semantics can be used to represent new, customized, or evolving notations.

Compared to the approaches that use intermediate languages, template semantics is parameterized and contains a richer set of composition operators. Users can play with the template parameters, so that the specification can be assigned different meanings. Compared to the approaches in [14, 33, 15], template semantics allows one to specify only how a notation differs from the common features of model-based notations rather than providing a complete semantic description.

Niu implemented a tool that takes as input a description of a notation in template semantics in higher-order logic [30], and a specification in that notation, and produces a next-state relation in logic for the specification. This approach is based on the work of Day and Joyce [14] and

uses a tool called Fusion [13, 12], which symbolically evaluates the semantic definition of a specification into a next-state relation in logic, and properties of the specification are checked using Fusion's model checker. But the next-state relation generated by Fusion does not retain the structure of the original specification, which limits the state space reduction techniques that can be applied. Furthermore, Fusion's model checker does not have optimization techniques, so model checking properties of a specification may not be efficient.

1.2 Thesis Description

In this thesis, we investigate using template semantics to parameterize the translation from model-based requirements notations to the input language of the SMV family of model checkers (Cadence SMV [28] and NuSMV [9]). Our thesis statement is:

> **Using template semantics as a parameterized intermediate language, specifications in different notations can be translated to SMV models that retain the structure of the original specifications, and have state spaces comparable to those of the original specifications.**

To validate our claim, we have built a translator, called Express, that takes as input a specification in a notation, written in template semantics' syntax[2], and a set of parameter values for the template predicates, encoding the notation's semantics, and produces an SMV model for the given specification. By choosing different parameter values that represent different notations, specifications in different notations can be translated to SMV.

In our approach, instead of writing a translator for each notation, the user chooses from a set of pre-defined parameter values for the parameter predicates of template semantics for each notation. Our single translator to SMV can handle many notations.

By translating from template semantics descriptions of a notation's semantics to SMV, we can take advantage of the optimization techniques of the SMV family of model checkers, and model check large and complex specifications. However, compared to the work of Niu[30], the user must choose from a fixed collection of parameter values.

[2] A representation of a specification in template semantics' syntax is mostly just a textual representation of a graphical language.

The pre-defined parameter values are hard-coded in Express. Our translator supports all of the template-parameter values and the composition operators that were used in [31] to describe the semantics of basic transition systems (BTS) [27], CSP [19], CCS [29], basic LOTOS [21], statecharts [17], RSML [24], and STATEMATE [18], plus some additional values that we found interesting. We can even create new notations by combining existing parameter values without any change to our translator.

1.3 Evaluation

In translating template semantics to SMV, our goal was to handle a large collection of notations, without sacrificing analysis efficiency (i.e., increasing the size of the state space). We evaluated our work based on the following criteria:

1. The SMV code should match the modularity of template semantics, so that the amount of work needed to add new template-parameter values and composition operators would be minimal.

2. The SMV code should match the composition structure of the specification to make it easy to check the correctness of our translation.

3. The translator should not introduce intermediate execution steps or variables, so that the counterexamples output by SMV would be in terms of the original specification.

4. The SMV model's state space should match as closely as possible the state space of the original specification if a model checking tool had been built for it directly.

We evaluated our work with two case studies, a heating system, and a single lane bridge, chosen to exercise a broad range of composition operators. Using SMV's simulation and model checking capability, we checked that our translator produces a model whose behaviour matches the expected behaviour of the original specification.

1.4 Contributions

The main contributions of this thesis are:

- A parameterized translator from template semantics to the SMV family of model checkers for model checking specifications written in a variety of model-based requirements notations.
- A description of how to model a rich set of composition operators – including rendezvous, environmental synchronization, sequence, choice, interrupt – within the fairly simple language features of the SMV input language.
- Support for the generation of new notation-specific translators by simply selecting different combinations of template-parameter values and composition operators. Therefore, it can be used for many more notations than originally described in template semantics.
- A modular approach to translation that limits the scope of changes needed to implement a new parameter value or composition operator, making it easier to construct a new notation-specific translator from template semantics than from a requirements notation directly.
- Validation of the claim of the authors of template semantics [31] that using template semantics considerably reduces the effort involved in constructing notation-specific analyzers.

1.5 Thesis Organization

In Chapter 2, we provide background on template semantics and the SMV language. In Chapter 3, we describe our method of translating template semantics to the input language of SMV. We briefly discuss the implementation of the translator in Chapter 4. We evaluate our translation with two case studies in Chapter 5, and conclude with a summary, discussion of limitations and future work in Chapter 6. We give the SMV models produced by our translator for the two case studies in the Appendices.

Chapter 2

Background

In this chapter, we provide the background needed to understand the rest of the thesis. We briefly describe template semantics, using one of our case studies, the heating system. Then we give an overview of the features we used in the input language of the SMV family of model checkers.

2.1 Case Study: Heating System

We use the heating system example, (originally from [5] and revised in [13]), as a case study to evaluate our translator. The system consists of a room to be heated, a furnace, and a controller. The room has a valve that controls airflow into the room: the valve can be open, half open, or closed. The room also has a sensor that measures the room's temperature and a thermostat by which a user can set the desired temperature. If the room temperature is lower than the desired temperature, the system warms the room by opening the valve, to increase the inflow of heated air; if the room continues to be too cold, the room requests heat. The system behaves analogously when the room temperature is too hot. The controller activates and deactivates the furnace on request from the room.

Figure 2.1 declares the variables needed for the specification of the heating system. Variable *valvePos* is the valve position of the room, which has the type of integer range from 0 to 2, and initial valve 0, which represents that the valve is closed; 1 represents that the valve is half open; and 2 represents that the valve is open. The variables *furnaceStartUp*, *waitedForWarm*, and *waitedForCool* are counters that make the system wait for the furnace to start, and wait for

```
/*variables*/
var valvePos: range(0,2) = 0;
var furnaceStartup: range(0,5) = 0;
var waitedForWarm : range(0,5) = 0;
var waitedForCool : range(0,5) = 0;
var requestHeat : bool = false;

/*environmental variables*/
evar setTemp : range(16,24);
evar actualTemp : range(10,30);

/*environment events : type env*/
event heatingSwitchOn, heatingSwitchOff, userReset, furnaceFault : env;

/*internal events : type intee*/
event activate, deactivate, furnaceReset, furnaceRunning : intee;

/*constants and macros*/
macro furnaceTimer, warmUpTimer, coolDownTimer : 5;
macro tooCold : (setTemp - actualTemp) > 2;
macro tooHot : (actualTemp - setTemp) > 2;
```

Figure 2.1: Heating System Variable Declarations

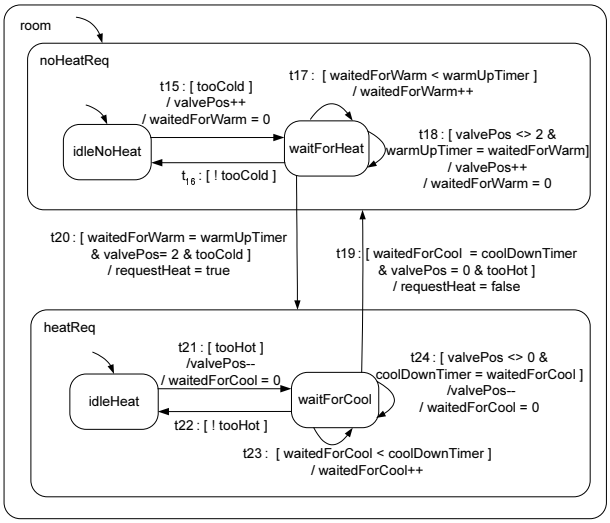


Figure 2.2: Room Composed HTS

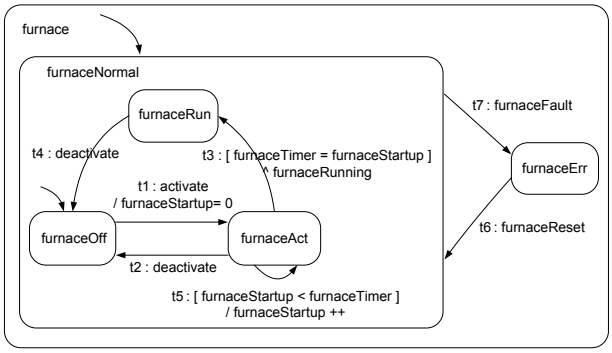


Figure 2.3: Furnace HTS

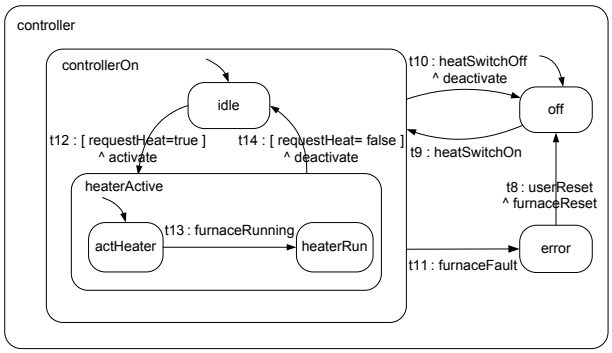


Figure 2.4: Controller HTS

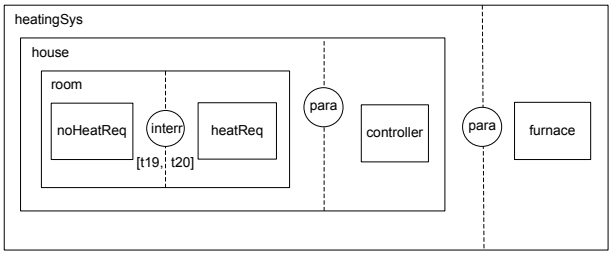


Figure 2.5: Heating System

a temperature change to take effect before trying another valve setting. Variable *requestHeat* is a boolean variable to indicate whether the room needs heat. The user can set the desired temperature using variable *setTemp*. The actual temperature (variable *actualTemp*) is sensed from the environment by the sensor. Constants *furnaceTimer*, *warmUpTimer*, and *coolDownTimer* represent the time needed to start up the furnace, and how long the system waits for a temperature change to take effect. Condition *tooCold* indicates the room is too cold when the difference between the desired temperature (*setTemp*) and the actual temperature (*actualTemp*) is greater than the constant 2. Condition *tooHot* indicates the room is too hot when the difference between the actual temperature (*actualTemp*) and the desired temperature (*setTemp*) is greater than the constant 2.

Figure 2.2, 2.3, and 2.4 show the state machine representations for the components *room*, *furnace*, and *controller*, which are composed to create the heating system (Figure 2.5). The elements of the state machines and composition operators used will be explained in the next section.

2.2 Template Semantics

Template semantics is a template-based approach to structuring the semantics of model-based notations. The basic computation model is a nonconcurrent, hierarchical transition system (HTS). An HTS is an extended state machine, adapted from basic transition systems [27] and statecharts [17]. A specification is a hierarchical composition of HTSs via composition operators. Concurrency is introduced via composition.

2.2.1 Syntax of HTS

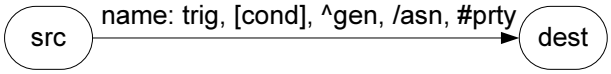
A **hierarchical transition system** (HTS) is an 8-tuple, $\langle S, S^I, S^F, S^H, E, V, V^I, T \rangle$, where

- S is a finite set of states.
- S^I and S^F are predicates describing the set of initial states and final states.
- S^H defines the state hierarchy.
- E is a finite set of events.

- V is a finite set of typed variables.
- V^I is a predicate that indicates the initial values of variables.
- T is a finite set of transitions.

All system elements (states, events, and variables) have unique names. The state hierarchy consists of two kinds of states: **super-states**, which contain other states, and **basic states**, which contain no other states. A super-state has a default child state that is entered when the super-state is a transition's destination.

A transition has the form,



where $src, dest \subseteq S$ are the transition's source and destination states, respectively; $name$ is the transition's name, which is unique throughout the specification; $trig \subseteq E$ is zero or more triggering events; $cond$ is a predicate over V; $gen \subseteq E$ are generated events; asn are assignments to some variables in V; and $prty$ is the transition's optional explicitly-defined priority.

Figure 2.3 shows the HTS *furnace* of the heating system, which has six states. The root state, *furnace*, which is a super-state, has two children, a default child state *furnaceNormal*, and a basic state *furnaceErr*; *furnaceNormal* itself is a super-state which has three children, a default basic child state *furnaceRun*, and two basic states *furnaceOff*, and *furnaceAct*.

A state is **active** when the HTS is in the state, or the HTS is in any of the state's descendant states. A state has a **rank** associated with it, which is the distance between the state and the root state. The rank of root state is 0, and the rank of a state is the rank of its parent state increased by 1. For example, the rank of *furnace* is *0*, and the rank of *furnaceRun* is *2*.

A transition's **scope** is the lowest common ancestor state of the transition's source and destination states. For example, the scope of *t1* is state *furnaceNormal*, and the scope of *t7* is state *furnace*.

2.2.2 Semantics of HTS

Template semantics uses **snapshots** to collect information about the system at observable points in its execution. A snapshot is an 8-tuple

$$\langle CS, IE, AV, O, CSa, IEa, AVa, Ia \rangle$$

that captures the current states CS, the current internal events IE, the current variable values AV, and the current outputs O to be communicated to the environment. CSa, AVa, IEa, Ia are auxiliary elements that accumulate data about states, variables values, and internal and external events, respectively. Notations use these auxiliary elements for different purposes.

There are two types of events in template semantics: external events, which are input events from the environment of the specification; and internal events, which are events generated by the specification. The internal events can be divided into two categories, the internal events used by the system only ($intern\_ev(E)$), and the internal events that can be sensed outside the system ($extern\_ev(E)$).

In template semantics, a **step** moves an HTS from one snapshot to a successor snapshot. A **micro-step** results from executing exactly one transition. A **macro-step** is a sequence of zero or more micro-steps that is initiated by new inputs from the environment.

The semantics of an HTS is represented as a collection of parameterized definitions that, taken together, describe allowable steps between snapshots. The parameterized definitions are:

- ***micro_step*** – a step between consecutive snapshots, due to the execution of at most one transition per HTS.
- ***macro_step*** – a sequence of zero or more micro-steps.
- ***enabled_trans*** – computes the set of transitions enabled by the current snapshot's states, events, and variable values.
- ***execute*** – constrains the set of transitions that execute in the next snapshot.
- ***apply*** – applies the executing transitions' actions (i.e., generated events and variable assignments) to the current snapshot, to derive the next snapshot.
- ***reset*** – resets the current snapshot with new inputs at the beginning of a macro-step.

Parameter values instantiate these definitions to create a notation-specific step semantics. There are 22 template parameters, which are shown in Table 2.1.

Snapshot Elements	Start of Macro-step $reset\_XX$	Micro-step $next\_XX$
CS	$reset\_CS(ss, I)$	$next\_CS(ss, \tau, CS')$
IE	$reset\_IE(ss, I)$	$next\_IE(ss, \tau, IE')$
AV	$reset\_AV(ss, I)$	$next\_AV(ss, \tau, AV')$
O	$reset\_O(ss, I)$	$next\_O(ss, \tau, O')$
CSa	$reset\_CSa(ss, I)$	$next\_CSa(ss, \tau, CSa')$
IEa	$reset\_IEa(ss, I)$	$next\_IEa(ss, \tau, IEa')$
Ia	$reset\_Ia(ss, I)$	$next\_Ia(ss, \tau, Ia')$
AVa	$reset\_AVa(ss, I)$	$next\_AVa(ss, \tau, AVa')$
Additional Parameters	$en\_states(ss, \tau)$ $en\_events(ss, \tau)$ $en\_cond(ss, \tau)$ $resolve\_conflicts(vv_1, vv_2, vv)$ $pri(\Gamma)$ $macro\_semantics$	

Table 2.1: Template Parameters (values to be provided by users)

- The column ***reset_XX*** lists the parameters used in template definition ***reset***: each parameter is a function that resets a snapshot element *XX* in snapshot ss, removing old data and incorporating new system inputs I.
- The column ***next_XX*** lists the parameters used in ***apply***: each parameter is a predicate that constrains how a snapshot element *XX* is updated with respect to transition τ's actions.
- Predicates ***en_states(***ss***,***τ***)***, ***en_events(***ss***,***τ***)***, and ***en_cond(***ss***,***τ***)*** are used in ***enabled_trans*** to specify when a transition τ is enabled with respect to its source state(s), its triggering event(s), and its enabling condition(s).
- Predicate ***resolve_conflicts(***vv_1***,***vv_2***,***vv***)*** defines a notation's policy for resolving conflicts in variable assignments. Variable assignment vv is the resolution of variable assignments vv_1, and vv_2.
- Parameter ***pri*** determines the priority scheme among enabled transitions, where Γ is a set

of transitions. *pri* returns the highest priority transitions in Γ.

- Parameter ***macro-semantics*** determines whether the macro-step semantics is simple or stable. In **simple** semantics, every macro-step is either a micro-step or an idle step, and the snapshot is reset with every step. Simple semantics can be either **diligent**, in which enabled transitions have priority over an idle step, or **nondiligent**. In **stable** semantics, a macro-step is a maximal sequence of micro-steps, starting with a reset snapshot and ending with a **stable** snapshot, in which no transition is enabled.

Table 2.2 (from [32]) provides the semantics of seven notations concisely in template semantics. For example, we used the semantics of STATEMATE for the heating system. STATEMATE uses snapshot elements CS, IE, AV, O, and Ia, but does not use CSa, IEa, and AVa. The parameter values for STATEMATE for each needed template parameter are explained in the following list:

- $reset\_CS : ss.CS$
  At start of each macro-step, snapshot element CS is unchanged.

- $reset\_IE : \phi$
  At start of each macro-step, snapshot element IE is emptied of any old internal events.

- $reset\_Ia : I.ev$
  At start of each macro-step, snapshot element Ia is set to the input events, which is I's ev field.

- $reset\_O : \phi$
  At start of each macro-step, snapshot element O is emptied of any old internal events.

- $reset\_AV : assign(ss.AV, I.var)$
  At start of each macro-step, in snapshot element AV, the environment variables take values from the input variables, which is I's var field, and other variables keep the same values.

- $next\_CS : CS' = entered(dest(\tau))$
  When a transition executes, snapshot element CS is set to be the default-state(s) of the transition's destination state(s).

Table 2.2: Template Parameter Values for Various Notations

Parameter	CCS	CSP	Basic LOTOS	BTS	statecharts [17]	RSML	STATEMATE
reset_CS(ss, I)	$ss.CS$				$ss.CS$		
next_CS(ss, τ, CS')	$CS' =$ *entered* $(dest(\tau))$						
reset_CSa(ss, I)	n/a				$ss.CS$	n/a	n/a
next_CSa(ss, τ, CSa')	n/a				$CSa' = \emptyset$	n/a	n/a
en_states(ss, τ)	*src*$(\tau) \subseteq ss.CS$				*src*$(\tau) \subseteq ss.CSa$	*src*$(\tau) \subseteq ss.CS$	
reset_IE(ss, I)	n/a		n/a	n/a	$\emptyset$		
next_IE(ss, τ, IE')	n/a		n/a	n/a	$IE' = ss.IE \cup$ *gen*(τ)	$IE' =$ *gen*(τ) $\cap$ *intern_ev(E)*	$IE' =$ *gen*(τ)
reset_IEa(ss, I)	n/a		n/a	n/a	n/a	n/a	n/a
next_IEa(ss, τ, IEa')	n/a		n/a	n/a	n/a	n/a	n/a
reset_Ia(ss, I)	$I.ev$		$I.ev$	n/a	$I.ev$		
next_Ia(ss, τ, Ia')	$Ia' = ss.Ia \cup$ *gen*(τ)		true	n/a	$Ia' = ss.Ia$	$Ia' = \emptyset$	
en_events(ss, τ)	*trig*$(\tau) \subseteq ss.Ia$		*trig*$(\tau) \subseteq ss.Ia$	n/a	*trig*$(\tau) \subseteq ss.IE \cup ss.Ia$		
reset_O(ss, I)	$\emptyset$		n/a	n/a	$\emptyset$		
next_O(ss, τ, O')	$O' =$ *gen*(τ)		n/a	n/a	$O' = ss.O \cup$ *gen*(τ)	$O' = ss.O \cup$ (*gen*(τ) $\cap$ *extern_ev(E)*)	$O' =$ *gen*(τ)
reset_AV(ss, I)	n/a			*assign(ss.AV,I.var)*			
next_AV(ss, τ, AV')	n/a			$AV' =$ *assign*$(ss.AV,$ *eval*$(ss.AV,$ *asn*$(\tau)))$	$AV' =$ *assign*$(ss.AV,$ *eval*$((ss.AV, ss.AVa)$ *asn*$(\tau)))$	$AV' =$ *assign*$(ss.AV,$ *eval*$(ss.AV,$ *asn*$(\tau)))$	$AV' =$ *assign*$(ss.AV,$ *eval*$(ss.AV,$ *last*$($*asn*$(\tau))))$
reset_AVa(ss, I)	n/a			n/a	*assign(ss.AV,I.var)*	n/a	n/a
next_AVa(ss, τ, AVa')	n/a			n/a	$AVa' = ss.AVa$	n/a	n/a
en_cond(ss, τ)	n/a			$ss.AV \models$ *cond*(τ)	$ss.AV, ss.AVa \models$ *cond*(τ)	$ss.AV \models$ *cond*(τ)	
macro_semantics	simple diligent			simple nondiligent	stable	stable	stable
pri(Γ)	no priority			no priority	no priority	no priority	lowest-ranked scope
resolve_conflicts(vv$_1$,vv$_2$,vv)	n/a			n/a	n/a	n/a	*resolve*$_{\mathrm{STM}}$*(vv$_1$, vv$_2$, vv)*

entered(s) : set of states when state s is entered, including s's ancestors and relevant descendants' default states
I.ev, I.var : are events and variables that are inputs of the specifi cation
assign(X,Y) : updates assignments X with assignments Y
eval (X,A) : evaluates expressions in A with respect to assignments X, and returns variable-value assignments for A
last(A) : a sub-sequence of A, comprising only the last assignment to each variable in the sequence of assignments A
resolve$_{\mathrm{STM}}$*(vv$_1$, vv$_2$, vv)* : variable assignment *vv* is chosen nondeterministically from variable assignments *vv*$_1$, and *vv*$_2$
gen(τ), dest(τ), src(τ), trig(τ), cond(τ), and *asn(τ)* are accessor functions on transition τ

- $next\_IE : IE' = gen(\tau)$
  When a transition executes, snapshot element IE is set to the transition's set of generated events.

- $next\_Ia : Ia' = \phi$
  When a transition executes, snapshot element Ia is emptied.

- $next\_O : O' = gen(\tau)$
  When a transition executes, snapshot element O is set to the transition's set of generated events.

- $next\_AV : AV' = assign(ss.AV, eval(ss.AV, last(asn(\tau))))$
  In STATEMATE, a transition can make multiple assignments to the same variable; however, when a transition executes, only the last assignment to the variable has an effect, and the assignment expression are evaluated with respect to the current variable value in AV.

- $en\_states : ss(\tau) \subseteq ss.CS$
  The transition's source state is in the current set of states.

- $en\_events : trig(\tau) \subseteq ss.IE \cup ss.Ia$
  The transition's triggering event(s) is in the set of IE or Ia.

- $en\_cond : ss.AV \models cond(\tau)$
  The transition's condition is satisfied by the variable values in AV.

- $macro\_semantics : stable$
  STATEMATE's step-semantics is stable macro semantics.

- $pri : scope\ outer$
  In STATEMATE, a transition with scope nearer the top of the state hierarchy has priority over a transition farther away from it.

- $resolve\_conflicts : resolve_{STM}(vv_1, vv_2, vv)$
  STATEMATE allows multiple variable assignments to the same variable in a micro-step. When a conflict happens, STATEMATE resolves the conflict by assigning the variable a value chosen nondeterministically from the possible values.

2.2.3 Composition Operators

Composition operators are used to compose a collection of HTSs in a specification. The composition operators control when the component HTSs execute and how the HTSs share data (e.g., generated events). A composition is a binary operation, whose operands are called the left component and the right component, and whose result is a named composite HTS that can be further composed. Composition operators are defined as parameterized, composite micro-step relations that relate pairs of consecutive snapshot collections. Template semantics currently include eight composition operators: parallel, parallel Harel, interleaving, environmental synchronization, rendezvous, sequence, choice, and interrupt. We provide a brief description of these operators here. The meaning of these composition operators will be discussed in greater detail in Chapter 3 where we describe how to translate the composition operators to the SMV input language.

- In **parallel** composition, both components execute transitions in the same micro step if both components have enabled transitions; otherwise only one component executes and the other updates shared variables and events. The case where both components do not execute is not an allowable micro-step.
- **Parallel Harel**, which captures the meaning of parallel composition in Harel's original definition of statecharts [17], differs from parallel composition in that it does not force both components to execute if they are both enabled.
- In **interleaving** composition, only one component can execute transitions in a micro-step.
- In **environmental synchronization**, both components execute in the same micro-step if the executing transitions all have the same trigger event that is a designated synchronization event; otherwise, one or the other component takes a step in isolation, executing transitions not triggered by synchronization events.
- **Rendezvous** is the only composition operator in which events generated in one component are transfered to the other component within the same micro-step. In rendezvous composition, exactly one transition in the sending component generates a rendezvous event that triggers exactly one transition in the receiving component in the same micro-step; otherwise only one component can take a step, executing transitions that are not triggered by rendezvous events.

- In **sequence** composition, the left component executes in isolation until it terminates (i.e., reaches its final basic states), and then the right component executes in isolation. If the left component is a composite component, then all of its basic components must reach final basic states before the right component can start. There are three stages to the behaviour of sequence composition. In the first stage, the left component executes and the shared variables of the right component are updated. In the second stage, the left component reaches its final states, control transfers to the right component. In the third stage, the right component executes and the shared variables of the left component are updated.

- The **choice** composition nondeterministically chooses one component to execute in isolation; once the choice is made, the composite HTS behaves only like the chosen component, and never executes the other component.

- **Interrupt** composition allows control to pass between two components via a provided set of **interrupt transitions**. Interrupt transitions' source and destination states can be either states within a component or a component itself. There are six cases in the behaviour of interrupt composition. In the first case, the left component has enabled transitions, and any enabled interrupt transitions have lower priority than the enabled transitions in the left component. Therefore, the left component executes and the right component is updated. In the second case, one of the interrupt transitions is enabled and has priority over all enabled transitions in the left component, the interrupt transition executes, and the control passes from the left component to the right component. In the third case, the enabled transitions in the left component has the same priority as the enabled interrupt transitions, the interrupt composition nondeterministically choose either a component or an interrupt transition to execute. The final three cases of interrupt composition are symmetric to the first three cases, in that we now consider transitions whose source states are in the right component. Only one component ever has current states, so only one component can have enabled transitions in any snapshot.

Table 2.3 (from [32]) maps various notations' composition operators to the template semantics operators. For example, CCS and CSP's parallel composition ($a \rightarrow P \| a \rightarrow Q$), and basic LOTOS' parallel composition ($P|[a,b,c]|Q$) are template semantics' environmental synchronization composition. RSML, STATEMATE, and statecharts' OR-state composition are template se-

Composition	CCS	CSP	Basic LOTOS	BTS	statecharts [17]	RSML	STATEMATE
Parallel	n/a			n/a	parallel Harel	parallel	
Environmental-sync	$a \rightarrow P \| a \rightarrow Q$		$P \mid [a, b, c] \mid Q$	n/a	n/a	n/a	n/a
Rendezvous-sync	$a.P \mid \bar{a}.Q$	n/a	n/a	n/a	n/a	n/a	n/a
Interleaving	n/a	$P \mid\mid\mid Q$	$P \mid\mid\mid Q$	interleaving (macro)	n/a	n/a	n/a
Sequence	$P; Q$	$P; Q$	$P \gg Q$	concatenation (;)	n/a	n/a	n/a
Choice	$P + Q$	$P[]Q$	$P[]Q$	selection (OR)	n/a	n/a	n/a
Interrupt	n/a	n/a	n/a	n/a	OR-state composition		

Table 2.3: Compositions Operators for Notations

mantics' interrupt composition. RSML and STATEMATE's AND-state composition are mapped to template semantics' parallel composition, while statecharts' AND-state composition is template semantics' parallel Harel composition.

Figure 2.5 (on page 8) shows the composition of the heating system HTSs. In Figure 2.5, each dashed line is a composition, whose operator is named in the line's center circle, and whose two operands are the boxes that lie on either side of the line. HTSs *heatReq* and *noHeatReq* are composed using interrupt composition[1], which has associated with it the interrupt transitions *t19* and *t20* to transfer control between the two HTSs, *noHeatReq* and *heatReq*. The *room*, the *controller*, and the *furnace* execute concurrently via parallel compositions.

2.3 SMV Family of Model Checkers

The SMV family of model checkers includes Cadence SMV [1], a commercial tool designed by Cadence Design Systems Inc., and NuSMV [9], an academic tool developed jointly by ITC-IRST and Carnegie Mellon University. They both originated from SMV [28], and they use the SMV input language.

The SMV family of model checkers are symbolic model checkers. An SMV model describes the steps the system can take. Given a model written in its input language, and a property that describes the expected behaviour of the model, SMV automatically verifies whether the property holds in the model (given sufficient computational resources).

An SMV model can be described using a hierarchy of modules. It must have a `main` MOD-

[1]The component *room* could be represented as a single HTS, but for illustration purposes we treat it as a result of interrupt composition.

ULE, which is the top-level module. A `MODULE` bundles statements together. Modules can be hierarchical. All statements in all modules run *synchronously* in every SMV step. An SMV module is divided into sections using keywords.

Variables are declared in the `VAR` section. SMV provides built-in finite data types, such as boolean, enumerated type, and integer range. A variable can be a built-in data type or an instance of a module.

The `ASSIGN` section describes how variables change in a step. Variables are assigned new values in every SMV step: either a variable `x`'s current value is the current value of an expression; or its next value (`next(x)`) is the current value of an expression and the variable starts with an initial value (`init(x)`). In SMV, if assigned to a boolean variable, constant '1' represents true, and '0' represents false. Expression operators '!,' '&,' '$\rightarrow$,' '$\leftrightarrow$,' and '$|$' represent 'not,' 'and,' 'implies,' 'iff,' and 'or', respectively. SMV allows an assignment to be nondeterministic using curly brackets '$\{\}$'. Comments follow the symbol '--'.

SMV supports macro definitions, declared in the `DEFINE` section. Macros are replaced by their definitions, so they do not increase the system's state space. We sometimes refer to boolean macros as "flags".

In SMV, properties to be checked can be expressed in two different temporal logics: Computation Tree Logic (CTL), and Linear Temporal Logic (LTL)[2]. In NuSMV, they are described using keywords `SPEC` and `LTLSPEC` respectively. When a specification is discovered to be false, NuSMV and Cadence SMV construct and print a counterexample that shows a trace of the specification that falsifies the property. NuSMV's simulation feature offers the user the possibility of exploring the possible executions of an SMV model.

Figure 2.6 shows a small example of SMV model. In this model, there are two variables *x*, and *y*. *x* is a variable of integer range from 0 to 3. It is initialized to 0, and in each subsequent step, its value is increased by 1 unless it is equal to 3, in which case *x* is reset to 0. *y* is a variable of integer range from 0 to 6, it is always equal to $x+1$. *error* is a macro that is true if variable *x* underflows or overflows. There are two CTL properties in the model, the first property is that it's never true that variable *x* will either underflow or overflow; the second property is that in some future state, variable *y* can be greater than 4. Figure 2.7 shows the results returned by NuSMV. It returns true for the first property, and provides a trace to show that the second property does

[2]We assume the readers of this thesis are familiar with CTL and LTL.

```
MODULE main
  VAR
    x: 0..3;
    y : 0..6;
  ASSIGN
    init(x) := 0;
    next(x) := (x+1) mod 4;
    y := x+1;
  DEFINE
    error := (x<0 | x>3);
  SPEC AG !(error)  --true
  SPEC EF (y > 4)   --false
```

Figure 2.6: Variable assignment in SMV

not hold. Since *y* is always equal to $x+1$, and *x* ranges from 0 to 3, the range of *y* is from 1 to 4 although it is declared to be from 0 to 6.

Invariants of the specification can be expressed as boolean expressions following the keyword `INVAR`. As an alternative in addition to the `ASSIGN` section, the transition relation can be given explicitly in terms of variables (e.g., `x`) and their next values (e.g., `next(x)`) after the keyword `TRANS`. Fairness constraints, which declare a condition is true infinitely often, can be specified in a section that begins with the keyword `FAIRNESS`.

In Express, a module is used for two different purposes. The first usage is to reuse statements of a module by creating an instance of the module. The second usage is to group variables or macros together in a module. An instance of this type of module is a record that has a field for each variable or macro declared in the module. For example, if `a` identifies an instance of a module, then the expression `a.b` identifies the internal variable or macro named `b` within module instance `a`. This record can be passed as a parameter to another module, so the whole set of variables or macros can be used by another module. For example, if a record `a` has fields `b` and `c`, then passing record `a` as a parameter to a module allows the module to reference `b` and `c` using `a.b` and `a.c`. Subrecords can also be passed to modules. We use different naming conventions to differentiate these two usages. Module names beginning with "_" denote modules that contain only statements and no variables or macros. Otherwise, the module name is used to group variables or macros together to be passed as a parameter to another module.

```
-- specification AG (!error) is true
-- specification AF y > 4 is false
-- as demonstrated by the following execution sequence
-- loop starts here --
-> State 1.1 <-
    error = 0
    x = 0
    y = 1
-> State 1.2 <-
    x = 1
    y = 2
-> State 1.3 <-
    x = 2
    y = 3
-> State 1.4 <-
    x = 3
    y = 4
-> State 1.5 <-
    x = 0
    y = 1
```

Figure 2.7: Results Returned by NuSMV

2.4 Summary

In this chapter, we briefly described template semantics and the SMV family of model checkers to provide the background needed to understand the translation from a specification in template semantics to an model in SMV, which will be discussed in the next chapter.

Chapter 3

Translation to SMV

In this chapter, we show our method for translating a composed hierarchical transition system into a collection of SMV modules, structured to match the organization of template semantics. The translator takes as input a specification in a notation and the parameter values for the notation's semantics, and produces a model in the SMV language of the specification. We explain our translation method in terms of the SMV model produced for input template parameters and a specification. The implementation of our translator is straightforward and will be described in Chapter 4.

3.1 Architecture of Generated SMV Model

Figure 3.1 shows the module structure of the generated SMV model for a specification. Boxes represent SMV modules. A solid arrow represents the instantiation of the source module to create a record of SMV variables and/or macros, which is passed as a parameter to the destination module. The dashed arrows show sub-modules and encodings of template-parameter values (e.g., the implementation of template parameter *reset_CS* is a sub-module of module `reset`, and the implementation of template parameter *pri* is encoded in the definition of module `enabled`). Entities in the figure that share names with template parameters or composition operators correspond to those elements. Next, we describe the details of Figure 3.1.

Figure 3.1 describes a micro-step from a snapshot (`pss`) to its value at the end of a step (`pss'`). A macro-step is a sequence of zero or more micro-steps. At the start of each macro-step,

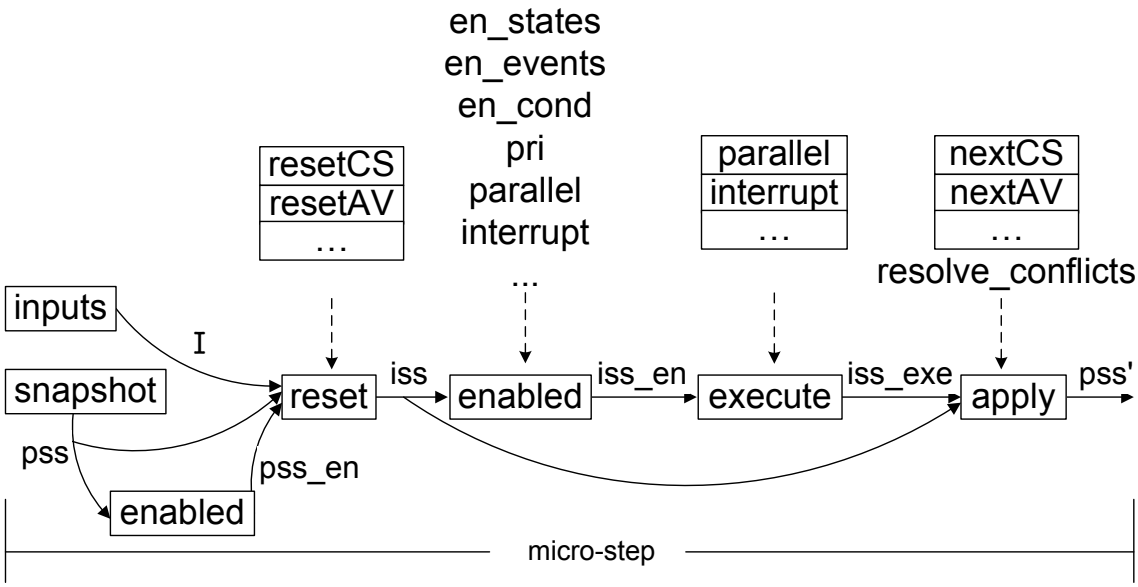


Figure 3.1: Architecture of SMV Model

the snapshot elements have to be reset, We handle the start of each macro-step via a conditional reset in a micro-step. Record `pss` is the current snapshot of a specification. It is an instantiation of module `snapshot`, and contains SMV variables that represent the specification's states, variables, and events. Record `I` is the current input, and contains SMV variables that represent environment variables and events. `pss'` is the value of `pss` in the next step, generated by `next` statements in the module `apply`. A step in SMV corresponds to a micro-step in the original specification.

Module `reset` realizes template-semantics definition *reset*, defining record `iss` in terms of snapshot `pss`, modified to incorporate inputs `I` if the system is stable. For each provided template parameter *reset_XX*[1], the module has a sub-module `resetXX` that sets the value of the snapshot element *XX*, according to the semantics of the provided parameter's value. Record `iss` has the structure of a `pss` record, but it consists of macros rather than variables, so it does not add to the model's state space.

To conquer the complexity of the semantics of model-based notations, following the decomposition of template semantics, we separate the concepts of what parts of the system are enabled

[1] XX represents a snapshot element, e.g., *reset_CS* is the template parameter that resets the snapshot element CS.

from what parts of the system execute. We use boolean macros in the enabled and execute modules to capture and communicate this information throughout the SMV model.

Module `enabled` indicates whether entities are enabled in some snapshot. It realizes the template-semantics definitions of *enabled_trans* and *pri*: it takes a snapshot as a parameter, and identifies the transitions, HTSs, and composed HTSs enabled in that snapshot. The module uses template parameters *en_states*, *en_events*, and *en_cond* to test whether snapshot states, events, and variable values enable transitions. It uses template parameter *pri* to select the transitions of highest priority for possible execution. The module uses the semantics of the composition operators to decide whether a composed HTS is enabled based on the enabled status of its components. Record `iss_en` is the result of instantiating module `enabled` with snapshot `iss`, this record identifies entities that are enabled in the reset snapshot `iss`. Record `pss_en` is the result of instantiating module `enabled` with the current snapshot `pss`. `pss_en` is used to test whether `pss` is a stable snapshot (in which no transition is enabled). If a notation's macro-semantics are simple rather than stable, `pss_en` can be omitted. Records `iss_en` and `pss_en` contain only macro definitions, so they do not add to the model's state space.

Module `execute` realizes the meaning of the specification's composition operators by constraining the subset of enabled entities that execute in a step. Record `iss_exe` is the result of instantiating module `execute` with information from `iss_en` about enabled entities. `iss_exe` contains macros, and one variable of enumerated type for each HTS whose value indicates which transition (if any) of the HTS is taken in a step.

Module `apply` realizes template-semantics definition *apply*. It uses information in `iss_exe` about which entities execute, and applies the executing transitions' actions to the reset snapshot `iss`, producing the next snapshot `pss'`. For each provided template parameter *next_XX*, the module has a sub-module `nextXX` that updates snapshot element *XX* to reflect the actions of the executing transitions.

In addition (and not shown), there is an SMV module `initss` that contains statements that initialize the snapshot elements in `pss`.

All of the above modules are declared in the SMV `main` module. Figure 3.2 shows the `main` module for the heating system with STATEMATE semantics, which use stable macro-semantics. Module names beginning with "_" denote modules that contain only statements and no variables or macros. Module `snapshot`, which declares variables for states, events, and variables of a

specification, is instantiated as a record called `pss`. Record `pss` is passed as an argument to module `initss` to get initialized. Module `inputs` is instantiated as a record called `I`, which declares events and variables from the environment. Module `enabled`, which takes `pss` as an argument, is instantiated as a record called `pss_en` that determines whether snapshot `pss` is a stable snapshot. The macro `stable`, which keeps track of whether the specification reaches a stable snapshot, is a macro declared in the record `pss_en` (`pss_en.stable`). Module `reset`, taking records `pss`, `I`, and macro `pss_en.stable` as arguments, is instantiated as a record called `iss`, which is the reset snapshot. Module `enabled` is instantiated again with argument `iss`, and results in a record called `iss_en`, which identifies the macros of enabled entities in the snapshot `iss`. Module `execute`, taking `iss_en` as an argument, is instantiated as a record called `iss_exe` that determines the macros of executing entities in snapshot `iss`. Module `apply` updates snapshot element in `pss` according to the executing transitions determined in `iss_exe`.

```
MODULE main
  VAR
    -- pss contains variables storing snapshot elements
    pss : snapshot;
    -- initss is a module with "init" statements
    _initss : initss(pss);
    -- I is a record of inputs
    I : inputs;
    -- pss_en contains macros identifying enabled entities in
    pss_en: enabled (pss);
    -- iss contains macros of type "snapshot"
    iss: reset(pss_en.stable, pss, I);
    -- iss_en contains macros identifying enabled entities in
    iss_en: enabled (iss);
    -- iss_exe contains macros identifying executing entities
    iss_exe: execute (iss_en);
    -- apply is a module with "next" statements
    _apply : apply (pss, iss, iss_exe);
```

Figure 3.2: Main Module for Stable Macro-semantics

Figure 3.3 shows the `main` module for a system of simple macro-semantics, it is similar to the `main` module for a system that uses stable macro-semantics, except that, the model does

not check whether the system is stable, so there is no need to calculate whether snapshot `pss` is enabled after the execution of a micro-step; therefore, no `pss_en` variable is declared, and `reset` does not depend on the macro `pss_en.stable` to decide whether the system should read inputs. The difference between diligent and nondiligent simple macro-semantics is captured in the `execute` module (Section 3.7).

```
MODULE main
  VAR
    -- pss contains variables storing snapshot elements
    pss : snapshot;
    -- initss is a module with "init" statements
    _initss : initss(pss);
    -- I is a record of inputs
    I : inputs;
    -- iss contains macros of type "snapshot"
    iss: reset(pss, I);
    -- iss_en contains macros identifying enabled entities in
    iss_en: enabled (iss);
    -- iss_exe contains macros identifying executing entities
    iss_exe: execute (iss_en);
    -- apply is a module with "next" statements
    _apply : apply (pss, iss, iss_exe);
```

Figure 3.3: Main Module for Simple Macro-semantics

In the following sections, we provide more details for the modules `snapshot, inputs, initss, reset, enabled, execute,` and `apply`.

3.2 Snapshot Module

Module `snapshot` represents a specification's snapshot. A snapshot is an 8-tuple:

$$\langle CS, IE, AV, O, CSa, IEa, AVa, Ia \rangle.$$

Because not all elements are used in all notations, the module `snapshot` only declares a submodule for each snapshot element that is used in a notation's semantics. For example, in STATEMATE, only the snapshot elements CS, IE, AV, Ia and O are used. Figure 3.4 shows the `snapshot` module for any specification written in STATEMATE statecharts, where the data

types `states`, `variables`, `intEvents`, `IaEvents`, and `outputs` are sub-modules that group the specification's states, variables, internal events, environment events, and output events, respectively. Snapshot elements of the same type have the same sub-module type. Thus in a notation using snapshot elements CSa, CS and CSa could both be instances of the module `states`.

Figure 3.5 shows the `snapshot` module for any specification written in statecharts, which uses the snapshot elements CS, IE, AV, CSa, AVa, Ia and O.

```
MODULE snapshot
  VAR
    CS  : states;
    AV  : variables;
    IE  : intEvents;
    Ia  : IaEvents;
    O   : outputs;
```

Figure 3.4: Snapshot Module (STATEMATE)

```
MODULE snapshot
  VAR
    CS  : states;
    AV  : variables;
    IE  : intEvents;
    O   : outputs;
    CSa : states;
    AVa : variables;
    Ia  : IaEvents;
```

Figure 3.5: Snapshot Module (Statecharts)

```
MODULE states
 VAR
  furnace_state: {furnaceRun,furnaceOff,furnaceAct,furnaceErr,noState};
  ...
 DEFINE
  in_furnaceOff := furnace_state=furnaceOff;
  in_furnaceNormal := in_furnaceOff|in_furnaceAct|in_furnaceRun;
  ...
```

Figure 3.6: Snapshot Sub-module for CS (Heating System)

Figure 3.6 shows part of the sub-module `states` for the heating system[2]. The sub-module contains for each HTS (e.g., *furnace*) an enumerated variable (`furnace_state`), whose type has a value for each of the HTS's basic states, plus a value `noState` to indicate that the HTS is

[2] Specification can be found on page 8.

not active. The sub-module also defines for every state a boolean macro (e.g., `in_furnaceOff`, `in_furnaceNormal`) that indicates whether that state is a current state in the snapshot; a super-state is current if one of its child states is current. Thus, we represent the specification's state hierarchy using variables only for the basic states. This representation uses $\lceil log_2(n+1) \rceil$ space for an HTS with n basic states.

Currently, in our implementation, we assume that CS and CSa always represent a set of basic states. However, the representation of CSa depends on the type of information stored. If CSa is used to record all previous states visited in the current macro-step, to avoid an infinite macro-step, then a boolean variable is needed for every basic state and super-state, and the set is represented as the characteristic predicate over these variables.

```
MODULE variables
  VAR
    valvePos : 0..2;
    actualTemp : 10..30;
    requestHeat : boolean;
    furnaceStartupTime : 0..5;
    warmUpTime : 0..5;
    coolDownTime : 0..5;
    setTemp : 16..24;
    error_variables : boolean;
```

Figure 3.7: Snapshot Sub-module for AV (Heating System)

Sub-module `variables` contains an SMV variable of appropriate type for each of the specification's data variables. Our translator currently supports variables of types boolean, enumerated, and integer range. Figure 3.7 shows the `variables` sub-module for the heating system. Variables `actualTemp`, and `requestHeat` have types of integer range, and boolean, respectively. The last variable `error_variables` is an extra boolean variable to catch variable underflow and overflow errors; its use will be explained in Section 3.8.

```
MODULE intEvents
  VAR
    activate : boolean;
    deactivate : boolean;
    furnaceReset : boolean;
    furnaceRunning : boolean;
```

Figure 3.8: Snapshot Sub-module for IE (Heating System)

Sub-module `intEvents` contains a boolean variable for each of the specification's internal events to indicate whether the event is occurring in the current snapshot. Figure 3.8 shows the declarations of the `intEvents` module for the heating system.

```
MODULE IaEvents
  VAR
    heatSwitchOn : boolean;
    heatSwitchOff : boolean;
    userReset : boolean;
    furnaceFault : boolean;
```

Figure 3.9: Snapshot Sub-module for Ia (Heating System)

Sub-module `IaEvents` contains a boolean variable for each of the specification's environment events that are needed for snapshot element Ia, to indicate whether the event is occurring in the current snapshot. Figure 3.9 shows the declarations of the `IaEvents` module for the heating system.

```
MODULE outputs
  VAR
    activate : boolean;
    deactivate : boolean;
    furnaceReset : boolean;
    furnaceRunning : boolean;
```

Figure 3.10: Snapshot Sub-module for O (Heating System)

Sub-module `outputs` declares a boolean variable for each of the events that can be sensed by the environment. In STATEMATE, the output events are the set of internal events of the system. Figure 3.10 shows the `outputs` module for the heating system. The `outputs` module

has the same set of events as defined in `intEvents` module, since the definition of internal events and output events are the same in STATEMATE [3].

3.3 Inputs Module

```
MODULE inputs
  VAR
    ev : envEvents;
    var : envVars;
```

Figure 3.11: Input Module for I

Module `inputs` declares the events and variables that are inputs of the specification. Figure 3.11 shows the `inputs` module, which has the field `ev` that instantiates the `envEvents` sub-module to represent input events, and the field `var` that instantiates the `envVars` sub-module to represent variables from the environment.

```
MODULE envEvents
  VAR
    heatSwitchOn : boolean;
    heatSwitchOff : boolean;
    userReset : boolean;
    furnaceFault : boolean;
```

Figure 3.12: Input Sub-module for $I.ev$ (Heating System)

```
MODULE envVars
  VAR
    setTemp : 16..24;
    actualTemp : 10..30;
```

Figure 3.13: Input Sub-module for $I.var$ (Heating System)

[3] In the case of STATEMATE, snapshot element O duplicates the information in snapshot element IE. A future optimization would be to recognize this duplication and reduce the state space of the system.

Figure 3.12 shows the sub-module `envEvents` for the heating system, which contains a boolean variable for each input event [4]. Figure 3.13 shows the sub-module `envVars` for the heating system. This sub-module contains a variable for each of the environment variables, which can have type enumerated, integer range, or boolean, declared in the specification. In the heating system, the user sets the desired temperature using the environment variable `setTemp`, and the actual temperature, `actualTemp`, is sensed by the sensor from the environment.

3.4 Initss Module

```
MODULE initss(pss)
  ASSIGN
    init(pss.CS.furnace_state) := furnaceOff;
    ...
    init(pss.IE.activate) := 0;
    ...
    init(pss.Ia.heatSwitchOn) := 0;
    ...
    init(pss.AV.valvePos) := {closed};
    ...
    init(pss.AV.error_variables) := 0;
```

Figure 3.14: Initss Module (Heating System)

Module `initss` initializes all snapshot elements of the system. Figure 3.14 shows part of the initialization of the heating system, which has snapshot elements CS, AV, IE, Ia and O. The state variable for an HTS is initialized to the default basic-state descendant of the HTS's root state. All event variables are initialized to 0 since there are no generated events when the system starts. A specification provides the initial values of all variables, so the variables are initialized according to the system's specification. In template semantics, there can be multiple possible initial values for a variable, so we use the nondeterministic assignment ($\{\}$) of SMV to capture

[4] In most notations, sub-modules `envEvents` and `IaEvents` are the same, except in CCS and CSP, where generated events belong to snapshot element Ia, thus `IaEvents` has both input events and generated events.

these options. The last statement initializes the boolean variable that catches variable underflow and overflow errors, which is set to 0 to indicate no errors exist when the system starts.

3.5 Reset Module

The template-semantics function, *reset*, is applied to the snapshot at the start of a macro-step, to read new inputs from the environment and to remove data about the previous macro-step. It is defined in terms of template parameters *reset_XX*, which specify how each snapshot element *XX* is reset. SMV module `reset` sets each snapshot element *XX* in a sub-module `resetXX`, which realizes the semantics of the provided template-parameter value for *reset_XX*. Because an SMV step corresponds to a micro-step, for notations with simple macro-step semantics, each micro-step is a macro-step, and the snapshot is reset in each SMV step. For notations with stable macro-step semantics, whether an SMV step needs to include a reset depends on whether the snapshot is a stable snapshot. We simulate macro-steps by including a conditional reset within every SMV step, adapting the work of Chan et al. [8]. The module `enabled` includes a macro `stable` that indicates when the system is stable (i.e., when a new macro-step is starting); module `reset` takes `stable` as a parameter and changes snapshot elements only when `stable` is true.

```
MODULE reset(ss,I)
  VAR
    CS : resetCS(ss,I);
    Ia : resetIa(ss,I);
    O  : resetO(ss,I);
```

Figure 3.15: Reset Module for CCS

```
MODULE reset(stable,ss,I)
  VAR
    CS : resetCS(stable,ss,I);
    AV : resetAV(stable,ss,I);
    IE : resetIE(stable,ss,I);
    Ia : resetIa(stable,ss,I);
    O  : resetO(stable,ss,I);
```

Figure 3.16: Reset Module for STATEMATE

Figure 3.15 and Figure 3.16 show the `reset` modules for CCS, and STATEMATE, respectively. They differ in that, first, the modules only contain the sub-modules to represent the applicable snapshot elements; second, STATEMATE uses stable macro-semantics, so the `reset` module and its sub-modules for STATEMATE depend on the value of the macro `stable`, and CCS uses simple macro-semantics, so the modules do not depend on whether the snapshot is stable.

```
MODULE resetXX(stable,ss,I)
  --stable macro-semantics
  DEFINE
     x := case
       stable : ...; --reset x with I
       1 : ss.XX.x;  --not changed
     esac
```

```
MODULE resetXX(ss,I)
   --simple macro-semantics
   DEFINE
     x:= ...; --reset x with I
```

Figure 3.17: ResetXX Sub-module

Figure 3.17 shows an outline of the form of the `resetXX` modules for stable macro-semantics and simple macro-semantics. In stable macro-semantics, snapshot element *x* is reset when the system is stable, otherwise, it keeps its value from snapshot `ss`. In simple macro-semantics, *x* is reset in each micro-step.

Next, we explain how to map *reset* parameter values to SMV, using stable macro-semantics. The mapping of parameter values using simple macro-semantics is similar. Table 3.1 shows the `reset` parameter functions and their possible values (from Table 2.2) that are supported by our translator; the last column shows the figure number that describes the corresponding SMV modules. All examples are for the heating system specification.

Sub-module `resetCS` implements the function *reset_CS(ss,I)*. For all supported notations, reset does not change the value of CS. Figure 3.18 shows part of the SMV sub-module for `resetCS` in stable macro-semantics for the HTS *furnace*. This `case` statement is used to differentiate the affect of the starting of a macro-step with the affect of a micro-step [5]. The macros for other HTSs are defined similarly.

[5]This statement can be compressed to `furnace_state := ss.CS.furnace_state`. A future optimization would be to compress this type of statement, which could result in a simpler BDD transition relation.

Parameter Name	Parameter Value	Notations	SMV Module
$reset\_CS$	$ss.CS$	CCS, CSP, LOTOS, BTS, statecharts, RSML, STATEMATE	Figure 3.18
$reset\_CSa$	n/a	CCS, CSP, LOTOS, RSML, STATEMATE	n/a
	$ss.CS$	statecharts	Figure 3.19
$reset\_IE$	n/a	CCS, CSP, LOTOS, BTS	n/a
	ϕ	statecharts, RSML, STATEMATE	Figure 3.20
$reset\_IEa$	n/a	CCS, CSP, LOTOS, BTS, statecharts, RSML, STATEMATE	n/a
$reset\_O$	n/a	LOTOS, BTS	
	ϕ	CCS, CSP, statecharts, RSML, STATEMATE	(not shown)
$reset\_AV$	n/a	CCS, CSP, LOTOS	n/a
	$assign(ss.AV, I.var)$	BTS, statecharts, RSML, STATEMATE	Figure 3.21
$reset\_AVa$	n/a	CCS, CSP, LOTOS, BTS, RSML, STATEMATE	n/a
	$assign(ss.AV, I.var)$	statecharts	Figure 3.22
$reset\_Ia$	n/a	BTS	n/a
	$I.ev$	CCS, CSP, LOTOS, statecharts, RSML, STATEMATE	Figure 3.23
	$Ia \cup I.ev$	(pre-defined value)	Figure 3.24

Table 3.1: Predicates $reset\_XX(ss, I)$
(n/a means not applicable)

```
MODULE resetCS(stable,ss,I)
  DEFINE
    furnace_state := case
      stable : ss.CS.furnace_state;
      1 : ss.CS.furnace_state;
    esac;
    ...
```

Figure 3.18: ResetCS Sub-module : $ss.CS$ (Heating System)

```
MODULE resetCSa(stable,ss,I)
  DEFINE
    furnace_state := case
      stable : ss.CS.furnace_state;
      1 : ss.CSa.furnace_state;
    esac;
    ...
```

Figure 3.19: ResetCSa Sub-module : $ss.CS$ (Heating System)

Sub-module `resetCSa` implements the function *reset_CSa(ss,I)*. The parameter value $ss.CS$ for *reset_CSa* indicates that at the start of each macro-step, the system copies the contents of snapshot element CS from the beginning of the step into auxiliary element CSa. This parameter value is used in statecharts semantics. No other notation we have described using template semantics use snapshot element CSa. Figure 3.19 shows how to represent the semantics of *reset_CSa* parameter value $ss.CS$ in the sub-module of `resetCSa` for stable macro-semantics.

```
MODULE resetIE(stable,ss,I)
  DEFINE
    activate := case
      stable : 0;
      1 : ss.IE.activate;
    esac;
    ...
```

Figure 3.20: ResetIE Sub-module : ϕ (Heating System)

Sub-module `resetIE` implements the function *reset_IE(ss,I)*, which specifies how to reset the internal events at the start of each macro-step. Figure 3.20 shows part of the `resetIE` sub-

module using the semantics of *reset_IE* parameter value ϕ. When the system is stable, an internal event (e.g., *activate*) is reset to 0 to indicate that it is no longer active; otherwise, it keeps the value from the previous snapshot. Sub-module `reset0` for parameter value ϕ is similar to this module.

Sub-module `resetIEa` implements the function *reset_IEa(ss,I)*. Currently, no supported notations uses snapshot element IEa, so we have not implemented any parameter values for this snapshot element.

```
MODULE resetAV(stable,ss,I)
  DEFINE
    valvePos := ss.AV.valvePos;
    setTemp := case
      stable : I.var.setTemp;
      1 : ss.AV.setTemp;
    esac;
    ...
```

Figure 3.21: ResetAV Sub-module : $ss.AV$ (Heating System)

Sub-module `resetAV` implements the function *reset_AV(ss,I)*, which specifies how to reset the variables at the start of each macro-step. Parameter value $assign(ss.AV, I.var)$ indicates that at the start of each macro-step, the environment variables read values from `I`'s `var` field, and other variables keep the same values. Figure 3.21 shows part of the `resetAV` sub-module for the heating system, using this parameter value. The variable *setTemp*, which is an environment variable, reads from `I.var.setTemp`. The variable *valvePos*, as an internal variable, keeps its value from the previous snapshot.

```
MODULE resetAVa(stable,ss,I)
  DEFINE
    valvePos := case
      stable : ss.AV.valvePos;
      1 : ss.AVa.valvePos;
    esac;
    setTemp := case
      stable : I.var.setTemp;
      1 : ss.AVa.setTemp;
    esac;
    ...
```

Figure 3.22: ResetAVa Sub-module : $ss.AVa$ (Heating System)

Sub-module `resetAVa` implements the function *reset_AVa(ss,I)*, which specifies how to reset the auxiliary variables at the start of each macro-step. Parameter value $assign(ss.AV, I.var)$ is used in statecharts. At the start of each macro-step, the environment variables of AVa read values from `I`'s `var` field, and other variables of AVa read the value from the previous snapshot element AV; otherwise, the variables keep their values from the previous snapshot. Figure 3.22 shows `resetAVa` sub-module using this semantics, that is, if the system is stable, the variable *setTemp*, as an environment variable, reads from `I.var.setTemp`. The variable *valvePos*, as an internal variable, reads the value from the snapshot element AV.

```
MODULE resetIa(stable,ss,I)
  DEFINE
    heatSwitchOn := case
      stable : I.ev.heatSwitchOn;
      1 : ss.Ia.heatSwitchOn;
    esac;
    ...
```

Figure 3.23: ResetIa Sub-module : $I.ev$ (Heating System)

```
MODULE resetIa(stable,ss,I)
  DEFINE
    heatSwitchOn := case
      ss.Ia.heatSwitchOn : 1;
      stable : I.ev.heatSwitchOn;
      1 : 0;
    esac;
    ...
```

Figure 3.24: ResetIa Sub-module : $Ia \cup I.ev$ (Heating System)

Sub-module `resetIa` implements the function *reset_Ia(ss,I)*, which specifies how to read inputs from the environment events at the start of each macro-step. Figure 3.23 shows part of the `resetIa` sub-module for the heating system using the parameter value $I.ev$. When the system is stable, a macro for an external event (e.g., `heatSwitchOn`) reads from `I.ev.heatSwitchOn`[6]; otherwise, it has the same value as in `ss`. Figure 3.24 shows the `resetIa` sub-module using the parameter value $Ia \cup I.ev$, that is, if the event (e.g., `heatSwitchOn`) is in snapshot `ss`, the corresponding macro maintains the event, otherwise, if the system is stable, it reads from `I.ev.heatSwitchOn`, otherwise it keeps the value from the previous snapshot, which is 0.

3.6 Enabled Module

Module `enabled` creates a record of macros that indicate whether transitions, HTSs, and composed HTSs are enabled in snapshot `ss`. Figure 3.25 shows these enabled macros (in shadow boxes) and their dependencies. Each transition `t` has five macros: three macros (e.g., `enStates_t, enEvents_t, and enCond_t`) test the three transition enabling conditions, based on the provided parameter values; one macro (e.g., `ent`) combines these tests to determine whether the transition is enabled, realizing the template definition *enabled_trans*; and one macro (e.g., `t`) tests whether the transition is **priority enabled**, which means it is among the enabled transitions with the highest priority using the definition of template predicate *pri*. An HTS `hts`

[6] In Cadence SMV, this statement can be simplified to `stable : 0,1`, which relies on Cadence SMV to nondeterministically choose whether the event happens, and the boolean variable for `I.ev.heatSwitchOn` is not needed; However, in NuSMV, a macro cannot be assigned a value nondeterministically. Therefore, in order to use both tools, we include the variables in `I` even though this increases the size of the state space.

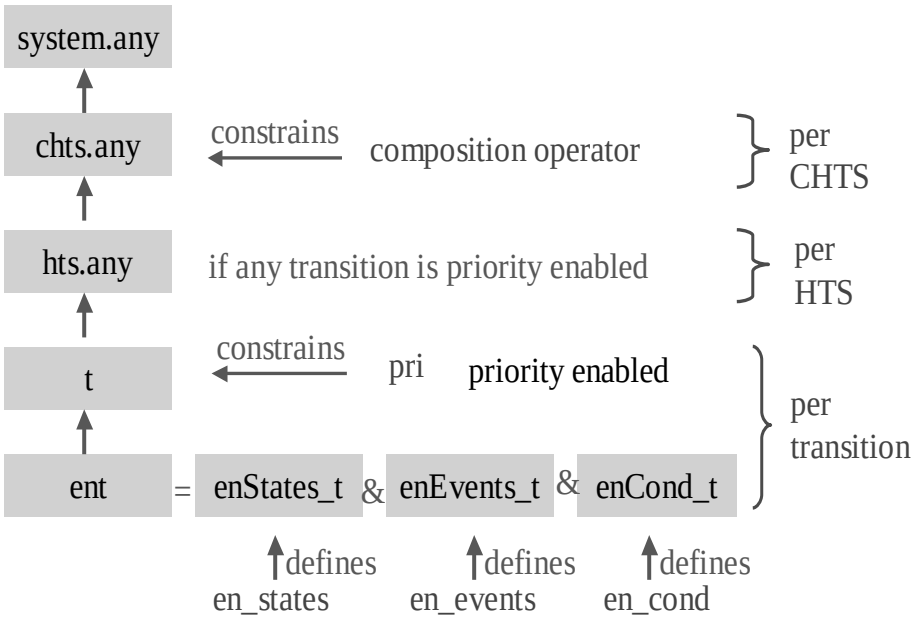


Figure 3.25: Dependency of Enabled Macros

is enabled (e.g., macro `hts.any`) if any of the transitions is priority enabled. Whether a composed HTS `chts` is enabled (e.g., macro `chts.any`) depends on the enabled macros of its two components, and the semantics of the composition operator. Finally, the top level composed HTS is enabled means that the system is enabled (e.g., macro `system.any`), which is used to force diligence of the system if required.

```
MODULE enabled(ss)
 VAR
   noHeatReq:  enabled_noHeatReq(ss);
   heatReq:    enabled_heatReq(ss);
   controller: enabled_controller(ss);
   furnace:    enabled_furnace(ss);
   roomIntrTrans:  enabled_roomIntrTrans(ss); -- interrupt trans
 DEFINE
   room.any := noHeatReq.any | heatReq.any | roomIntrTrans.any;
   house.any := room.any | controller.any;
   heatingSys.any := house.any | furnace.any;
```

Figure 3.26: Enabled Module (Heating System)

Figure 3.26 shows module `enabled` for the heating system. The macros associated with each HTS are set in separate sub-modules. Each HTS `h` has a macro named `h.any` that indicates whether the HTS is enabled, based on whether any of the HTS's transitions are enabled. Each composed component `c`, created by a composition operator, has a macro `c.any` that indicates whether `c` is enabled, based on its sub-components' `h1.any and h2.any` macros, and the semantics of the composition operator. For a specification with interrupt composition, there is a sub-module (e.g., `enabled_roomIntrTrans`) that determines whether the interrupt transitions are enabled. This sub-module is similar to the enabled sub-module of an HTS.

```
MODULE enabled_furnace(ss)
  DEFINE
    ...
    enStates_t3 := ss.CS.in_furnaceAct;
    enEvents_t3 := 1;  -- no trigger event
    enCond_t3 := ss.AV.furnaceTimer = ss.AV.furnaceStartup;
    ent3 := enStates_t3 & enEvents_t3 & enCond_t3;
    ...
    enStates_t7 := ss.CS.in_furnaceNormal;
    enEvents_t7 := ss.Ia.furnaceFault;
    enCond_t7 := 1;    -- no guard condition
    ent7 :=enStates_t7 & enEvents_t7 & enCond_t7;
    ...
    t3 := ent3 & !ent6 & !ent7;
    ...
    t6 := ent6;
    t7 := ent7;
    any := t1|t2|t3|t4|t5|t6|t7;
```

Figure 3.27: Enabled Sub-module (Furnace HTS)

Figure 3.27 shows part of enabled sub-module (`enabled_furnace`) for HTS *furnace*. Each HTS enabled sub-module defines five macros for each of the HTS's transitions as explained in Figure 3.25. Next, we explain how the STATEMATE parameter values are used to create Figure 3.27, along with other parameter values. Later, we will explain how the composition operators affect the enabling of components.

Parameter Name	Parameter Value	Notations	SMV code
en_states	$src(\tau) \subseteq ss.CS$	CCS, CSP, LOTOS, BTS, RSML, STATEMATE	Figure 3.29
	$src(\tau) \subseteq ss.CSa$	statecharts	Figure 3.30
en_events	$trig(\tau) \subseteq ss.Ia$	CCS, CSP, LOTOS, BTS, RSML, STATEMATE	Figure 3.31
	n/a	BTS	n/a
	$trig(\tau) \subseteq ss.IE \cup ss.Ia$	statecharts, RSML, STATEMATE	Figure 3.32
en_cond	n/a	CCS, CSP, LOTOS	n/a
	$ss.AV \models cond(\tau)$	BTS, RSML, STATEMATE	Figure 3.33
	$ss.AV, ss.AVa \models cond(\tau)$	statecharts	Figure 3.30

Table 3.2: Predicate $enabled\_trans$

Table 3.2 shows the three transition-enabling predicates and their values. In the following, we show how to translate the different values of these parameters using the simple example in Figure 3.28. This example has two transitions *t1* and *t2*, one environment event *a*, one internal event *b*, and two variables *x* and *y*. The transitions have no actions. In transition *t2*, the syntax `cur(y)` is used in statecharts to refer to the current value of variable y; if used in other notations, `cur` is ignored, and the expression refers simply to y. [7]

[7] `cur` is valid only when the parameter value of template parameter *en_cond* uses AVa. In the notations supported by Express, only statecharts uses AVa.

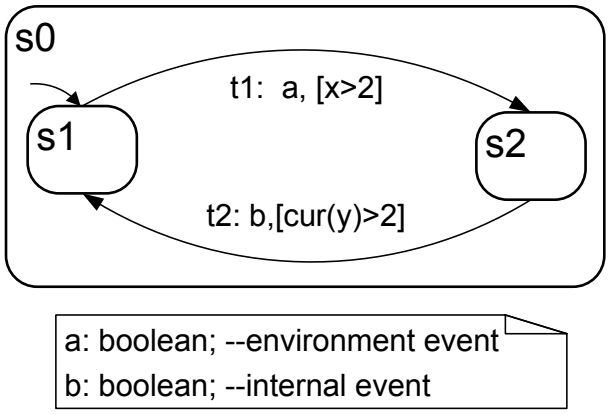


Figure 3.28: Example to Illustrate *enabled_trans*

```
DEFINE
  enState_t1 := ss.CS.in_s1;
  enState_t2 := ss.CS.in_s2;
```

Figure 3.29: en_states : $src(\tau) \subseteq ss.CS$ for Figure 3.28

```
DEFINE
  enState_t1 := ss.CSa.in_s1;
  enState_t2 := ss.CSa.in_s2;
```

Figure 3.30: en_states : $src(\tau) \subseteq ss.CSa$ for Figure 3.28

There are two possible parameter values for template predicate *en_states*. Figure 3.29 shows the state condition of *t1* and *t2* for parameter value $src(\tau) \subseteq ss.CS$ for the example in Figure 3.28. The source state of each transition must be active in the current snapshot element CS for the transition to be enabled. Figure 3.30 shows the state condition of *t1* and *t2* for parameter value $src(\tau) \subseteq ss.CSa$ for the example in Figure 3.28. The source state of each transition must be active in the auxiliary snapshot element CSa for the transition to be enabled.

```
DEFINE
  enEvents_t1 := ss.Ia.a;
  enEvents_t2 := 0;
```

Figure 3.31: en_events : $trig(\tau) \subseteq ss.Ia$ for Figure 3.28

```
DEFINE
  enEvents_t1 := ss.Ia.a;
  enEvents_t2 := ss.IE.b;
```

Figure 3.32: en_events : $trig(\tau) \subseteq ss.IE \cup ss.Ia$ for Figure 3.28

Predicate *en_events* has two possible parameter values. Figure 3.31 shows the mapping of value $trig(\tau) \subseteq ss.Ia$ for the example in Figure 3.28, which indicates that a transition's triggering event must be in the auxiliary snapshot element Ia to enabled the transition. Transition *t1*'s

triggering event *a* is from snapshot element Ia. Transition *t2* is not a useful transition since it is triggered by internal event *b*, which cannot belong to Ia, thus, `enEvent_t2` is always 0, and the transition would never be taken with this *en_events* parameter value. Figure 3.32 shows the mapping of value $trig(\tau) \subseteq ss.IE \cup ss.Ia$ for the example in Figure 3.28, which indicates that a transition's triggering event exists either in the auxiliary snapshot element Ia, or the current snapshot element IE, depending on whether the triggering event is an external or internal event. Transition *t1*'s triggering event *a* is from snapshot element Ia because it is an environment event. Transition *t2*'s triggering event *b* is from snapshot element IE because it is an internal event.

```
DEFINE
  enCond_t1 := ((ss.AV.x)>2);
  enCond_t2 := ((ss.AV.y)>2);
```

Figure 3.33: en_cond : $ss.AV \models cond(\tau)$ for Figure 3.28

```
DEFINE
  enCond_t1 := ((ss.AVa.x)>2);
  enCond_t2 := ((ss.AV.y)>2);
```

Figure 3.34: en_cond : $ss.AV, ss.AVa \models cond(\tau)$ for Figure 3.28

Predicate *en_cond* has two possible parameter values. Figure 3.33 shows the mapping of parameter value $ss.AV \models cond(\tau)$ for the example in Figure 3.28, which indicates that a transition's guard condition is evaluated with respect to the current variable values in AV. Figure 3.33 shows the mapping of parameter value $ss.AV, ss.AVa \models cond(\tau)$ for the example in Figure 3.28, which means, by default, the guarding condition is evaluated with respect to the auxiliary variable values in AVa, but if the guarding condition uses the current snapshot element (e.g., syntax `cur(y)` in transition *t2*), it is evaluated with respect to the current variable values in AV. In this mapping, transition *t1*'s guarding condition is evaluated according to AVa, and transition *t2*'s guarding condition is evaluated according to AV.

A transition is enabled when its three transition-enabling conditions are true. In template semantics, only enabled transitions of the highest priority can be chosen to execute, therefore, based on the enabled macros for all transitions in an HTS, we need to know which enabled transitions are priority enabled (i.e., really have a chance to execute), which is defined by template parameter *pri*.

Parameter Value	Notations	SMV Module
noPri	CCS, CSP, LOTOS, statecharts, RSML	Figure 3.35
scope outer	STATEMATE	Figure 3.36
scope inner	(pre-defined value)	Figure 3.37
explicit	(pre-defined value)	(not shown)

Table 3.3: Predicate $pri(\Gamma)$

```
MODULE enabled_furnace(ss)
  DEFINE
    ...
    t1 := ent1;
    t2 := ent2;
    t3 := ent3;
    t4 := ent4;
    t5 := ent5;
    t6 := ent6;
    t7 := ent7;
```

Figure 3.35: pri : $noPri$ (Furnace HTS)

Table 3.3 shows all the possible parameter values of predicate ***pri***. Parameter value $noPri$ indicates that all transitions in a system have the same priority. Figure 3.35 shows the definition of priority-enabled macros for all transition in HTS ***furnace***, for ***pri*** parameter value $noPri$. Since all transitions have the same priority, any transition, if enabled, is priority enabled.

```
MODULE enabled_furnace(ss)
  DEFINE
    ...
    t1 := ent1 & !ent6 & !ent7;
    t2 := ent2 & !ent6 & !ent7;
    t3 := ent3 & !ent6 & !ent7;
    t4 := ent4 & !ent6 & !ent7;
    t5 := ent5 & !ent6 & !ent7;
    t6 := ent6;
    t7 := ent7;
```

Figure 3.36: pri : $scope\ outer$ (Furnace HTS)

Figure 3.36 shows the definition of priority-enabled macros using parameter value $scope\ outer$, where transitions *t7* and *t6* have higher priority than *t1, t2, t3, t4, and t5*, because their scopes[8] are nearer the top of the state hierarchy. Therefore, *t1* is priority enabled only if it is enabled and *t7* and *t6* are not enabled. *t6* and *t7* are priority enabled if they are enabled. This priority scheme is used in Figure 3.27 for the *furnace* HTS.

```
MODULE enabled_furnace(ss)
  DEFINE
    ...
    t1 := ent1;
    t2 := ent2;
    t3 := ent3;
    t4 := ent4;
    t5 := ent5;
    t6 := ent6 & !ent1 & !ent2 & !ent3 & !ent4 & !ent5;
    t7 := ent7 & !ent1 & !ent2 & !ent3 & !ent4 & !ent5;
```

Figure 3.37: pri : $scope\ inner$ (Furnace HTS)

Figure 3.37 shows the definition of priority-enabled macros using parameter value $scope$

[8] A transition's scope is the lowest common ancestor state of the transition's source and destination states.

$inner$, which is the opposite of $scope\ outer$. In this semantics, the transitions whose scopes are farther away from the top of the state hierarchy have higher priority, therefore, *t6* or *t7* can only be really enabled if none of the transitions from *t1* to *t5* is enabled.

Parameter value $explicit$ means that each transition in a system has an explicit integer value associated with it to indicate the priority of the transition. In this priority scheme, a partial order for all transitions can be determined, and a transition is priority enabled only if no transition of higher priority is enabled.

An HTS is enabled if any of its transitions are priority enabled. The semantics of the composition operator affects whether a composed HTS is enabled. The enabled macro `c.any` for each composed HTS `c` can be defined according to its components' enabled macros (`c1.any` and `c2.any`) and the semantics of the composition operator. Figure 3.38 shows the definitions of enabled macros for each of the composition operators in template semantics.

If the operator is parallel, parallel Harel, interleaving, choice, or sequence composition, `c.any` is the disjunction of the two components' macros (`c1.any`, `c2.any`).

If the operator is interrupt composition, `c.any` is the disjunction of `c1.any`, `c2.any`, and `interrTran.any`, which indicates whether any of the interrupt transitions are enabled. [9]

If the operator is environmental synchronization on events *a* and *b* [10], extra enabled macros are needed for each synchronization event (e.g., `c1.a_trig`) to indicate whether a component (e.g., `c1`) is enabled by an environment synchronization event, and a macro `c1.env_other_trig` to indicate that the HTS is enabled on transitions not triggered by any of the synchronization events. The composed HTS `c` is enabled when both components are enabled on the same synchronization event, or when either component has enabled transitions that are not related to the synchronization events. For each composed component that is a descendant of an environmental synchronization operator, a macro (e.g., `a_trig`) is added for each synchronization event (e.g., `a`) to the `enabled` module to indicate whether the component is enabled on the synchronization event, and a macro (`env_other_trig`) to indicate whether the component is enabled on other events. Each of these macros is equal to the disjunction of the subcomponents' macros. Figure 3.40 shows the definitions of these extra enabled macros for an HTS X (shown in Fig-

[9] Interrupt composition passes control between two components if an enabled interrupt transition has higher priority than any enabled transition in the active component. We will explain how priority is enforced in the section 3.7.7.

[10] Express requires that if a transition is triggered by an environmental or rendezvous event, it is triggered by only one event.

```
MODULE enabled(ss)
  DEFINE
    ...
    --parallel, paraHarel, interleaving, choice, sequence
    c.any := c1.any | c2.any;

    --interrupt
    c.any := c1.any | c2.any | interrTran.any;

    --environmental synchronization on a, b
    c.any :=   c1.a_trig & c2.a_trig
             | c1.b_trig & c2.b_trig
             | c1.env_other_trig
             | c2.env_other_trig;

    --rendezvous synchronization on a, b
    c.any :=   c1.a_trig & c2.a_gen
             | c1.a_gen & c2.a_trig
             | c1.b_trig & c2.b_gen
             | c1.b_gen & c2.b_trig
             | c1.rend_other_trig
             | c2.rend_other_trig;
```

Figure 3.38: Enabled Macros for Composite HTS Using Different Operators

ure 3.39) that is a descendant of an environmental synchronization on events *a* and *b* in the composition hierarchy. HTS X is enabled on event *a* (`a_trig`) when either transition *t1* or transition *t2* is priority enabled; it is enabled on event *b* (`b_trig`) if transition *t3* is priority enabled; it is enabled on other events (`env_other_trig`) if transition *t4* is priority enabled.

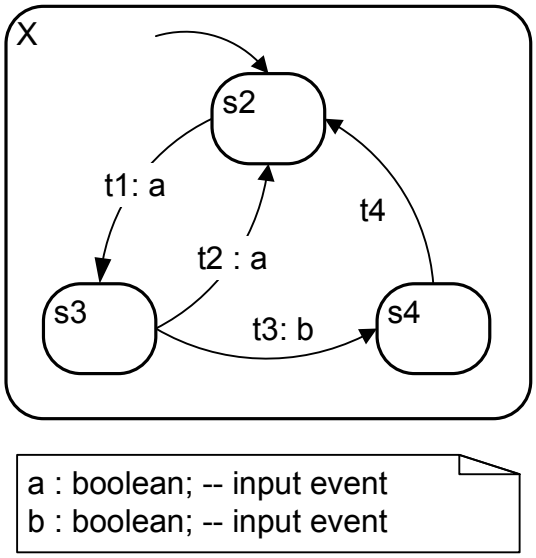


Figure 3.39: Example to Illustrate Extra Enabled Macros for Environmental Synchronization

```
MODULE enabled_X(ss)
  DEFINE
    ...
    a_trig := t1 | t2;
    b_trig := t3;
    env_other_trig := t4;
    ...
```

Figure 3.40: Enabled Macros for Environmental Synchronization for Figure 3.39

If the operator is rendezvous synchronization on event *a* and *b*, extra enabled macros are needed for each synchronization event (e.g., *a*) to indicate whether a component (e.g., `c1`) is enabled on *a* (e.g., `c1.a_trig`), or is generating this rendezvous synchronization event (e.g., `c1.a_gen`), or is enabled with transition triggers that are not related to any of the synchronization events `c1.rend_other_trig`. The composed HTS `c` is enabled when one component HTS is triggered on a rendezvous event, while the other component is generating the same event, or if either of the component HTSs has enabled transitions that are not related to the synchronization events. For each composed component that is a descendant of an environmental synchronization operator, two macros (e.g., `a_trig` and `a_gen`) are added for each rendezvous event (e.g., `a`) to the `enabled` module to indicate whether the component is enabled on or generating

the rendezvous event, and a macro (`rend_other_trig`) to indicate whether the component is enabled on other events. Each of these macros is equal to the disjunction of the subcomponents' macros. Figure 3.42 shows the definitions of these extra enabled macros for an HTS X (shown in Figure 3.41, which differs from Figure 3.39 in transition *t2*) that is a descendant of a rendezvous composition on events *a* and *b*. HTS X is enabled on event *a* (`a_trig`) when transition *t1* is priority enabled; it generates event *a* (`a_gen`) when transition *t2* is priority enabled; it is enabled on event *b* (`b_trig`) if transition *t3* is priority enabled; it cannot generate event *b*, therefore, `b_gen` is always 0; it is enabled on other events (`rend_other_trig`) if transition *t4* is priority enabled.

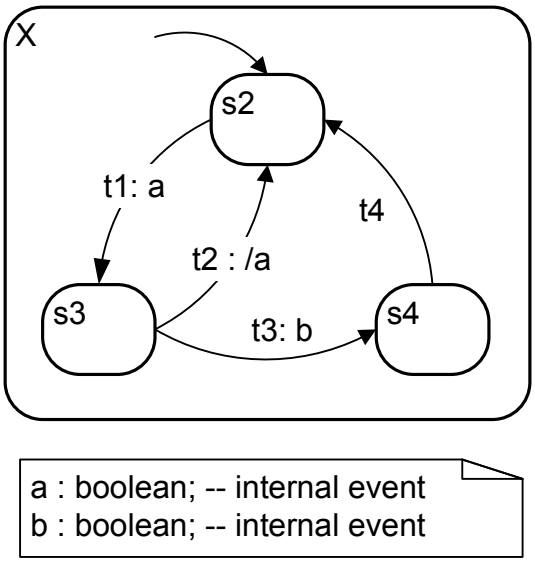


Figure 3.41: Example to Illustrate Extra Enabled Macros for Rendezvous

```
MODULE enabled_X(ss)
  DEFINE
    ...
    a_trig := t1;
    a_gen := t2;
    b_trig := t3;
    b_gen := 0;
    rend_other_trig := t4;
    ...
```

Figure 3.42: Enabled Macros for Rendezvous Composition for Figure 3.41

3.7 Execute Module

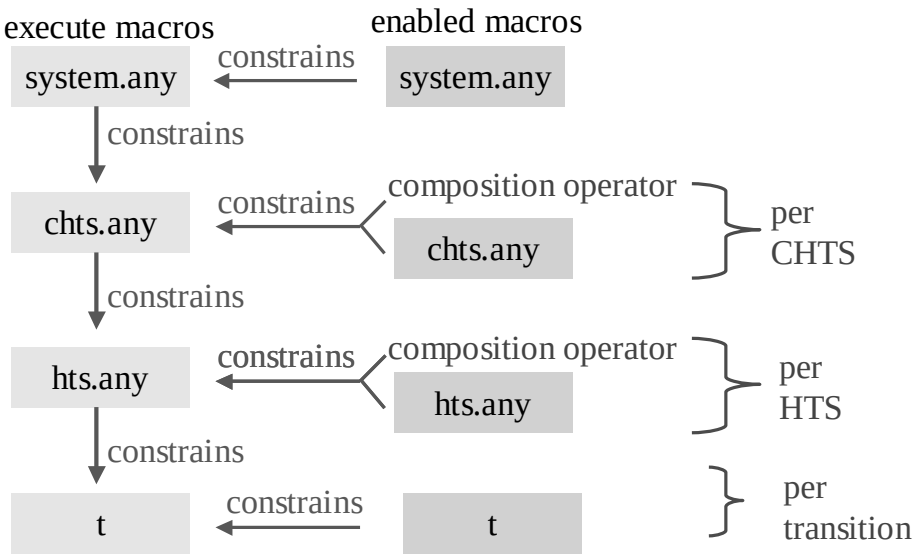


Figure 3.43: Dependency of Execute Macros

Module `execute` reads the enabled macros (passed as parameter `en`), and decides which enabled transitions and components should execute. Module `execute` realizes the meaning of the specification's composition operators by creating `iss_exe`, which is mostly a set of macros

indicating whether entities execute in the current step. Module `execute` works by constraining which entities can execute, rather than by selecting entities to execute. This way, the model's behaviours include all executions that meet these constraints. Figure 3.43 shows both execute macros (left column) and enabled macros (right column) of a system and their dependencies. Each execute macro is constrained by its corresponding enabled macro, for example, a system executes (execute macro `system.any` on the left column) only if it is enabled (enabled macro `system.any` on the right column). A component of a composition operator (either an HTS or a composed HTS) executes if it is enabled, the composed HTS executes, and the semantics of the composition operator allows it to execute. A transition executes if it is enabled and the HTS it belongs to allows it to execute.

```
MODULE execute(en)
  VAR
    noHeatReq:  execute_noHeatReq(en.noHeatReq);
    heatReq:    execute_HeatReq(en.heatReq);
    controller: execute_controller(en.controller);
    furnace:    execute_furnace(en.furnace);
    -- interrupt transitions
    roomIntrTrans:  execute_roomIntrTrans(en.roomIntrTrans);
    room:   interrupt(en.noHeatReq, en.heatReq,
                        en.roomIntrTrans, noHeatReq,
                    heatReq, roomIntrTran);
    house:  parallel(en.room, en.controller,
                    room, controller);
    heatingSys:  parallel(en.house, en.furnace,
                         house, furnace);
  INVAR
    -- require diligence
    en.heatingSys.any -> heatingSys.any
```

Figure 3.44: Execute Module (Heating System)

Figure 3.44 shows the module `execute` for the heating system, which takes the record of enabled macros,`en`, as an argument. The macros associated with each HTS (e.g., *furnace*) are set in the HTS's related sub-module (e.g., `execute_furnace`), and the macros for each composed component (e.g., *house*) are set in an instance of the component's composition operator's sub-module (e.g., `parallel`). For interrupt composition, there is another sub-module (e.g.,

`execute_roomIntrTrans`), which constrains whether the interrupt transitions execute; this sub-module is similar to the execute sub-module for an HTS. Every component `c` has a macro named `c.any` that indicates whether the component executes in the current step.

For a specification with composition operators, our translator generates one module for each type of composition operator that exists in the specification, and this module is instantiated for every instance of the operator in the specification. For example, there are two parallel compositions, and one interrupt composition in the heating system, and our translator generates one SMV sub-module for parallel composition (`parallel`), which is instantiated twice in the SMV `execute` module (`house` and `heatingSys`), and one SMV sub-module for interrupt composition (`interrupt`), which is instantiated once (`room`) in the `execute` module.

Template parameter *macro_semantics* defines a notation's macro-step semantics, which is either simple or stable. Simple semantics can be either diligent, or nondiligent. Stable semantics is always diligent since a stable snapshot indicates the end of a macro-step. We have explained that we handle a macro-step using a sequence of micro-steps in which the first micro-step, representing the start of the macro-step, has a conditional reset of the snapshot elements. Therefore, we need only to consider whether a system is diligent or nondiligent. In a diligent system, enabled transitions have priority over an idle step. Figure 3.44 shows the heating system with diligent semantics. The invariant ensures that if the system is enabled, it must execute. If a system is nondiligent, it might not execute even when it is enabled. This semantics can be realized by removing the invariant from the above SMV model, thus the SMV model checker will choose nondeterministically either to execute enabled transitions or to leave all snapshot elements unchanged.

Figure 3.45 shows part of the sub-module `execute_furnace`. Each HTS sub-module declares an enumerated variable `tran`, whose type has a value for each of the HTS's transitions, plus a value `noTranExe`, for the case where the HTS executes no transition in a step. A macro (e.g., `t1`) is defined for each transition to indicate whether the transition executes in the step. The sub-module's invariant constrains the model so that only priority-enabled transitions can execute. Because we use an enumerated variable (`tran`), only one transition in an HTS can execute in each step.

Next, we describe how the SMV sub-modules represent the composition operators of template semantics by using the macros in the `execute` module (which we call the `exe` record)

```
MODULE execute_furnace(en)
  VAR
    tran: {t1exe,t2exe,t3exe,t4exe,t5exe,t6exe,t7exe,noTranExe};
  DEFINE
    t1 := tran=t1exe;
    t2 := tran=t2exe;
    ...
    any := !(tran=noTranexe);
  INVAR
    (t1 -> en.t1)
   &(t2 -> en.t2)
   &(t3 -> en.t3)
   &(t4 -> en.t4)
   &(t5 -> en.t5)
   &(t6 -> en.t6)
   &(t7 -> en.t7)
```

Figure 3.45: Execute Sub-module (Furnace HTS)

to constrain the behaviour of their components. All the composition operators of template semantics are represented using SMV synchronous composition of modules. Basically, each composition operator constrains when its sub-components can execute. Figure 3.46 shows the parts common to all composition modules. A composition module takes as parameters the enabled sub-records (`enLeft, enRight`) and execution sub-records (`exeLeft, exeRight`) for its two operands, and produces an execution sub-record for the composed component. The execute macro (`any`) for the composed component is equal to the disjunction of the execution macros (`exeLeft.any`, and `exeRight.any`) for its components. Each composition module uses an invariant to constrain the execution of its components according to the behaviour of composition. The composition operator modules vary on how their components should execute when the composite HTS can execute. Some composition operators require additional macros in the `enabled` and `execute` modules for all component HTSs and composed component HTSs below them in the hierarchy, but these macros are added only as needed.

```
MODULE opername(enLeft, enRight, exeLeft, exeRight)
  DEFINE
    any := exeLeft.any | exeRight.any;
  INVAR
    ...
```

Figure 3.46: Common Parts of Composition Operator Sub-modules

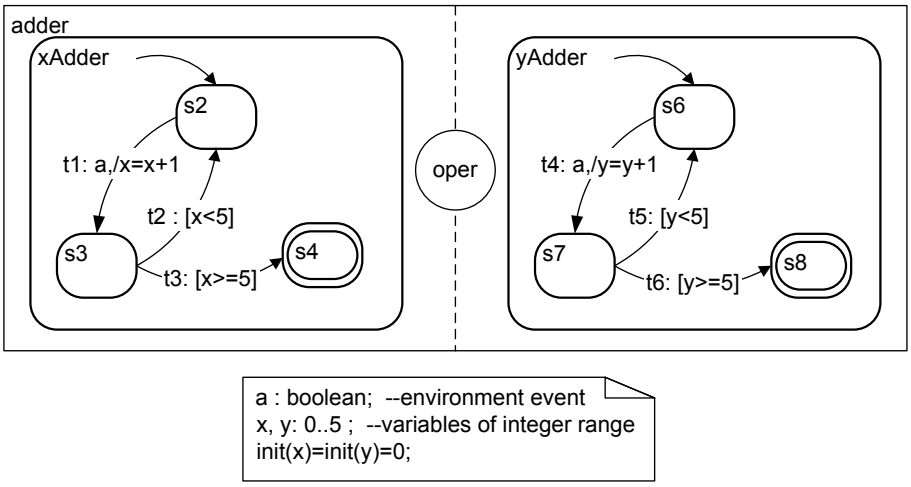


Figure 3.47: Example to Illustrate Composition

Figure 3.47 shows a specification of two HTSs composed by a composition. The two HTSs, *xAdder* and *yAdder*, increase the values of variable *x* and variable *y* when they receive the environment event *a*. Variables *x* and *y* are of type integer range from 0 to 5, and they are initialized to 0 at the start of system execution. Both HTSs reach their final states (states *s4* and *s8*) when the variables reach their maximum values. We will use this specification to discuss parallel, parallel Harel, interleaving, choice, and sequence compositions, and we will use three variations of this specification in Figure 3.54, 3.57, and 3.61 to discuss environmental synchronization, rendezvous, and interrupt compositions. For the properties we describe, we assume a fairness constraint that all environment events happen infinitely often, so that when the system waits for an input, it will eventually get the input.

3.7.1 Parallel

Template semantics has two types of parallel compositions: parallel composition and parallel Harel composition. In parallel composition, if both components are enabled, they both execute; otherwise, if only one component is enabled, the component executes and the other component does not. In parallel Harel composition, even if both components are enabled, one of the components may not execute.

SMV's synchronous composition of modules matches the meaning of template semantics' parallel composition operator directly. However, in order to represent a specification that uses multiple types of composition, including parallel, we use the `exe` macros to capture the meaning of parallel composition explicitly.

```
MODULE parallel(enLeft,enRight,exeLeft,exeRight)
 DEFINE
  any := exeLeft.any | exeRight.any;
 INVAR
  (any ->
      ((enLeft.any -> exeLeft.any) & (enRight.any -> exeRight.any)))
```

Figure 3.48: Parallel Composition

Figure 3.48 shows the module representing the parallel composition of template semantics. Each component must execute when enabled, even if both components are enabled. For the specification of Figure 3.47 using parallel composition, the transitions *t1* and *t4*, when enabled, execute in the same step; as can *t2* and *t5*; and *t3* and *t6*. Therefore, the following properties hold in the specification:

- Property 1 : variable *x* is always equal to variable *y*.
  `AG (pss.AV.x=pss.AV.y)`[11]
- Property 2: transition *t1* and transition *t4* always execute at the same time.
  `AG (exe.xAdder.t1=exe.yAdder.t4)`

Figure 3.49 shows the module representing the parallel Harel composition in template semantics. It captures the meaning of parallel composition in Harel's original definition of statecharts

[11] `AG f` means that formula `f` holds globally in the system.

[17]: if both components are enabled, both may execute, or only one may execute. It differs from parallel composition in that, when both components are enabled, parallel Harel composition allows only one component to execute, while parallel forces both component to execute. No operator invariant is needed to capture the behaviour of parallel Harel because the top-level `execute` module enforces diligence (i.e., if the system is enabled then it must execute), which means that one of `exeLeft.any` or `exeRight.any` must be true.

```
MODULE paraHarel(enLeft,enRight,exeLeft,exeRight)
  DEFINE
    any := exeLeft.any | exeRight.any;
```

Figure 3.49: Parallel Harel Composition

For the specification of Figure 3.47 using parallel Harel composition, when transitions *t1* and *t4* are both enabled, there are three options for a step: 1) both execute; 2) *t1* executes, but *t4* does not execute; 3) *t4* executes, but *t1* does not. One of the options is nondeterministically chosen by SMV. Therefore, Properties 1 and 2 described above do not hold in this specification, but the following properties hold :

- Property 3: eventually variable *x* is equal to variable *y*.
  `AF (pss.AV.x=pss.AV.y)`[12]
- Property 4: transition *t1* and transition *t4* may execute at the same time.
  `EF (exe.xAdder.t1=exe.yAdder.t4)`[13]

3.7.2 Interleaving

Interleaving composition allows only one component to execute transitions in each step. Figure 3.50 shows the module representing interleaving composition[14] . The invariant disallows both components from executing in the same micro-step. If we use interleaving composition in Figure 3.47, when transitions *t1* and *t4* are both enabled, only either *t1* or *t4* can execute. In this

[12] `AF f` means that formula `f` is inevitable, that is, `f` holds in some future state for all paths the system can take.
[13] `EF f` means that formula `f` is reachable, that is, there exists a path that f holds in some future state.
[14] SMV provides asynchronous composition of modules, which is similar to interleaving, but does not enforce diligence.

```
MODULE interleaving(enLeft,enRight,exeLeft,exeRight)
  DEFINE
    any:=exeLeft.any | exeRight.any;
  INVAR
   !(exeLeft.any & exeRight.any)
```

Figure 3.50: Interleaving Composition

specification, Properties 1, 2, and 4 do not hold because at each step only one HTS can execute a transition; but Property 3 holds, since eventually both HTSs will reach their final states.

3.7.3 Choice

The choice composition operator nondeterministically chooses one component to execute in isolation. Once the choice is made, the composite machine behaves only like the chosen component, and never executes the other component.

Figure 3.51 shows the module representing choice composition. To capture the nondeterministic choice on which component to execute, a boolean variable `choice` is added in the sub-module. This variable is set nondeterministically by SMV to either 0 or 1, when chosen, it keeps its value. For example, if 1 is chosen, the left component is chosen to execute, the right component is ignored; otherwise, the right component is chosen to execute, and the left component is ignored.

In the specification in Figure 3.47 using choice composition, only one HTS is chosen to execute, and the other HTS is ignored. Therefore, transition *t1* and transition *t4* cannot execute at the same step, so Properties 2 and 4 are false. Property 3 is true, because the variables are both set to 0 initially. It is not possible that *x* and *y* are always equal, which indicates that Property 1 is false. However, the following property holds in this specification.

- Property 5: there exists some future state in which variable *x* reaches its maximal value 5, and variable *y* is still 0.
  `EF(pss.AV.x=5 & pss.AV.y=0)`

```
MODULE choice(enLeft,enRight,exeLeft,exeRight)
  DEFINE
    any := exeLeft.any | exeRight.any;
  VAR
    choice: boolean;
  ASSIGN
    init(choice) := {0,1};
    next(choice) := choice;
  INVAR
    (choice -> !exeRight.any)
  & (!choice -> !exeLeft.any)
```

Figure 3.51: Choice Composition

3.7.4 Sequence

In sequence composition, the left component executes in isolation until it reaches its final states, and then the right component executes in isolation. To be used in sequence composition, the left component must have final states, otherwise, the right component does not have a chance to execute.

```
MODULE sequence(enLeft,enRight,exeLeft,exeRight)
  DEFINE
    any := exeLeft.any | exeRight.any;
  INVAR
    (!enLeft.final -> !exeRight.any)
  & (enLeft.final ->  !exeLeft.any)
```

Figure 3.52: Sequence Composition

Figure 3.52 shows the module representing sequence composition. To capture whether the left component has reached its final state, and to decide whether the right component can start to execute if enabled, an additional macro called `final` is added to the `enabled` module for each HTS and each composed HTS that is a descendant of a sequence composition. For an HTS, the macro is set to be the disjunction of the macros indicating whether the HTS is in a final state, and

added to the HTS's corresponding `enabled` sub-module. Figure 3.53 shows the `final` macro for HTS *xAdder* (Figure 3.47 using sequence composition). For a composed HTS (other than the composed HTS of a sequence composition) that is a descendant of the sequence composition, macro `final` is equal to the conjunction of the macros indicating whether the sub-components are in final states. For sequence composition, the composed component is in its final state if its right component is in its final states. Using these macros from the `enabled` module, the left component in sequence composition can only execute if it is not in its final states.

According to the specification, HTS *xAdder* executes in isolation until it reaches its final state *s4*, then HTS *yAdder* starts to execute if it is enabled. Properties 2 and 4 do not hold because it is not possible that in sequence composition both HTSs execute in a step. Property 1 does not hold because before HTS *xAdder* reaches its final state, and *x* is 5, HTS *yAdder* stays still, with *y* being is 0, so it is not possible that *x* and *y* are always equal. Property 3 holds because eventually both HTSs reach their final states, and the value of *x* and *y* are equal. In addition, Property 5 holds because HTS *xAdder* has to stop executing, before HTS *yAdder* starts to execute.

```
MODULE enabled_xAdder(ss)
  DEFINE
    ...
    final := ss.CS.in_s4;
    ...
```

Figure 3.53: Enabled Macro `Final` for Sequence Composition for Figure 3.47

3.7.5 Environmental Synchronization

In environmental synchronization composition, two components can take a step if they both take transitions triggered by the same **synchronization** event; otherwise their behaviour is interleaved (taking transitions not based on a synchronization event). The SMV module representing environmental synchronization is customized for the particular set of synchronization events.

In a system that uses environmental synchronization, we assume that the system can be synchronized on only one synchronization event at each micro-step. If environmental synchronization is used multiple places in the specification, the synchronization set at an ancestor level of

composition must explicitly be made part of the set at all descendant levels to represent the semantics of some notations such as LOTOS.

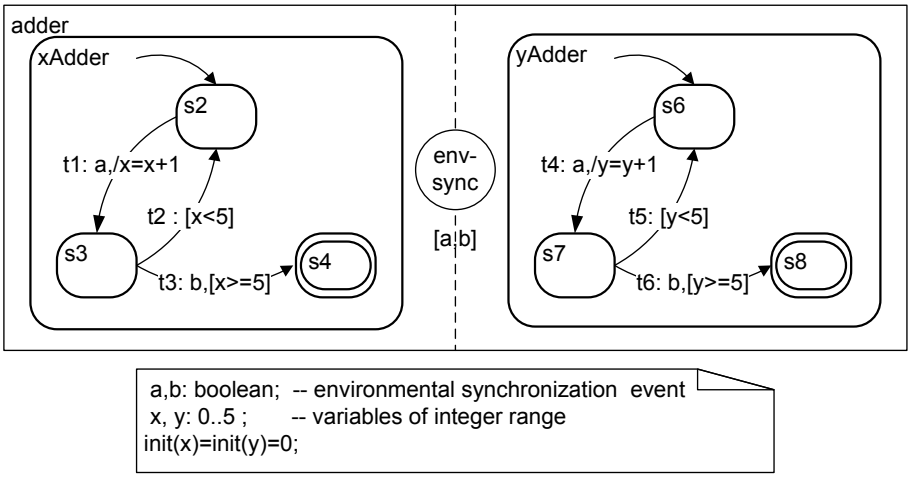


Figure 3.54: Example to Illustrate Environmental Synchronization Composition

Figure 3.54 shows a variation of the specification in Figure 3.47. It composes two HTSs using an environmental synchronization composition, and events *a* and *b* are designated synchronization events that are from the environment. Transitions *t1* and *t4*, *t3* and *t6* are two pairs of transitions that should be synchronized.

To indicate whether each component is executing transitions triggered by a synchronization event, we introduce for each synchronization event a macro (e.g., `e_trig`) in the `exe` record for each HTS and each composed HTS that is a descendant of the environmental synchronization composition. For each HTS, this macro is the disjunction of all transitions that are triggered by the event *e*, and it is added in the HTS's corresponding `execute` sub-module. For each composed HTS, this macro is the disjunction of its components' `e_trig` macros, and it is added in the `execute` module.

To capture whether each component is executing transitions triggered on events other than the synchronization events, we introduce a macro, `env_other_trig`, for each HTS and each composed HTS that is a descendant of the environmental synchronization composition. For each HTS, this macro is the disjunction of all transitions that are triggered by events other than synchronization events, and it is added in the HTS's corresponding `execute` sub-module. For each

composed HTS, this macro is the disjunction of its components' `env_other_trig` macros, and it is added in the `execute` module.

Within the sub-module that represents the environmental synchronization composition, we also introduce for each synchronization event an execute macro (e.g., `e_trig`) to determine whether the composed HTS that is created by environmental synchronization is synchronized on the event *e*. The synchronization occurs if both component executes a transition triggered by e. We also introduce a macro, `env_other_trig`, to determine whether the composed HTS that is created by environmental synchronization is executing transitions triggered by events other than the synchronization events.

```
MODULE envsync_a_b(enLeft,enRight,exeLeft,exeRight)
  DEFINE
    any := exeLeft.any | exeRight.any;
    a_trig := exeLeft.a_trig & exeRight.a_trig;
    b_trig := exeLeft.b_trig & exeRight.b_trig;
    env_other_trig :=  exeLeft.env_other_trig
                     | exeRight.env_other_trig;
  INVAR
  -- left and right are triggered on same sync event
    (exeLeft.a_trig <-> exeRight.a_trig)
  & (exeLeft.b_trig <-> exeRight.b_trig)
  -- component triggered by single sync event
  & !(exeLeft.a_trig & exeLeft.b_trig)
  -- when synchronizing,
  -- no transition triggered by other events can execute
  & (a_trig|b_trig) -> !env_other_trig)
  -- interleaved behaviour
  & (!(a_trig|b_trig) -> !(exeLeft.any & exeRight.any))
```

Figure 3.55: Environmental Synchronization Composition on Events a and b

Figure 3.55 shows the SMV sub-module for environmental synchronization on the event set $\{a, b\}$. Macros `exeLeft.a_trig` and `exeRight.a_trig` indicate whether the left and the right component are executing transitions triggered by the event a. When synchronizing, both components take transitions on the same synchronization event. The invariant constrains the `exe` flags such that if the left component takes transitions triggered by a, the right component must

also, and similarly for event b. As well, each component cannot take transitions triggered by more than one synchronization event. Furthermore, when synchronizing, neither components can take transitions triggered by other events. Finally, if transitions are not being taken on synchronization events, then only one of the components can execute.

```
MODULE execute_xAdder(en)
  DEFINE
    ...
    a_trig := t1;
    b_trig := t3;
    env_other_trig := t2;
    ...
```

```
MODULE execute_yAdder(en)
  DEFINE
    ...
    a_trig := t4;
    b_trig := t6;
    env_other_trig := t5;
    ...
```

Figure 3.56: Execution Macros for Environmental Synchronization for Figure 3.54

Figure 3.56 shows how the HTSs for *xAdder* and *yAdder* have the extra macros for the synchronization events $\{a, b\}$. There are no lower level compositions in Figure 3.54, so additional macros in the `execute` module is not needed in this example.

For the specification of Figure 3.54, transitions *t1* and *t4* always execute together in the same step, so do transitions *t3* and *t6*, therefore, Properties 1, 2, 3 and 4 hold in this specification. In this specification, Property 5 cannot be true because *x* and *y* are always equal.

3.7.6 Rendezvous

Rendezvous composition is found in notations such as CCS, and it means that exactly one transition in the sending component generates a **rendezvous** event that triggers in the *same* micro-step exactly one transition in the receiving component. Otherwise the behaviour of the components is interleaved (taking transitions not based on rendezvous events). Similar to environmental synchronization, this composition is based on an explicit set of rendezvous events, so the SMV module is customized for the set of rendezvous events.

In a system that uses rendezvous composition, we assume that the system can be synchronized on only one rendezvous event at each micro-step.We assume a transition cannot both be triggered by a rendezvous event and generate another rendezvous event.

To indicate whether each component is executing a transition triggered by a rendezvous event *e*, and is generating a rendezvous event *e*, we introduce for each rendezvous event two extra

macros (e.g., `e_trig`, `e_gen`) in the `exe` record for each HTS and each composed HTS that is a descendant of the rendezvous composition. For each HTS, macro `e_trig` is the disjunction of all transitions that are triggered by the event *e*; macro `e_gen` is the disjunction of all transitions that generates the event. They are defined in the HTS's corresponding `execute` sub-module. For each composed HTS, these two macros are the disjunction of its components' corresponding macros, and they are defined in the `execute` module.

To capture whether exactly one transition executes in each component, we introduce one extra macro, `more_than_one`, in the `exe` record for each HTS and each composed HTS that is a descendant of the rendezvous composition. Macro `more_than_one` is always false in an HTS because only one transition can be taken in a micro-step. This macro is added in the HTS's corresponding `execute` sub-module. For each composed HTS that is a descendant of the rendezvous composition, this macro is the disjunction of the left and right component's `more_than_one` macros, and the conjunction of the left and right component's execute flags, which means that a composed HTS has more than one executing transitions if either component has more than one executing transitions, or if both components execute. The macro `more_than_one` for each composed HTS is added in the `execute` module.

We also introduce for each rendezvous event an execute macro (e.g., `e_rend`) to determine whether the composed HTS that is created by rendezvous composition is rendezvousing on the event *e*. A rendezvous occurs if the left component executes a transition triggered by e and the right component generates e, or vice versa for each rendezvous event. This macro is declared in the sub-module that represents the rendezvous composition.

Figure 3.57 shows another variation of the specification in Figure 3.47, in which the two HTSs are composed by an rendezvous composition. Events *a* and *b* are designated rendezvous events that are internal events. Transitions *t1* and *t4*, *t3* and *t6* are two pairs of transitions that should be synchronized.

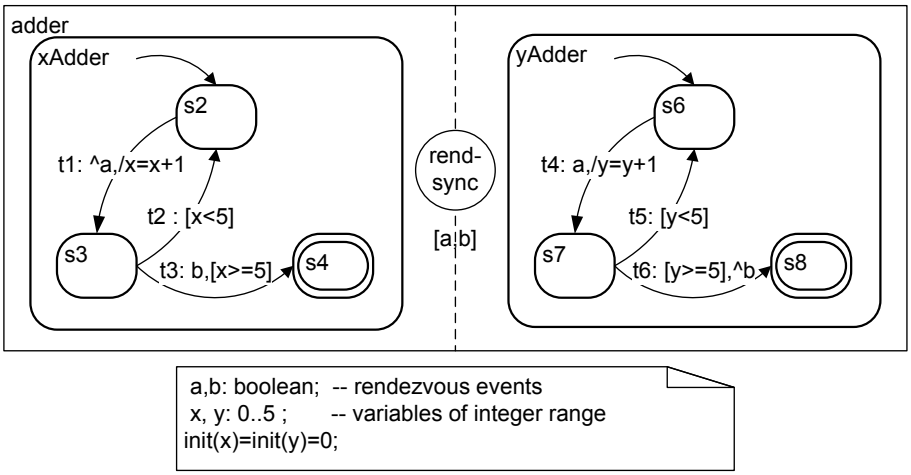


Figure 3.57: Example to Illustrate Rendezvous Composition

```
MODULE rendezvous_a_b(enLeft,enRight,exeLeft,exeRight)
  DEFINE
    any:=exeLeft.any | exeRight.any;
    -- rendezvous means one generates and other triggers
    a_rend := (exeLeft.a_trig & exeRight.a_gen)
            | (exeLeft.a_gen & exeRight.a_trig);
    b_rend := (exeLeft.b_trig & exeRight.b_gen)
            | (exeLeft.b_gen & exeRight.b_trig);
  INVAR
    -- left and right are trig/gen on same sync event
    (exeLeft.a_trig <-> exeRight.a_gen)
  & (exeLeft.a_gen  <-> exeRight.a_trig)
  & (exeLeft.b_trig <-> exeRight.b_gen)
  & (exeLeft.b_gen  <-> exeRight.b_trig)
    -- if rendezvous, only one trans execute
    -- in each component
  & ((a_rend | b_rend) ->
      ! (exeLeft.more_than_one | exeRight.more_than_one))
  -- interleaved behaviour
  & (!(a_rend|b_rend) -> !(exeLeft.any & exeRight.any))
```

Figure 3.58: Rendezvous Composition on Events a and b

Figure 3.58 shows the SMV module for rendezvous synchronization with rendezvous events $\{a, b\}$. Macros `exeLeft.a_trig` and `exLeft.a_gen` indicate whether the left component is taking a transition that is triggered by, or generates a rendezvous event a. A rendezvous (e.g., `a_rend`) occurs if the left component executes a transition triggered by a and the right component generates a, and vice versa for each rendezvous event. The invariant enforces the constraint that in a rendezvous, the left component must be triggered on a rendezvous event when the right component generates that event, and the opposite for each synchronization event. The invariant also enforces the constraint that only one transition can be taken in each component if a rendezvous is occurring using the macro `more_than_one` in the `exe` record. By limiting the components to one transition each, we ensure the two transitions that are taken are triggered by or generate rendezvous events.

```
MODULE execute_xAdder(en)
  DEFINE
    ...
    a_trig := 0;
    a_gen := t1;
    b_trig := t3;
    b_gen := 0;
    more_than_one := 0;
    ...
```

```
MODULE execute_yAdder(en)
  DEFINE
    ...
    a_trig := t4;
    a_gen : = 0;
    b_trig := 0;
    b_gen := t6;
    more_than_one :=0;
    ...
```

Figure 3.59: Execution Macros for Rendezvous Composition for Figure 3.57

Figure 3.59 shows the `a_trig`, `a_gen`, `b_trig`, `b_gen`, and `more_than_one` execute macros for the two HTSs in the specification in Figure 3.57.

```
MODULE enabled(ss)
  VAR
    xAdder : enabled_xAdder(ss, sync_events);
    yAdder : enabled_yAdder(ss, sync_events);
    ...
  DEFINE
    -- include all transitions that generate a
    sync_events.a := xAdder.t1;
    -- include all transitions that generate b
    sync_events.b := yAdder.t6;
    ...

MODULE enabled_xAdder(ss, sync_events)
  DEFINE
    enStates_t3 := ss.CS.in_s3 ;
    enEvents_t3 := sync_events.b;
    enCond_t3 := ss.AV.x>=5;
    t3 := enStates_t3 & enEvents_t3 & enCond_t3;
    ...

MODULE enabled_yAdder(ss, sync_events)
  DEFINE
    enStates_t4 := ss.CS.in_s6 ;
    enEvents_t4 := sync_events.a;
    enCond_t4 := 1;
    t4 := enStates_t4 & enEvents_t4 & enCond_t4;
    ...
```

Figure 3.60: Enabled Modules for Rendezvous Composition

Rendezvous composition also effects the way transitions are enabled. We have to represent how a transition in one component generates a rendezvous event that enables within the *same* step a transition in another component. Figure 3.60 shows part of the enabled macros in the `enabled` module for the specification in Figure 3.57. We introduce for each rendezvous event a

macro (e.g., `sync_events.a`) to be equal to the disjunction of the priority-enabled status of all transitions that generate that event. These macros are then used to determine the enabled macros for transitions that are triggered by rendezvous events. In the example, the enabling of transition *t3* of the *xAdder* HTS depends on the rendezvous macro for *a* rather than the event's status in the snapshot, and the enabling of transition *t4* of the *yAdder* HTS depends on the rendezvous macro for *b*. In fact, we can eliminate rendezvous events from the snapshot.

For the specification of Figure 3.57, transitions *t1* and *t4* always execute together in the same step, as do transitions *t3* and *t6*; therefore, Properties 1, 2, 3 and 4 hold. Property 5 does not hold.

3.7.7 Interrupt

Interrupt composition allows control to pass between two components via a provided set of interrupt transitions. Only one component in interrupt composition ever has current states, so only one component can have enabled transitions. At any micro-step, either the active component or an interrupt transition is chosen to execute, based on which has higher priority. If the active component and the interrupt transitions have the same priority, either may be chosen nondeterministically to execute. Therefore, interrupt composition depends on the value of template parameter *pri*.

In interrupt composition, besides the left and right components, the set of interrupt transitions is considered to be the third component, which we call the **interrupt component**. An `enabled` sub-module and an `execute` sub-module for the interrupt component are constructed by the translator to set the enabled macros and the execution macros for interrupt transitions, respectively.

To capture the priorities of the three components, an extra enabled macro, `pri`, is added for each transition, each HTS, and each composed HTS that is a descendant of the interrupt composition. For each transition (including interrupt transitions), it is defined by the parameter value of template predicate *pri*. For each HTS (including the interrupt component), it is the priority of the highest-priority enabled transition. For each composed HTS (other than a composed HTS that uses interrupt composition), it is the higher `pri` of the two subcomponents. For a composed HTS that uses interrupt composition, it is the highest `pri` among the three components. The macros for an HTS and its transitions are added to the HTS's corresponding `enabled` sub-module. The macros for composed HTSs are added to the `enabled` module.

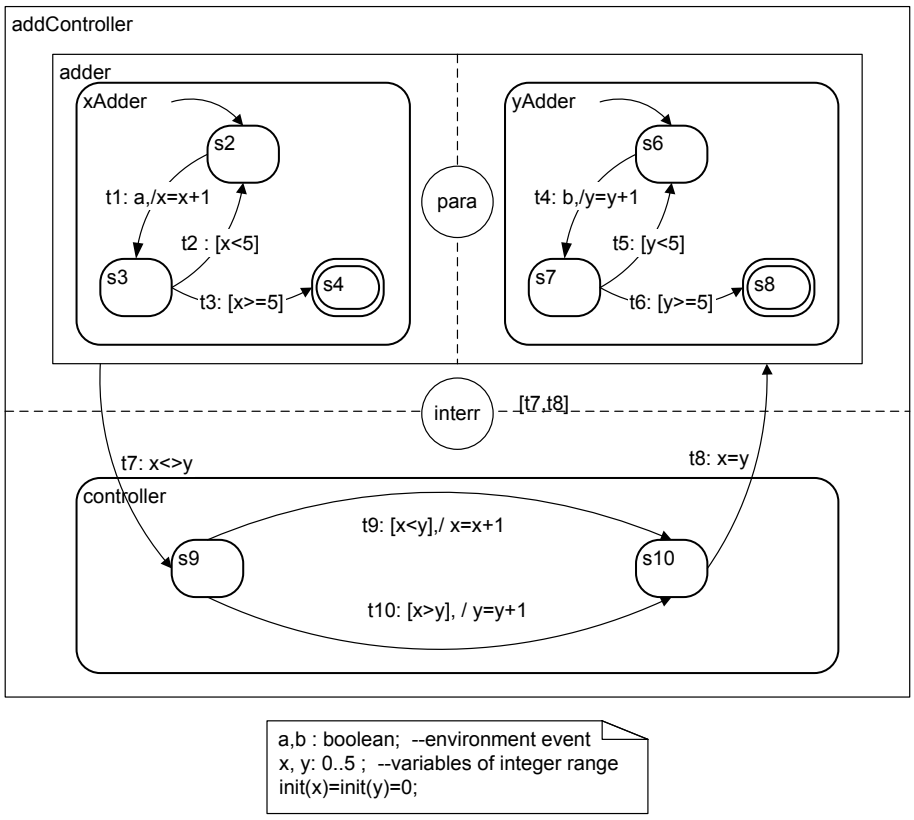


Figure 3.61: Example to Illustrate Interrupt Composition

We use the specification in Figure 3.61 (a variation of the specification in Figure 3.47) to illustrate this composition. In Figure 3.61, two HTSs *xAdder* and *yAdder* are composed by parallel composition, it is then composed with HTS *controller* using interrupt composition. Transitions *t7* and *t8* are interrupt transitions. We will use the priority $scope\ outer$ scheme in this specification. HTS *xAdder* increases the value of variable *x* when it receives the environment event *a*, and HTS *yAdder* increases the value of variable *y* when it receives the environment event *b*. At the start of the execution, HTSs *xAdder* and *yAdder* are in their initial states *s2* and *s6*, and they wait for the environment events to trigger their transitions. If both events occur, parallel composition forces both variables to increase by executing transitions *t1* and *t4*, and then executes transitions *t2* and *t5* to return to states *s2* and *s6* to wait for inputs. If only one event occurs, one variable increases its value and the other variable does not change. Then because the two variables are not

equal, the interrupt transition *t7* is enabled, which has higher priority than any enabled transitions in the composed HTS *adder*; therefore, *t7* executes, and passes the control from composed HTS *adder* to the HTS *controller*. The controller, based on the value of the two variables, increases the value of the smaller variable and passes control back to the composed HTS *adder*. When *adder* is entered, both *xAdder* and *yAdder* are in their initial states, and wait for transitions to be triggered.

For this example, an enabled sub-module (`enabled_addControllerIntrTran`) and an execution sub-module (`execute_addControllerIntrTran`) are constructed by the translator to set the enabled macros and the execution macros for interrupt transitions, respectively.

```
MODULE interrupt(enLeft,enRight,enIntrTrans,
                 exeLeft,exeRight,exeIntrTrans)
  DEFINE
    any := exeLeft.any | exeRight.any | exeIntrTrans.any;
  INVAR
      -- execute component with highest priority trans
      (exeLeft.any -> (enLeft.pri <= enIntrTrans.pri))
    & (exeRight.any -> (enRight.pri <= enIntrTrans.pri))
    & (exeIntrTrans.any ->
         ( (enIntrTrans.pri <= enLeft.pri)
          &(enIntrTrans.pri <= enRight.pri)) )
      -- cannot execute more than one of the two
      -- components or an interrupt trans
    & !(exeLeft.any & exeIntrTrans.any)
    & !(exeRight.any & exeIntrTrans.any)
```

Figure 3.62: Interrupt Composition

Figure 3.62 shows the module `interrupt` to represent interrupt composition. Interrupt composition has additional parameters, `enIntrTrans` and `exeIntrTrans`, which are sub-records of the `enabled` and `execute` modules and hold the enabling and execute flags for the interrupt transitions. Macros `enLeft.pri`, `enLeft.pri`, and `enIntrTrans.pri` indicate the priority of the left component, the right component, and the interrupt component. At any step, either the active component, or the interrupt component is chosen to execute, based on which has higher priority. If they have the same priority, either may be chosen nondeterministi-

cally to execute.

Figure 3.63 shows the priority fields that are added to `enabled_addControllerIntrTran` sub-module for the interrupt transitions, and to `enabled_xAdder` sub-module for the *xAdder* HTS. Using priority scheme $scope\ outer$, transition priorities are based on the scopes of the transitions; field `pri` is set to the priority of the priority-enabled transitions. Lower `pri` values denote higher priorities, with constant `MAX_PRI` representing the lowest priority in the system.

```
MODULE enabled_addControllerIntrTran(ss)
  DEFINE
    -- rank of scope
    pri_t7 := 0;
    pri_t8 := 0;
    pri := case
      t7 : pri_t7;
      t8 : pri_t8;
      1 : MAX_PRI;
    esac;
    ...
MODULE enabled_xAdder(ss)
  DEFINE
    -- rank of scope
    pri_t1 := 2;
    pri_t2 := 2;
    pri_t3 := 2;
    pri := case
      t1 : pri_t1;
      t2 : pri_t2;
      t3 : pri_t3;
      1 : MAX_PRI;
    esac;
```

Figure 3.63: Enabled Macros `pri` for Interrupt Composition for Figure 3.61

For the specification of Figure 3.61, Properties 1 and 2 do not hold because it is possible that both events *a* and *b* do not occur at the same time; thus transition *t1* may not execute together with transition *t4*. Property 3 holds because eventually when the system reaches the final states, the values of *x* and *y* are the same. Property 4 holds because it is possible that both events occur,

and both transitions *t1* and *t4* execute. Property 5 does not hold because whenever *x* and *y* are not equal, the *controller* HTS becomes active, and adjusts the values of *x* and *y* to be equal before control goes back to the *adder* composed HTS. Therefore, at any time, the controller forbids the difference between the two variables to be greater than 1 (Property 6).

- Property 6: At any time, the difference between variable *x* and *y* won't be greater than 1.

```
AG(((pss.AV.y - pss.AV.x)<=1)
        | ((pss.AV.x - pss.AV.y)<=1))
```

For a specification that uses interrupt transitions, the execution of the interrupt transitions update snapshot elements as the regular transitions do.

3.8 Apply Module

```
MODULE apply(pss,iss,exe)
  VAR
    nextCS : nextCS(pss,iss,exe);
    nextAV : nextAV(pss,iss,exe);
    nextIE : nextIE(pss,iss,exe);
    nextIa : nextIa(pss,iss,exe);
    nextO  : nextO(pss,iss,exe);
```

Figure 3.64: Apply Module (STATEMATE)

Module `apply` sets the next value of snapshot `pss`, based on the reset snapshot `iss`, and the effects of the executing transitions (including interrupt transitions) constrained by the `execute` module (parameter `exe`). It updates each snapshot element *XX* in a separate sub-module `nextXX`, which realizes the semantics of the provided template-parameter value of *next_XX*. For example, Figure 3.64 shows the module `apply` for STATEMATE.

Table 3.4 (page 73) and Table 3.5 (page 74) show *next_XX* parameter values supported by our translator. In the following, we explain how to represent these parameter values in SMV.

Parameter Name	Parameter Value	Notations	SMV Module
$next\_CS$	$CS' = entered(dest(\tau))$	statecharts, RSML, STATEMATE, CCS, CSP, LOTOS	Figure 3.65
$next\_CSa$	n/a	CCS, CSP, LOTOS, BTS, RSML, STATEMATE	n/a
	$CSa' = \phi$	statecharts	Figure 3.67
$next\_IE$	n/a	CCS, CSP, LOTOS, BTS	n/a
	$IE' = gen(\tau)$	STATEMATE	Figure 3.69
	$IE' = gen(\tau) \cap intern\_ev(E)$	RSML	Figure 3.70
	$IE' = ss.IE \cup gen(\tau)$	statecharts	Figure 3.71
	$IE' = ((ss.IE - trig(\tau)) \cup gen(\tau))$	(pre-defined value)	Figure 3.72
$next\_IEa$	n/a	CCS, CSP, LOTOS, BTS, statecharts, RSML, STATEMATE	n/a
$next\_O$	$O' = n/a$	LOTOS	n/a
	$O' = gen(\tau)$	CCS, CSP, STATEMATE	Figure 3.73
	$O' = ss.O \cup gen(\tau)$	statecharts	Figure 3.75
	$O' = ss.O \cup (gen(\tau) \cap extern\_ev(E))$	RSML	Figure 3.74
$next\_Ia$	n/a	BTS	n/a
	$true$	LOTOS	(not shown)
	$Ia' = \phi$	RSML, STATEMATE	Figure 3.77
	$Ia' = ss.Ia$	statecharts	Figure 3.78
	$Ia' = ss.Ia \cup gen(\tau)$	CCS, CSP	Figure 3.79
	$Ia' = ((Ia - trig(\tau)) \cup gen(\tau))$	(pre-defined value)	Figure 3.80

Table 3.4: Predicate $next\_XX(ss, \tau, XX')$ (to be continued)

Parameter Name	Parameter Value	Notations	SMV Module
$next\_AV$	n/a	CCS, CSP, LOTOS	n/a
	$AV' = \textit{assign}(ss.AV,$ $\textit{eval}(ss.AV,$ $last(\textit{asn}(\tau))))$	STATEMATE (choice 2)	Figure 3.84
	$AV' = \textit{assign}(ss.AV,$ $\textit{eval}(ss.AV,$ $\textit{asn}(\tau)))$	BTS, RSML (choice 3)	Figure 3.85
	$AV' = \textit{assign}(ss.AV,$ $\textit{eval}((ss.AV, ss.AVa),$ $\textit{asn}(\tau)))$	statecharts (choice 4)	Figure 3.86
	$AV' = \textit{assign}(ss.AV,$ $\textit{eval}(ss.AV,$ $\textit{accum}(\textit{asn}(\tau))))$	(pre-defined value) (choice 5)	Figure 3.87
	$AV' = \textit{assign}(ss.AV,$ $\textit{eval}(ss.AV,$ $\textit{nondeterm}(\textit{asn}(\tau))))$	(pre-defined value) (choice 6)	Figure 3.88
$next\_AVa$	n/a	CCS, CSP, LOTOS, BTS, RSML, STATEMATE	n/a
	$AVa' = ss.AVa$	statecharts	Figure 3.89

Table 3.5: Predicate $next\_XX(ss, \tau, XX')$ (continued)

3.8.1 NextCS Sub-module

Sub-module `nextCS` implements the predicate *next_CS(*ss, τ, CS'*)*, which has one supported parameter value $CS' = entered(dest(\tau))$. This value indicates that the next state of the HTS's state variable is the basic-state descendant of the destination state of the executing transition. If an HTS has no hierarchy, this parameter value simplifies to $CS' = dest(\tau)$, the destination state of the executing transition.

```
MODULE nextCS(pss,iss,exe)
  ASSIGN
    next(pss.CS.furnace_state) := case
      exe.furnace.t1 : furnaceAct;
      exe.furnace.t2 : furnaceOff;
      exe.furnace.t3 : furnaceRun;
      exe.furnace.t4 : furnaceOff;
      exe.furnace.t5 : furnaceAct;
      exe.furnace.t6 : furnaceOff;
      exe.furnace.t7 : furnaceErr;
      1 : iss.CS.furnace_state;
    esac;
    ...
```

Figure 3.65: NextCS Sub-module : $CS' = entered(dest(\tau))$ (Heating System)

In the `nextCS` sub-module, the next value of each state variable, one per HTS, in snapshot element CS is set using a `case` statement that contains a condition for each of the transitions of the HTS. Figure 3.65 shows the `case` statement for `furnace_state`, which is the state variable for the HTS *furnace* in the heating system. HTS *furnace* has 7 transitions (*t1* to *t7*): if *t1* is executed, in the next step, the state variable `furnace_state` is `furnaceAct`; if *t6* is executed, the state variable is `furnaceOff` in the next step, which is the default basic-state descendant of *t6*'s destination state `furnaceNormal`. If no transition executes, the state variable has the same value as it does in the reset snapshot `iss`.

Using interrupt composition causes changes to the `nextCS` sub-module, for example, if the transition chosen to execute is one of the interrupt transitions, the system leaves the states of its source component and enters the appropriate states in the destination component. Figure 3.66 shows how the next value of state variables `noHeatReq_state` and `heatReq_state` in the

`nextCS` sub-module for the heating system are modified to accommodate interrupt transitions *t19* and *t20*. When interrupt transition *t20* executes, the current state of the source component becomes `noState`, and the current state of the destination component becomes `idleHeat`, the default basic-state descendant of the transition's destination.

```
MODULE nextCS(pss,iss,exe)
  ASSIGN
    next(pss.CS.noHeatReq_state):=case
      exe.noHeatReq.t15 : waitForHeat;
      ...
      exe.roomIntrTrans.t19 : idleNoHeat;
      exe.roomIntrTrans.t20 : noState;
      1 : iss.CS.noHeatReq_state;
    esac;
    next(pss.CS.heatReq_state):=case
      exe.heatReq.t21 : waitForCool;
      ...
      exe.roomIntrTrans.t19 : noState;
      exe.roomIntrTrans.t20 : idleHeat;
      1 : iss.CS.heatReq_state;
    esac;
    ...
```

Figure 3.66: NextCS Sub-module Updated for Interrupt Transitions

3.8.2 NextCSa Sub-module

Sub-module `nextCSa` implements the predicate *next_CSa(*ss, τ, CSa'*)*. The parameter value $CSa' = \phi$ means that if any transition in an HTS executes, the state variable of this HTS in the auxiliary element CSa' is reset to `noState`; otherwise, it keeps the same value as in the reset snapshot `iss`.

```
MODULE nextCSa(pss,iss,exe)
  ASSIGN
    next(pss.CSa.furnace_state) := case
      exe.furnace.t1 : noState;
      exe.furnace.t2 : noState;
      exe.furnace.t3 : noState;
      exe.furnace.t4 : noState;
      exe.furnace.t5 : noState;
      exe.furnace.t6 : noState;
      exe.furnace.t7 : noState;
      1 : iss.CSa.furnace_state;
    esac;
    ...
```

Figure 3.67: NextCSa Sub-module : $CSa' = \phi$ (Heating System)

Figure 3.67 shows an example of HTS *furnace* for *next_CSa* parameter value $CSa' = \phi$. If any transition in HTS *furnace* executes, the state variable, `pss.CSa.furnace_state`, is set to `noState`; otherwise, it keeps the same value as in the reset snapshot `iss`.

3.8.3 NextIE Sub-module

Sub-module `nextIE` implements the predicate *next_IE(*ss, τ, IE'*)*. In the `nextIE` sub-module, the next value of each event variable in snapshot element IE is set using a `case` statement that contains a condition for each of the transitions of the system that generates the event.

In the following, we show how to map the different values of this predicate using the simple example in Figure 3.68. In this example, events *a*, and *b* are internal events, but event *a* is an internal event used by the system only ($a \in intern\_ev(E)$), and event *b* is an internal event that can be sensed outside the system ($b \in extern\_ev(E)$).

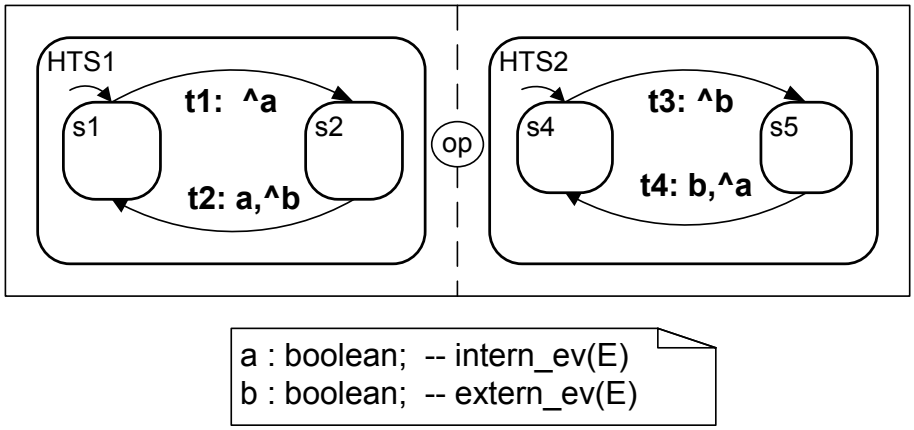


Figure 3.68: Example to Illustrate *next_IE*, and *next_O*

Parameter value $IE' = gen(\tau)$ means that in the next step, the snapshot element IE is the set of events that are generated by executing the transition τ. Figure 3.69 shows the `case` statement for events *a* and *b* for Figure 3.68, applying the semantics of parameter value $IE' = gen(\tau)$. In this example, if transition *t1* in HTS1 or transition *t4* in HTS2 executes, event *a* is generated; otherwise, this event does not occur since no transition generates it.

```
MODULE nextIE(pss,iss,exe)
  ASSIGN
    next(pss.IE.a):=case
      exe.HTS1.t1 : 1;
      exe.HTS2.t4 : 1;
      1 : 0;
    esac;
    next(pss.IE.b):=case
      exe.HTS1.t2 : 1;
      exe.HTS2.t3 : 1;
      1 : 0;
    esac;
```

Figure 3.69: NextIE Sub-module: $IE' = gen(\tau)$ for Figure 3.68

Parameter value $IE' = gen(\tau) \cap intern\_ev(E)$ is used for RSML. In RSML, the snapshot element IE is the set of events generated by executing τ that are used only by the system, so

event *a* may belong to snapshot element IE, but event *b* may not. Figure 3.70 shows the `nextIE` sub-module for the example in Figure 3.68 with this semantics. It has a case statement only for internal event *a*. The events that belong to $intern\_ev(E)$ and $extern\_ev(E)$ can be determined syntactically, so the translator uses this information to produce the appropriate SMV code.

```
MODULE nextIE(pss,iss,exe)
  ASSIGN
    next(pss.IE.a):=case
      exe.HTS1.t1 : 1;
      exe.HTS2.t4 : 1;
      1 : 0;
    esac;
```

Figure 3.70: NextIE Sub-module : $IE' = gen(\tau) \cap intern\_ev(E)$ for Figure 3.68

Figure 3.71 shows the `nextIE` sub-module for the example in Figure 3.68, applying parameter value $IE' = ss.IE \cup gen(\tau)$, which is used for statecharts where internally generated events persist throughout a macro-step. For example, if either transition *t1* or transition *t4* executes, the event occurs; if the event *a* already exists (`iss.IE.a` is true), the event is maintained; otherwise, it does not occur in the next snapshot. Since statecharts does not distinguish between internal and external events, *a* and *b* are both set in the `nextIE` sub-module.

```
MODULE nextIE(pss,iss,exe)
  ASSIGN
    next(pss.IE.a):=case
      exe.HTS1.t1 : 1;
      exe.HTS2.t4 : 1;
      1 : iss.IE.a;
    esac;
    next(pss.IE.b):=case
      exe.HTS1.t2 : 1;
      exe.HTS2.t3 : 1;
      1 : iss.IE.b;
    esac;
```

Figure 3.71: NextIE Sub-module : $IE' = ss.IE \cup gen(\tau)$ for Figure 3.68

Figure 3.72 shows the `nextIE` sub-module for the example in Figure 3.68, applying param-

eter value $IE' = (ss.IE - trig(\tau)) \cup gen(\tau)$. This pre-defined parameter value indicates that an internal event, when generated, persists in a macro-step until the event is used to trigger a transition. In this example, if either transition *t1* or *t4* executes, the event *a* occurs; if *t2* executes, because event *a* is a triggering event of transition *t2*, the event is consumed, and thus does not occur in next step; if it already exists (`iss.IE.a` is true), the event persists; otherwise, it does not occur in the next snapshot.

```
MODULE nextIE(pss,iss,exe)
  ASSIGN
    next(pss.IE.a):=case
      exe.HTS1.t1 : 1;
      exe.HTS2.t4 : 1;
      exe.HTS1.t2 : 0;
      1 : iss.IE.a;
    esac;
    next(pss.IE.b):=case
      exe.HTS1.t2 : 1;
      exe.HTS2.t3 : 1;
      exe.HTS2.t4 : 0
      1 : iss.IE.b;
    esac;
```

Figure 3.72: NextIE Sub-module : $IE' = ((ss.IE - trig(\tau)) \cup gen(\tau))$ for Figure 3.68

3.8.4 NextIEa Sub-module

Sub-module `nextIEa` implements the predicate *next_IEa(*ss, τ, IEa'*)*. Currently, none of the supported notations use snapshot element IEa, and the only parameter value is n/a.

3.8.5 NextO Sub-module

Sub-module `nextO` implements the predicate *next_O(*ss, τ, O'*)*. In the `nextO` sub-module, similar to the `nextIE` sub-module, the next value of each event variable in snapshot element O is set using a `case` statement that contains a condition for each of the transitions of the system

that generate the event. We continue to use Figure 3.68 to illustrate our mapping of the different parameter values in predicate *next_O*.

Parameter value $O' = gen(\tau)$ means that in the next step, the snapshot element O is the set of events that are generated by executing the transition τ. It indicates that a generated event can be sensed by the environment of the system only at the micro-step after it is generated. Figure 3.73 shows the `nextO` sub-module for the example in Figure 3.68 using this parameter value. In this example, if transition *t1* in HTS1 or transition *t4* in HTS2 executes, it generates output event *a*, otherwise, the output event *a* does not occur since no transition generates it.

```
MODULE nextO(pss,iss,exe)
  ASSIGN
    next(pss.O.a):=case
      exe.HTS1.t1 : 1;
      exe.HTS2.t4 : 1;
      1 : 0;
    esac;
    next(pss.O.b):=case
      exe.HTS1.t2 : 1;
      exe.HTS2.t3 : 1;
      1 : 0;
    esac;
```

Figure 3.73: NextO Sub-module : $O' = gen(\tau)$ for Figure 3.68

Parameter value $O' = ss.O \cup gen(\tau)$ means that in the next step, the snapshot element O is the union of the original events in O and the event(s) generated by executing the transition τ. It indicates that a generated output event persists throughout a macro-step. Figure 3.74 shows the `nextO` sub-module for Figure 3.68 for this parameter value. If either transition *t1* or transition *t4* executes, the output event *a* occurs; if it already exists (`iss.O.a` is true), it is maintained; otherwise, it does not occur in the next snapshot.

```
MODULE nextO(pss,iss,exe)
  ASSIGN
    next(pss.O.a):=case
      exe.HTS1.t1 : 1;
      exe.HTS2.t4 : 1;
      1 : iss.O.a;
    esac;
    next(pss.O.b):=case
      exe.HTS1.t2 : 1;
      exe.HTS2.t3 : 1;
      1 : iss.O.b;
    esac;
```

Figure 3.74: NextO Sub-module : $O' = ss.O \cup gen(\tau)$ for Figure 3.68

Parameter value $O' = ss.O \cup (gen(\tau) \cap extern\_ev(E))$ is used for RSML, where the snapshot element O is the set of events that are generated by executing τ and belong to the set of external events ($extern\_ev$). The generated outputs persist throughout the micro-step. Figure 3.75 shows the `nextO` sub-module for Figure 3.68. Compared to the `nextIE` sub-module (Figure 3.70) for this example, in which only the internal event *a* gets updated, the `nextO` sub-module only has a case statement for external event *b*, which is generated when either transition *t2* in HTS1 or transition *t3* in HTS2 executes.

```
MODULE nextO(pss,iss,exe)
  ASSIGN
    next(pss.O.b):=case
      exe.HTS1.t2 : 1;
      exe.HTS2.t3 : 1;
      1 : iss.O.b;
    esac;
```

Figure 3.75: NextO Sub-module : $O' = ss.O \cup gen(\tau) \cap extern\_ev(E)$ for Figure 3.68

3.8.6 NextIa Sub-module

Sub-module `nextIa` implements the predicate *next_Ia(*ss, τ, Ia'*)*. In the following, we show how to map the different values of the template predicate *next_Ia* for the simple example in Figure 3.76. In this example, events *a*, and *b* are environment events that can be generated also by the system.

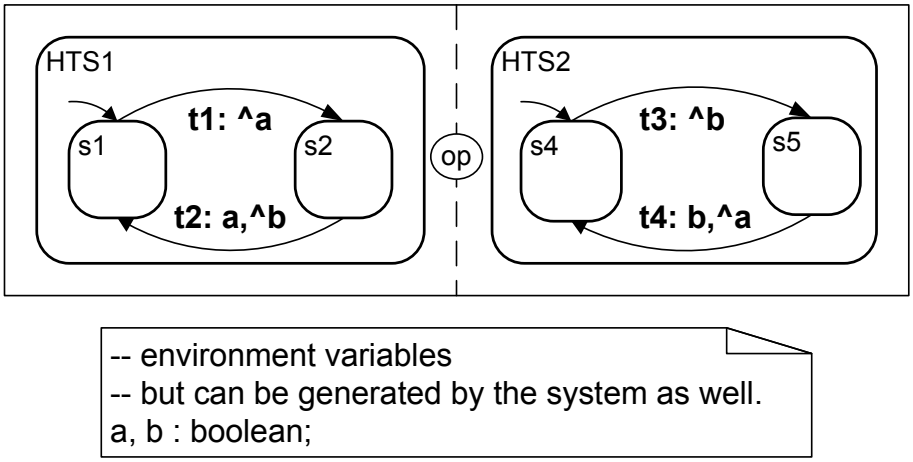


Figure 3.76: Example to Illustrate *next_Ia*

Parameter value $true$ is used for notations with simple macro-semantics, where we don't care about the next value of the auxiliary variable since we read the inputs in each step. The `nextIa` sub-module for this parameter value is empty, i.e., the next values for Ia elements could be any values.

Parameter value $Ia' = \phi$ means that in the next step, the auxiliary variables for the inputs in snapshot Ia are cleared. Figure 3.77 shows the `nextIa` sub-module for Figure 3.76, applying this semantics.

```
MODULE nextIa(pss,iss,exe)
  ASSIGN
    next(pss.Ia.a) := 0;
    next(pss.Ia.b) := 0;
```

Figure 3.77: NextIa Sub-module : $Ia' = \phi$ for Figure 3.76

Figure 3.78 shows the `nextIa` sub-module for Figure 3.76, applying the parameter value $Ia' = ss.Ia$, which means that input events which are read at the start of a macro-step are valid throughout the macro-step.

```
MODULE nextIa(pss,iss,exe)
  ASSIGN
    next(pss.Ia.a) := iss.Ia.a;
    next(pss.Ia.b) := iss.Ia.b;
```

Figure 3.78: NextIa Sub-module : $Ia' = ss.Ia$ for Figure 3.76

Parameter value $Ia' = ss.Ia \cup gen(\tau)$ indicates that in the next snapshot, the snapshot element Ia is the union of the original events in Ia and the events generated by executing the transition τ. This parameter value is used by CCS and CSP, which does not distinguish between input and internal events, and does not use snapshot element IE, therefore, the generated events are added to Ia directly, and these events persist throughout a macro-step. Figure 3.79 shows the corresponding `nextIa` sub-module for Figure 3.76. For example, if transition *t1* or transition *t4* executes, the event is generated; if the event existed in the previous snapshot, it still exists; otherwise, it keeps the value 0.

```
MODULE nextIa(pss,iss,exe)
  ASSIGN
    next(pss.Ia.a) := case
      exe.HTS1.t1 : 1;
      exe.HTS2.t4 : 1;
      1 : iss.Ia.a;
    esac;
    next(pss.Ia.b) := case
      exe.HTS1.t2 : 1;
      exe.HTS2.t3 : 1;
      1 : iss.Ia.b;
    esac;
```

Figure 3.79: nextIa Sub-module : $Ia' = ss.Ia \cup gen(\tau)$ for Figure 3.76

Figure 3.80 shows the `nextIa` module for Figure 3.76, applying the *next_Ia* parameter value $Ia' = ((Ia - trig(\tau)) \cup gen(\tau))$. This pre-defined parameter value also does not distinguish

between input and internal events. It differs from the previous parameter value in that an event, if used to trigger a transition, no longer persists. In Figure 3.80, when transition *t2* executes, the transition consumes event *a*, thus the corresponding event variable is reset; and when transition *t4* executes, the event variable for *b* is reset.

```
MODULE nextIa(pss,iss,exe)
  ASSIGN
    next(pss.Ia.a) := case
      exe.HTS1.t1 : 1;
      exe.HTS2.t4 : 1;
      exe.HTS1.t2 : 0;
      1 : iss.Ia.a;
    esac;
    next(pss.Ia.b) := case
      exe.HTS1.t2 : 1;
      exe.HTS2.t3 : 1;
      exe.HTS2.t4 : 0;
      1 : iss.Ia.b;
    esac;
```

Figure 3.80: NextIa Sub-module : $Ia' = ((Ia - trig(\tau)) \cup gen(\tau))$ for Figure 3.76

3.8.7 NextAV Sub-module

Sub-module `nextAV` implements the predicate *next_AV(*ss, τ, AV'*)*. In the following, we first show how predicate *resolve_conflicts* may affect the `nextAV` sub-module, then we describe how to map the different values of the predicate *next_AV* (shown in Table 3.5, on page 74) to SMV using the simple example in Figure 3.81, where both variables *x* and *y* are variables of integer range type.

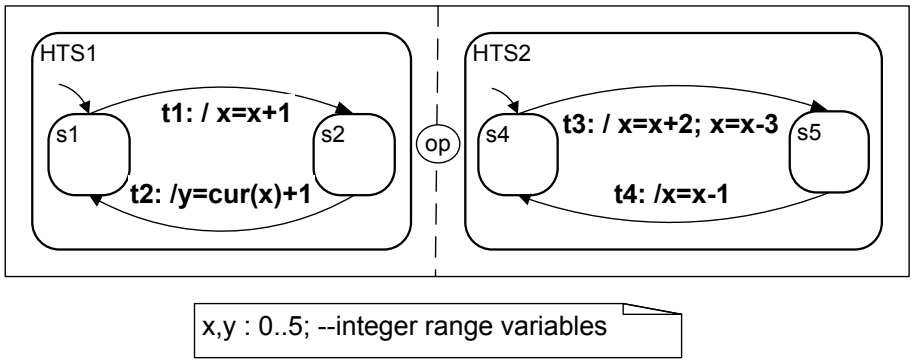


Figure 3.81: Example to Illustrate *next_AV* and *next_AVa*

For most notations, multiple variable assignments to the same variable in different components are not allowed in a system's specification. In the `nextAV` sub-module for these notations, the next value of each variable in snapshot element AV is updated using a `case` statement that contains a condition for each transition of the system that updates this variable. Figure 3.82 shows the next value assignment for variable x for a notation without multiple variable assignments: in each step, if a transition executes (e.g., `exe.HTS1.t1`), the variable x is assigned the result of the transition's assignment (e.g., macro `xt1`, which that will be explained shortly).

```
MODULE nextAV(pss,iss,exe)
  ASSIGN
    next(pss.AV.x):=case
      exe.HTS1.t1 : xt1;
      exe.HTS2.t3 : xt3;
      exe.HTS2.t4 : xt4;
      1 : iss.AV.x;
    esac;
  DEFINE
    --macros to carry the result for x of executing t1
    xt1 := ...;
    ...
```

Figure 3.82: Next Value Assignment for Notations without Multiple Variable Assignments

Some notations, such as STATEMATE, allow multiple variable assignments to the same variable in a micro-step. Template semantics provides a template predicate *resolve_conflicts* that specifies how to resolve such conflicts, which affects the result of the `nextAV` sub-module.

Figure 3.83 shows the next value assignment for variable x using predicate *resolve_conflicts* parameter value *resolve*$_{STM}$, which specifies that if multiple variable assignments happen on the same variable at the same step, one result is assigned to the variable nondeterministically. In the `nextAV` sub-module, the next value of each variable in snapshot element AV is updated using a `case` statement that contains a condition for each possible combination of transitions of the system that updates this variable. For example, if HTS1 executes *t1* and HTS2 executes *t3*, the next value of variable x is assigned nondeterministically to either the result of executing *t1* (e.g., macro `xt1`), or the result of executing *t3* (e.g., macro `xt3`); if only one transition executes that updates the variable, the next value is assigned to the result of executing that transition; otherwise, the variable is unchanged from the variable's value in the reset snapshot `iss`.

```
MODULE nextAV(pss,iss,exe)
  next(pss.AV.x):=case
    exe.HTS1.t1 & exe.HTS2.t3 : {xt1, xt3};
    exe.HTS1.t1 & exe.HTS2.t4 : {xt1, xt4};
    exe.HTS1.t1 : xt1;
    exe.HTS2.t3 : xt3;
    exe.HTS2.t4 : xt4;
    1 : iss.AV.x;
  esac;
  DEFINE
    --macros to carry the result of executing t1
    xt1 := ...;
    ...
```

Figure 3.83: Notations with variable resolve conflicts

In our translator, we perform static analysis to find the transitions for each HTS that may affect the value of each variable, and enumerate all possible combinations. However, some of the combinations may never be true, for example, if interleaving composition is used in Figure 3.81, *t1* and *t3* are not allowed to execute together. Although our translation does not eliminate these combinations, it does not affect the result of the variable assignment since these combinations never become true.

For the `nextAV` sub-module, it is possible that overflow or underflow happens for integer range variables. For example, in Figure 3.81, where the execution of transition *t1* might cause variable x to overflow, and the execution of transition *t4* might cause it to underflow. NuSMV

reports a syntax error for each possible overflow or underflow error for integer range variables. However, Cadence SMV only reports a warning message for overflow or underflow problem. To allows both NuSMV and Cadence SMV to check the generated SMV code, we handle the overflow or underflow problem explicitly in our SMV code. We introduce two macros for each transition (e.g., *t1*) that updates each integer range variable (e.g., *x*). One macro (e.g., `xt1`) captures the value of the variable assignment when this transition executes: if no underflow or overflow will occur, the variable is assigned the value resulting from executing the transition, otherwise, it keeps its original value. A second macro (e.g., `xt1error`) indicates whether underflow or overflow happens when a transition executes.

In addition, the variable `error_variables` [15] catches any overflow or underflow errors of the system; it is the disjunction of all macros that indicate whether underflow or overflow happen, and the previous value of this variable. If any transition causes any variable to either underflow or overflow (e.g., `xt1error` or `yt2error`, etc.), this variable is set to 1, and it will stay 1 forever. We can check whether the specification has underflow or overflow errors by checking the CTL property `AG !(error_variables)`.

Next, we describe the translation of each *next_AV* parameter value in Table 3.5. The parameter values change only the definition of the macros, so we do not show the next statements for the variables.

Figure 3.84 shows how to define the transition macros (e.g., `xt1`, and `xt1error`) and the `error_variable` in the sub-module `nextAV` for the composed HTS in Figure 3.81 with STATEMATE semantics (choice 2). In STATEMATE, a transition can make multiple assignments to the same variable; however, only the last assignment to the variable has an effect (not the accumulation of previous assignments on the same transition), and the assignment expressions are evaluated with respect to the current variable values in AV. Therefore, when transition *t3* executes, the value of variable x is updated by the last assignment of this transition ($x = x-3$), and macros `xt3` and `xt3error` are with respect to only the last assignment to x by *t3*,

[15] This variable is initially set to 0 in `initss` module.

```
MODULE nextAV(pss,iss,exe)
 DEFINE
  -- x = x + 1, x : 0..5
  xt1 := case
   ((iss.AV.x + 1)>=0)&((iss.AV.x + 1)<=5) : (iss.AV.x + 1);
   1 : x;
  esac;
  xt1error := exe.HTS1.t1&(((iss.AV.x + 1)>=0)|((iss.AV.x + 1)<=5));

  -- x = x + 2; x = x - 3, x : 0..5
  xt3 := case
   ((iss.AV.x - 3)>=0)&((iss.AV.x - 3)<=5) : (iss.AV.x - 3);
   1 : x;
  esac;
  xt3error := exe.HTS2.t3&(((iss.AV.x - 3)<0)|((iss.AV.x - 3)>5));
  ...
 ASSIGN
  next(pss.AV.error_variables):=   iss.AV.error_variables
                         | xt1error | xt3error | xt4error
                         | yt2error;
```

Figure 3.84: NextAV Sub-module : Choice 2 for Figure 3.81

Template parameter value choice 3 is used for notations that permit only one assignment to a variable on a transition, such as BTS and RSML; if there are multiple assignments on the same transition, only the first assignment takes effect. Figure 3.85 shows the corresponding definitions of transition macros in the `nextAV` sub-module for the example in Figure 3.81.

```
MODULE nextAV(pss,iss,exe)
 DEFINE
   -- x = x + 2; x = x - 3, x : 0..5
  xt3 := case
   ((iss.AV.x + 2)>=0)&((iss.AV.x + 2)<=5) : (iss.AV.x + 2);
   1 : x;
  esac;
  xt3error := exe.HTS2.t3&(((iss.AV.x + 2)<0)|((iss.AV.x + 2)>5));
  ...
```

Figure 3.85: NextAV Sub-module : Choice 3 for Figure 3.81

```
MODULE nextAV(pss,iss,exe)
 DEFINE
  -- y = cur(x) + 1, y : 0..5, w.r.t. AV
  yt2 := case
   ((iss.AV.x + 1)>=0)&((iss.AV.x + 1)<=5) : (iss.AV.x + 1);
   1 : x;
  esac;
  yt2error := exe.HTS1.t2&(((iss.AV.x + 1)<0)|((iss.AV.x + 1)>5));

   -- x = x + 2; x = x -3, x : 0..5, w.r.t. AVa
  xt3 := case
   ((iss.AVa.x + 2)>=0)&((iss.AVa.x + 2)<=5) : (iss.AVa.x + 2);
   1 : x;
  esac;
  xt3error := exe.HTS2.t3&(((iss.AVa.x + 2)<0)|((iss.AVa.x + 2)>5));
  ...
```

Figure 3.86: NextAV Sub-module : Choice 4 for Figure 3.81

Template parameter value choice 4 is used for notations that have the `cur` syntax, such as statecharts, which indicates that when executing transition τ, if the variable assignment of transition τ uses the current variable value (e.g., `cur(x)`), the variable assignment expression is evaluated according to the current variable values (AV); otherwise, by default, the assignment expression is evaluated with respect to the auxiliary variable values in AVa. Figure 3.86 shows the corresponding definitions of transition macros in the `nextAV` module for the example in Figure 3.81. The execution of transition *t2* updates variable *y* based on the snapshot element

AV, while the execution of other transitions that update variable *x* is based on the snapshot element AVa.

Template parameter values choices 5 and choice 6 are for notations that can have multiple assignments to the same variable in the same transition. Choice 5 evaluates the variables by accumulating the changes in all variable assignments. Figure 3.87 shows the definition of macros `xt3` and `xt3error` in the `nextAV` using this semantics.

```
MODULE nextAV(pss,iss,exe)
 DEFINE
  -- x = ((x + 2) - 3), x : 0..5
  xt3 := case
   (((iss.AV.x + 2) - 3)>=0)&(((iss.AV.x + 2) - 3)<=5)
             : ((iss.AV.x +2) -3);
   1 : x;
  esac;
  xt3error := exe.HTS2.t3
             &((((iss.AV.x + 2) - 3)<0)|(((iss.AV.x + 2) - 3)>5));
  ...
```

Figure 3.87: NextAV Sub-module : Choice 5 for Figure 3.81

Choice 6 specifies that one assignment to the variable is chosen nondeterministically. Figure 3.88 shows the definitions of transition macros in the sub-module `nextAV` using this semantics. It defines two macros (e.g., `xt3c1, xt3c1error`) for each choice of the assignments in a transition. When transition *t3* executes, the value of variable *x* is updated using SMV's nondeterministic assignment. The `error_variables` is set to true if any action that could execute would underflow or overflow[16].

[16]Ideally, macro `error_variables` should only be set to true if the chosen variable assignment underfbws or overfbws. In Express, we have set the macro to be true if any of the variable assignments underfbw or overfbw, which is a limitation of our translator.

```
MODULE nextAV(pss,iss,exe)
 DEFINE
  -- x = x + 2
  xt3c1 := case
   ((iss.AV.x + 2)>=0)&((iss.AV.x + 2)<=5) : (iss.AV.x + 2);
   1 : x;
  esac;
  xt3c1error := exe.HTS2.t3&(((iss.AV.x + 2)<0)|((iss.AV.x + 2)>5));

  -- x = x - 3
  xt3c2 := case
   ((iss.AV.x - 3)>=0)&((iss.AV.x - 3)<=5) : (iss.AV.x-3);
   1 : x;
  esac;
  xt3c2error := exe.HTS2.t3&(((iss.AV.x - 3)<0)|((iss.AV.x - 3)>5));
  ...
 ASSIGN
  next(pss.AV.x):=case
    exe.HTS1.t1 : xt1;
    exe.HTS2.t3 : {xt3c1, xt3c2};
    exe.HTS2.t4 : xt4;
    1 : iss.AV.x;
  esac;
ASSIGN
 next(pss.AV.error_variables):=   iss.AV.error_variables
                          | xt1error | xt3c1error | xt3c2error
                          | xt4error | yt2error;
```

Figure 3.88: NextAV Sub-module : Choice 6 for Figure 3.81

3.8.8 NextAVa Sub-module

Sub-module `nextAVa` implements the predicate *next_AVa(*ss, τ, AVa'*)*. The parameter value $AVa' = ss.AVa$ means that in the next step, the snapshot element AVa keeps the value from the previous snapshot. Figure 3.89 shows the `nextAVa` sub-module for the example in Figure 3.81, applying the parameter value $AVa' = ss.AVa$.

```
MODULE nextAVa(pss,iss,exe)
  ASSIGN
    next(pss.AVa.x):=iss.AVa.x;
    next(pss.AVa.y):=iss.AVa.y;
```

Figure 3.89: NextAVa Sub-module : $AVa' = ss.AVa$ for Figure 3.81

3.9 Summary

In this chapter, we showed our method for translating a composed hierarchical transition system into a collection of SMV modules. We explained our translation method in terms of the SMV model produced for input template parameters and a specification. The implementation of our translator will be described in the next chapter.

Chapter 4

Implementation

In this chapter, we show the architecture of our translator, and provide an overview of the algorithm used.

4.1 Architecture of Express

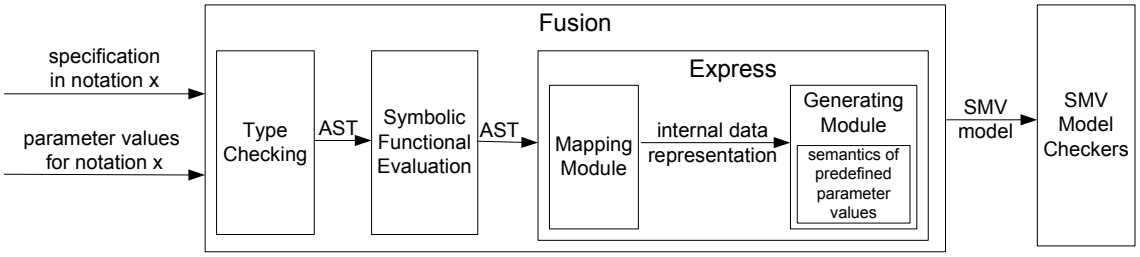


Figure 4.1: Architecture of Express

Figure 4.1 shows the architecture of our translator, Express. It is an extension of an existing tool called Fusion, which is written in the C language. It takes as input a specification of a notation, and a set of template parameter values encoding the notation's semantics. It uses Fusion's type checking and symbolic functional evaluation to generate two syntax trees, one for the specification and one for the parameter values. Thereafter, it reads the ASTs, generates internal data structures, and combines these internal data structures with pre-defined semantics of the

chosen parameter values, which are hard-coded in the translator, to generate an SMV model of the specification. Next, we briefly explain these steps.

The inputs to our translator are written in S+, Fusion's input language, which is based on the higher-order logic of the HOL theorem proving system [16]. S+ includes constructs for the declaration and definition of types and constants. In order to model a specification in template semantics written in S+, we declare types for states, events, variables, transitions, and HTSs. We can model a specification in template semantics as definitions for each of the parts, and compose them together to form an HTS. When modelling a composed hierarchical transition system, we model each HTS separately, and use composition operators to compose them together.

Fusion parses input written in S+, type checks the specification, and generates abstract syntax trees for each definition in the specification. Fusion implements a technique called symbolic functional evaluation [14] (SFE), which evaluates expressions by expanding definitions. SFE can be used to expand the definitions (including macros[1]) of a model into a single abstract syntax tree, which contains all the information for the specification. Using SFE, we can provide parameterized definitions, and let SFE expand the definitions using the appropriate parameter values.

Express automatically translates from the AST of the specification to an SMV model. It has two main modules: `Mapping` module and `Generating` module. The `Mapping` module parses the ASTs and records information using our own internal data structures. Thereafter, based on the template parameter values, the `Generating` module generates the corresponding SMV code and prints it to a file.

We want our SMV code to match the modularity of template semantics to verify the correctness of our translation, so the translation algorithm is based on the form of the template. Because each template parameter defines one aspect of a snapshot element in template semantics (i.e., *reset_AV* defines how to reset snapshot element AV at the start of each macro-step), it is not necessary to know the whole specification to generate the corresponding SMV code for this parameter. Furthermore, passing the AST for the whole specification to check the needed information for one parameter is not efficient. Therefore, it is worthwhile to parse the AST once and store the specification into individual units to speed up and facilitate the translation. In Express, there is a data structure to record the parameter value for each template parameter, and to

[1]For example, macro `tooCold`, which is defined in Figure 2.1 on page 8, is expanded by SFE.

represent snapshot elements.

4.2 High-Level Description of Algorithm

Figure 4.2 shows the high-level algorithm of the `Mapping` module. It maps information for template parameters, variables, input events, internal events (both *extern_ev*, and *intern_ev*), variables' initial value(s), HTSs, and composed HTSs. Finally, it checks whether the specification has environmental synchronization or rendezvous synchronization, and finds the set of synchronization events, which will be used to set enabled and execution macros for each participating HTS.

```
/*C functions for mapping module*/
/*map AST to internal data*/
Map_Tspara;                /*template parameters*/
Map_Variables;
Map_InputEvents;
Map_ExternalEvents;
Map_InternalEvents;
Map_VariableInitialValues;
Map_HTSs;
Map_HTSCompositions;

/*if exists, get the synchronization events of the system*/
Get_EnvironmentSyncEvents;
Get_RendSyncEvents;
```

Figure 4.2: High-Level Sequence for Mapping Module

In order to match the modularity of template semantics, our translator has one C function (e.g., `Gen_ModuleReset`) for each template definition (e.g., *reset*) to generate the corresponding SMV module (e.g., `reset`). It also has one C function (e.g., `Gen_ModuleResetAV`) for each template parameter (e.g., *reset_AV*) to generate the corresponding SMV sub-module (e.g., `resetAV`). The translator only recognizes a predefined set of parameter values for each template parameter, and has hard-coded these parameter values in its corresponding function. Our translator also has a C function (e.g., `Gen_ModuleParallel`) for each composition operator

(e.g., *parallel*) in template semantics, which generates the operator-related SMV module (e.g., `parallel`).

```
/*C functions for generating module*/
/*generate SMV snapshot module, and its sub-modules*/
Gen_ModuleStates;
Gen_ModuleEnvEvents;
Gen_ModuleIaEvents;
Gen_ModuleIntEvents;
Gen_ModuleEnvVariables;
Gen_ModuleInputs;
Gen_ModuleOutputs;
Gen_ModuleSnapshot;

/*generate the enabled and execute modules for all HTSs*/
Gen_HTSEnabled;
Gen_HTSExecute;

/*generate the modules for composition operators*/
Gen_ModuleOperators;

/*generate the modules for common template definitions*/
Gen_ModuleReset;
Gen_ModuleEnabled;
Gen_ModuleExecute;
Gen_ModuleApply;

Gen_ModuleInit;  /*initialize the snapshot elements*/
Gen_ModuleMain;  /*generate the SMV main module*/
Free_InternalData; /*free the internal data structures*/
```

Figure 4.3: High-Level Sequence for Generating Module

Figure 4.3 shows the high-level sequence of the `Generating` module. After declaring the snapshot module and its sub-modules, it defines enabled and execute sub-modules for each HTS in the specification, and the SMV modules for each type of composition operator in the specification. Thereafter, it generates the template-related functions, according to the provided

parameter values; generates the `initss` module and the `main` module of the SMV model; and frees the internal data structures.

From our experiments, Express seems to be an efficient translator. Because we use internal data structures to represent the specification, for each C function of different template parameters, only related internal data structures are walked over to generate the corresponding SMV code. Our translator may walk over the internal data structures multiple times to satisfy different composition and parameter values. However, the time for translation is negligible in comparison to the time for model checking.

Express is also easy to extend to incorprate the future evolution of template semantics. When a new template parameter or a new composition operator is added to template semantics, we need to add a corresponding C function. Adding extra parameter values to a template parameter only requires us to add some code in the corresponding function. Therefore, the work to update the translator is relatively straightforward.

4.3 Summary

In this chapter, we briefly described the architecture of our translator, and provided a high-level description of how an SMV model is generated.

Chapter 5

Evaluation

In this chapter, we discuss the evaluation of our work using the heating system that we introduced in Chapter 2, and a single lane bridge, described in this chapter. Then, we show the statistics on our case studies.

5.1 Case Study : Single Lane Bridge

The single lane bridge specification [26] models two cars travelling in two directions over a single-lane bridge. Cars travelling in different directions cannot be on the bridge at the same time. Cars travelling in the same direction can be on the bridge together, but they cannot pass each other. To ease our presentation, cars travelling in one direction are designated as red cars, and cars travelling in the other direction are blue cars.

Figure 5.1 declares the variables needed for the specification of the single lane bridge system. The single lane bridge keeps track of the number of red cars and blue cars on the bridge using variables *numRed* and *numBlue*, respectively. They have the type integer range from 0 to 2, and are initialized to 0. The system has eight environmental events. For example, event *entRedA* means a red car enters the bridge, and event *exitRedA* means a red car exits the bridge. The system has four internal events that are generated by the executing transitions. For example, when a red car enters the bridge, the system generates an *inRed* event to indicate that the bridge allows red cars to go through.

In this specification, each car is modelled by an HTS with four states that indicate a car

```
/*variables*/
var numRed : range(0,2) = 0;
var numBlue : range(0,2) = 0;

/*environment events : type env*/
event entRedA, exitRedA, entRedB, exitRedB, entBlueA, exitBlueA, entBlueB, exitBlueB : env;

/*internal events : type intee*/
event inRed; outRed; inBlue; outBlue : intee;
```

Figure 5.1: Single Lane Bridge System Variable Declarations

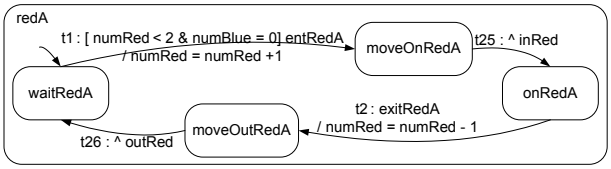


Figure 5.2: Red Car HTSs

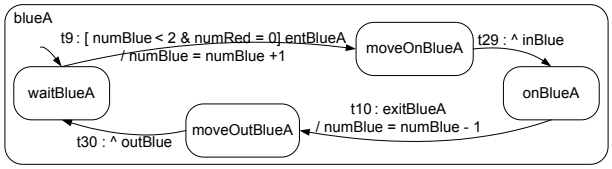


Figure 5.3: Blue Car HTSs

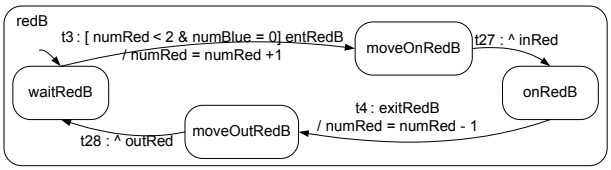


Figure 5.4: Red Car Coordinator HTSs

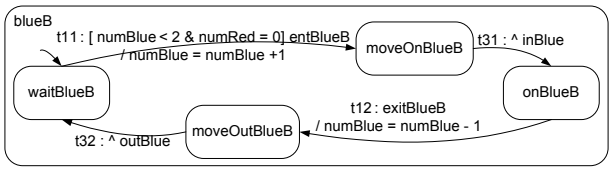


Figure 5.5: Blue Car Coordinator HTSs

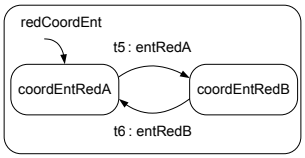


Figure 5.6: Bridge HTS

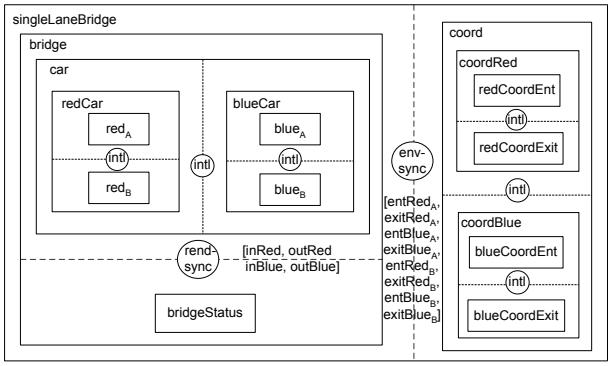


Figure 5.7: Single Lane Bridge

waiting to move onto the bridge, moving onto the bridge, being on the bridge, and moving out of the bridge. There are two cars of each colour (Figure 5.2 and Figure 5.3). For each colour, one coordinator HTS ensures that cars of that colour take turns entering the bridge, and another coordinator HTS ensures that cars of that colour exit the bridge in the order that they entered the bridge (Figure 5.4 and Figure 5.5). The bridge is modelled by an HTS with five states that indicate the number and colours of cars on the bridge (Figure 5.6).

Figure 5.7 shows the composition of the single lane bridge HTSs. It has three kinds of composition: interleaving, environmental synchronization, and rendezvous synchronization. The four car HTSs are interleaved to form composite component *car*. Component *car* and HTS *bridgeStatus* rendezvous on events *inRed, inBlue, outRed,* and *outBlue,* which communicate when a red or blue car enters or exits the bridge. The four coordinators are interleaved to form component *coord,* which synchronizes with component *bridge* on environmental events such as *entRedA, entRedB, exitRedA,* and *exitRedB.*

Parameter		CCS with variables	
		$resetXX(ss, I)$	$nextXX(ss, \tau)$
states	$CS =$	$ss.CS$	$entered(dest(\tau))$
	en_states	$src(\tau) \subseteq ss.CS$	
events	$IE =$	n/a	
	$I_a =$	$I.ev$	$ss.I_a \cup gen(\tau)$
	en_events	$trig(\tau) \subseteq ss.I_a$	
	$O =$	$\emptyset$	$gen(\tau)$
variables	$AV =$	$assign(ss.AV, I.var)$	$assign(ss.AV, eval(ss.AV, asn(\tau)))$
	en_cond	$ss.AV \models cond(\tau)$	
macro-semantics		simple diligent	
pri		none	
resolve_conflicts		n/a	

Table 5.1: Parameter Values for CCS with Variables

We use the semantics of CCS with shared variables for this example. Table 5.1 shows the parameter values for this variation of CCS notation. It differs from CCS in that it has snapshot element AV and its related parameter values to reflect the semantics of variables. It is also different from data-passing CCS [29], which allows internal events to carry data parameters.

5.2 Results

To validate our translator, we have inspected and tested our translation on every template-parameter value and every composition operator. We have not exercised every combination of parameter values and composition operators. Our validation is based on the assumption that the template parameters and composition operators are all separate concerns. Adding another parameter value or composition operator does not usually affect the behaviour of the others. Using interrupt composition causes changes to some of the `nextXX` sub-modules of `apply`, because the interrupt transitions may perform actions; but these changes simply add branches to conditional assignments, and do not affect existing assignments. Using rendezvous composition causes changes in how rendezvous transitions are enabled: macros are added for rendezvous events, and the enabled-event tests for rendezvous transitions are overwritten, but these changes do not affect the module's other assignments.

We use the heating system and the single lane bridge as examples to demonstrate our translation. They were chosen because they are specified in different notations, STATEMATE statecharts and CCS with variables, and because they use an extensive range of composition operators.

The automatically generated SMV model for these two case studies can be found in Appendices A and B. The generated SMV code matches the modularity of template semantics and its composition operators. Therefore, the translator keeps not only the structure of template semantics, but also the structure of the original specification.

An SMV step matches the definition of a micro-step, so no intermediate execution steps are introduced by our translator.

In the translation, we prefix names of variables, of events, of states, and of transitions with snapshot information without changing their meaning. Therefore, properties to be checked need to be prefixed, and the counterexamples for the translated specification contain prefixes. However, these changes are straightforward, and can be easily understood by users. David Fung, an undergraduate research assistant, created a tool to prefix automatically names in the properties being checked, and a tool to remove automatically prefixes from names in the counterexample generated by SMV. Thereafter, from the user's point of view, the properties and the counterexamples are with respect to the original specification.

Table 5.2 shows, by snapshot element, how the size of the SMV model resulting from our translation compares with the original specification. The basic states, internal and output events,

and variables of the specification have corresponding SMV variables in CS, IE, O, and AV, respectively. There is one extra boolean variable in AV to catch variable underflow and overflow problems. When used, the auxiliary snapshot elements, CSa, IEa, Ia, and AVa, contribute to the state space as appropriate for their parameter values. The input events $I.ev$ and variables $I.var$ also contribute to the state space when used in a specification. There is also one variable per HTS to represent which transition is chosen to execute. The only variable added for the composition operators is one for each choice composition to record the choice made between components. Therefore, the state space of our translation are comparable to the state space of the specification. Because rendezvous events are used within the step in which they are generated, we can describe their behaviour using only macros and no variables.

Snapshot Element	SMV Variables	
	Worst Case	STATEMATE, CCS with variables
CS	1 enumerated $(b+1)$ values	1 enumerated $(b+1)$ values
CSa	$b+s$ boolean	n/a
IE	i boolean	i boolean
IEa	i boolean	n/a
Ia	e boolean	e boolean
O	i boolean	i boolean
AV	$v+1$ typed	$v+1$ typed
AVa	v typed	n/a
$I.ev$	e boolean	e boolean
$I.var$	u typed	u typed
transitions (per HTS)	1 enumerated $(t+1)$ values	1 enumerated $(t+1)$ values

Table 5.2: SMV Model Size for Specification with i internal events, e input events, v variables (including u input variables), and (per HTS) b basic states, s super-states, and t transitions

Table 5.3 shows some statistics for our case studies. The count of event-related NuSMV boolean variables reflects the fact that we do not need variables for rendezvous events. The size of the state spaces for the heating system were calculated by NuSMV. The size of the state spaces

for the single lane bridge could not be calculated because NuSMV could not build the BDD for this more complicated specification. We use Cadence SMV for the single lane bridge and it does not report these statistics.

Snapshot Element	Number of SMV Variables	
	Single Lane Bridge	Heating System
CS	9 enumerated	4 enumerated
CSa	n/a	n/a
IE	4 boolean	4 boolean
IEa	n/a	n/a
Ia	8 boolean	4 boolean
O	4 boolean	4 boolean
AV	3 typed	8 typed
AVa	n/a	n/a
$I.ev$	8 boolean	4 boolean
$I.var$	0 typed	2 typed
transitions	9 enumerated	4 enumerated
state space		7.864e+18
reachable state space		6.4056e+09

Table 5.3: Case Study Statistics

We analyzed the generated SMV single lane bridge specification in Cadence SMV using the following properties:

- A red car and a blue car cannot enter the bridge at the same time.
- Two cars of the same colour cannot enter the bridge at the same time.
- A car cannot pass another car on the bridge.
- A red car cannot enter if the blue car is on the bridge, and vice versa.
- Any car can enter and leave the bridge.
- Each car can enter and leave the bridge infinitely often.

We analyzed the generated SMV heating system specification in NuSMV using the following properties:

- If the actual temperature is too low and stays too low after a timeout, the furnace will be turned on.
- If the actual temperature is too hot and stays too hot after a timeout, the furnace will be turned off.
- The furnace will be on if a room requests heat, and off if no room requests heat.
- Whenever the furnace fails, it will not start before the user resets it.

For these properties, the SMV model resulting from our translator matched the expected behaviour of the specification.

5.3 Summary

In this chapter, we described the evaluation of our translator using two examples to show that the translated models are correct, do not introduce any intermediate steps, and retain the structure of the original specifications and template semantics with comparable state space. Therefore, our work satisfies the evaluation criteria listed in Section 1.3.

Chapter 6

Conclusion and Future Work

In this chapter, we summarize our work discussing the contributions of the thesis and current limitations, and we conclude with ideas for future work.

6.1 Summary

This thesis investigated using template semantics to parameterize the translation from a requirements notation to the input language of existing and general-purpose analysis tools, the SMV family of model checkers.

We described a fully automated translator that takes the template-semantics description of a notation's meaning, a specification in the notation, and produces an SMV model for the specification. Using our translator, we can model check specifications written in a wide range of model-based notations, such as basic transition systems (BTS) [27], CSP [19], CCS [29], basic LOTOS [21], and several statecharts [17] variants.

We showed how to model a rich set of composition operators – including rendezvous, environmental synchronization, sequence, choice, interrupt – within the fairly simple language features of the SMV input language.

By using SMV modules to represent the common semantics, template parameters, and composition operators, we not only matched the modularity of template semantics, but also made the addition of a new parameter value or composition operator have a localized effect on the existing implementation. For example, adding a new parameter value for a template parameter would

involve the change of only the parameter-related SMV module, and adding a new composition operator would involve the creation of only one new SMV module; the translation of all other SMV modules should remain unchanged.

By using macros in SMV to hold reset snapshot, we successfully avoided introducing intermediate execution steps, so that the counterexamples output by SMV would be in terms of the original specification. Also by using macros to represent enabling and executing flags of all entities of the specification, we keep the state space of the translated SMV model comparable to that of the original specification.

Our translator implements a fixed set of commonly-used parameter values and composition operators. By simply selecting different combinations of parameter values and composition operators, our translator can be used for more notations than the notations listed above.

The creation of our translation validates the claim of the authors of template semantics that using template semantics considerably reduces the effort involved in constructing notation-specific analyzers. Because template semantics allows the users to only specify the parameter values that are specific to a notation, our translation handles the common semantics of notations, and only the translation of the notation-specific template parameter values is needed. Therefore, it should be easier to construct a new notation-specific translator from a template semantics representation of a notation's semantics than from other representation of its semantics.

6.2 Limitations

At this point in time, the number of template-parameter values and composition operators that our translator supports is fixed, which limits the set of requirements notations that the translator can map to SMV. It does not yet support template-parameter values of event queues, as are found in message-passing languages such as SDL [22]. Also our translator assumes that variables in the specification are of types supported in SMV (booleans, enumerated types, finite ranges of integers). Finally, all composition operators in template semantics are described as binary operators. We also have not fully explored the combination of different composition operators.

6.3 Future Work

In this thesis, we investigated using template semantics to parameterize the translation from a model-based notation to the input language of the SMV family of model checkers. There are other existing well-used model checkers, such as Spin [20] and Bogor [34], which have different state-space representation, reduction, and exploration algorithms. Therefore, each model checker may verify one type of software model efficiently, but verify other types less efficiently. In our future work, we plan to build a model checking framework that combines different model checkers, using template semantics as the intermediate language, to model check specifications of multiple notations. We also plan to explore how template semantics can be used for determining the patterns of different types of specifications, and determine which model checker is the "best fit" for analyzing each type of specification.

Appendix A

Generated SMV Model for Heating System

The heating system specification was given in page 8.

```
MODULE states
  VAR
    noHeatReq_state : {idleNoHeat,waitForHeat,noState};
    heatReq_state : {idleHeat,waitForCool,noState};
    controller_state : {off,error,idle,actHeater,heaterRun,noState};
    furnace_state : {furnaceOff,furnaceAct,furnaceRun,furnaceErr,noState};

  --define macros for all states
  DEFINE
    in_heatingSystem := in_house | in_furnace;
    in_house := in_room | in_controller;
    in_room := in_noHeatReq | in_heatReq;
    in_noHeatReq := in_idleNoHeat | in_waitForHeat;
    in_idleNoHeat := noHeatReq_state=idleNoHeat;
    in_waitForHeat := noHeatReq_state=waitForHeat;
    in_heatReq := in_idleHeat | in_waitForCool;
    in_idleHeat := heatReq_state=idleHeat;
    in_waitForCool := heatReq_state=waitForCool;
    in_controller := in_off | in_error | in_controllerOn;
    in_off := controller_state=off;
    in_error := controller_state=error;
    in_controllerOn := in_idle | in_heaterActive;
    in_idle := controller_state=idle;
    in_heaterActive := in_actHeater | in_heaterRun;
    in_actHeater := controller_state=actHeater;
    in_heaterRun := controller_state=heaterRun;
    in_furnace := in_furnaceNormal | in_furnaceErr;
    in_furnaceNormal := in_furnaceOff | in_furnaceAct | in_furnaceRun;
```

```
    in_furnaceOff := furnace_state=furnaceOff;
    in_furnaceAct := furnace_state=furnaceAct;
    in_furnaceRun := furnace_state=furnaceRun;
    in_furnaceErr := furnace_state=furnaceErr;

MODULE envEvents
  VAR
    heatSwitchOn : boolean;
    heatSwitchOff : boolean;
    userReset : boolean;
    furnaceFault : boolean;

MODULE IaEvents
  VAR
    heatSwitchOn : boolean;
    heatSwitchOff : boolean;
    userReset : boolean;
    furnaceFault : boolean;

MODULE intEvents
  VAR
    activate : boolean;
    deactivate : boolean;
    furnaceReset : boolean;
    furnaceRunning : boolean;

MODULE variables
  VAR
    valvePos : 0..2;
    setTemp : 16..24;
    actualTemp : 10..30;
    furnaceStartup : 0..5;
    waitedForWarm : 0..5;
    waitedForCool : 0..5;
    requestHeat : boolean;
    error_variables : boolean;

MODULE envVars
  VAR
    setTemp : 16..24;
    actualTemp : 10..30;

MODULE inputs
  VAR
    ev : envEvents;
    var : envVars;

MODULE outputs
```

```
  VAR
    activate : boolean;
    deactivate : boolean;
    furnaceReset : boolean;
    furnaceRunning : boolean;

MODULE snapshot
  VAR
    CS : states;
    AV : variables;
    IE : intEvents;
    OO : outputs;
    Ia : IaEvents;

-------------------------------------------------------------------------

MODULE enabled_noHeatReq(ss)
--en_states: srcBlgssCS
--en_events: trigBlgssIEUssIa
--en_states: AVEntailCond
  --define enabled macros for transition t15
  DEFINE
    enStates_t15:=(ss.CS.in_idleNoHeat);
    enEvents_t15:= 1;
    enCond_t15:=(((ss.AV.setTemp) - (ss.AV.actualTemp)) > (2));
    t15:=(enStates_t15)&(enEvents_t15)&(enCond_t15);

  --define enabled macros for transition t16
  DEFINE
    enStates_t16:=(ss.CS.in_waitForHeat);
    enEvents_t16:= 1;
    enCond_t16:=(!(((ss.AV.setTemp) - (ss.AV.actualTemp)) > (2)));
    t16:=(enStates_t16)&(enEvents_t16)&(enCond_t16);

  --define enabled macros for transition t17
  DEFINE
    enStates_t17:=(ss.CS.in_waitForHeat);
    enEvents_t17:= 1;
    enCond_t17:=((ss.AV.waitedForWarm) < (5));
    t17:=(enStates_t17)&(enEvents_t17)&(enCond_t17);

  --define enabled macros for transition t18
  DEFINE
    enStates_t18:=(ss.CS.in_waitForHeat);
    enEvents_t18:= 1;
    enCond_t18:=(((ss.AV.waitedForWarm) = (5)) & (!((ss.AV.valvePos) = (2))));
    t18:=(enStates_t18)&(enEvents_t18)&(enCond_t18);
```

```
  DEFINE
    any := t15|t16|t17|t18;
  --noHeatReq is inside interrupt operation, add priority macro
  --priority scheme is scopeOuter
  DEFINE
    pri_t15 := 3;
    pri_t16 := 3;
    pri_t17 := 3;
    pri_t18 := 3;
    pri:= case
      t15 : pri_t15;
      t16 : pri_t16;
      t17 : pri_t17;
      t18 : pri_t18;
      1 : 100;
    esac;


MODULE execute_noHeatReq(en)
  --transition declaration
  VAR
    tran : {t15_exe, t16_exe, t17_exe, t18_exe, noTran_exe};

  --execution macros for all transitions
  DEFINE
    t15 := (tran=t15_exe);
    t16 := (tran=t16_exe);
    t17 := (tran=t17_exe);
    t18 := (tran=t18_exe);
    any := !(tran=noTran_exe);

  --define INVAR constrait to define which transition to be executed
  INVAR
     (t15 -> en.t15)
    &(t16 -> en.t16)
    &(t17 -> en.t17)
    &(t18 -> en.t18)


--------------------------------------------------------------------------------

MODULE enabled_heatReq(ss)
--en_states: srcBlgssCS
--en_events: trigBlgssIEUssIa
--en_states: AVEntailCond
  --define enabled macros for transition t21
  DEFINE
```

```
    enStates_t21:=(ss.CS.in_idleHeat);
    enEvents_t21:= 1;
    enCond_t21:=(((ss.AV.actualTemp) - (ss.AV.setTemp)) > (2));
    t21:=(enStates_t21)&(enEvents_t21)&(enCond_t21);

  --define enabled macros for transition t22
  DEFINE
    enStates_t22:=(ss.CS.in_waitForCool);
    enEvents_t22:= 1;
    enCond_t22:=(!(((ss.AV.actualTemp) - (ss.AV.setTemp)) > (2)));
    t22:=(enStates_t22)&(enEvents_t22)&(enCond_t22);

  --define enabled macros for transition t23
  DEFINE
    enStates_t23:=(ss.CS.in_waitForCool);
    enEvents_t23:= 1;
    enCond_t23:=((ss.AV.waitedForCool) < (5));
    t23:=(enStates_t23)&(enEvents_t23)&(enCond_t23);

  --define enabled macros for transition t24
  DEFINE
    enStates_t24:=(ss.CS.in_waitForCool);
    enEvents_t24:= 1;
    enCond_t24:=(((ss.AV.waitedForCool) = (5)) & (!((ss.AV.valvePos) = (0))));
    t24:=(enStates_t24)&(enEvents_t24)&(enCond_t24);


  DEFINE
    any := t21|t22|t23|t24;
  --heatReq is inside interrupt operation, add priority macro
  --priority scheme is scopeOuter
  DEFINE
    pri_t21 := 3;
    pri_t22 := 3;
    pri_t23 := 3;
    pri_t24 := 3;
    pri:= case
      t21 : pri_t21;
      t22 : pri_t22;
      t23 : pri_t23;
      t24 : pri_t24;
      1 : 100;
    esac;


MODULE execute_heatReq(en)
  --transition declaration
  VAR
```

```
    tran : {t21_exe, t22_exe, t23_exe, t24_exe, noTran_exe};

  --execution macros for all transitions
  DEFINE
    t21 := (tran=t21_exe);
    t22 := (tran=t22_exe);
    t23 := (tran=t23_exe);
    t24 := (tran=t24_exe);
    any := !(tran=noTran_exe);

  --define INVAR constrait to define which transition to be executed
  INVAR
     (t21 -> en.t21)
    &(t22 -> en.t22)
    &(t23 -> en.t23)
    &(t24 -> en.t24)


--------------------------------------------------------------------------------

MODULE enabled_controller(ss)
--en_states: srcBlgssCS
--en_events: trigBlgssIEUssIa
--en_states: AVEntailCond
  --define enabled macros for transition t8
  DEFINE
    enStates_t8:=(ss.CS.in_error);
    enEvents_t8:=(ss.Ia.userReset=1);
    enCond_t8 := 1;
    t8:=(enStates_t8)&(enEvents_t8)&(enCond_t8);

  --define enabled macros for transition t9
MODULE states
  VAR
    noHeatReq_state : {idleNoHeat,waitForHeat,noState};
    heatReq_state : {idleHeat,waitForCool,noState};
    controller_state : {off,error,idle,actHeater,heaterRun,noState};
    furnace_state : {furnaceOff,furnaceAct,furnaceRun,furnaceErr,noState};

  --define macros for all states
  DEFINE
    in_heatingSystem := in_house | in_furnace;
    in_house := in_room | in_controller;
    in_room := in_noHeatReq | in_heatReq;
    in_noHeatReq := in_idleNoHeat | in_waitForHeat;
    in_idleNoHeat := noHeatReq_state=idleNoHeat;
    in_waitForHeat := noHeatReq_state=waitForHeat;
    in_heatReq := in_idleHeat | in_waitForCool;
```

```
    in_idleHeat := heatReq_state=idleHeat;
    in_waitForCool := heatReq_state=waitForCool;
    in_controller := in_off | in_error | in_controllerOn;
    in_off := controller_state=off;
    in_error := controller_state=error;
    in_controllerOn := in_idle | in_heaterActive;
    in_idle := controller_state=idle;
    in_heaterActive := in_actHeater | in_heaterRun;
    in_actHeater := controller_state=actHeater;
    in_heaterRun := controller_state=heaterRun;
    in_furnace := in_furnaceNormal | in_furnaceErr;
    in_furnaceNormal := in_furnaceOff | in_furnaceAct | in_furnaceRun;
    in_furnaceOff := furnace_state=furnaceOff;
    in_furnaceAct := furnace_state=furnaceAct;
    in_furnaceRun := furnace_state=furnaceRun;
    in_furnaceErr := furnace_state=furnaceErr;

MODULE envEvents
  VAR
    heatSwitchOn : boolean;
    heatSwitchOff : boolean;
    userReset : boolean;
    furnaceFault : boolean;

MODULE IaEvents
  VAR
    heatSwitchOn : boolean;
    heatSwitchOff : boolean;
    userReset : boolean;
    furnaceFault : boolean;

MODULE intEvents
  VAR
    activate : boolean;
    deactivate : boolean;
    furnaceReset : boolean;
    furnaceRunning : boolean;

MODULE variables
  VAR
    valvePos : 0..2;
    setTemp : 16..24;
    actualTemp : 10..30;
    furnaceStartup : 0..5;
    waitedForWarm : 0..5;
    waitedForCool : 0..5;
    requestHeat : boolean;
    error_variables : boolean;
```

```
MODULE envVars
  VAR
    setTemp : 16..24;
    actualTemp : 10..30;

MODULE inputs
  VAR
    ev : envEvents;
    var : envVars;

MODULE outputs
  VAR
    activate : boolean;
    deactivate : boolean;
    furnaceReset : boolean;
    furnaceRunning : boolean;

MODULE snapshot
  VAR
    CS : states;
    AV : variables;
    IE : intEvents;
    OO : outputs;
    Ia : IaEvents;

-----------------------------------------------------------------------------

MODULE enabled_noHeatReq(ss)
--en_states: srcBlgssCS
--en_events: trigBlgssIEUssIa
--en_states: AVEntailCond
  --define enabled macros for transition t15
  DEFINE
    enStates_t15:=(ss.CS.in_idleNoHeat);
    enEvents_t15:= 1;
    enCond_t15:=(((ss.AV.setTemp) - (ss.AV.actualTemp)) > (2));
    ent15:=(enStates_t15)&(enEvents_t15)&(enCond_t15);

  --define enabled macros for transition t16
  DEFINE
    enStates_t16:=(ss.CS.in_waitForHeat);
    enEvents_t16:= 1;
    enCond_t16:=(!(((ss.AV.setTemp) - (ss.AV.actualTemp)) > (2)));
    ent16:=(enStates_t16)&(enEvents_t16)&(enCond_t16);

  --define enabled macros for transition t17
  DEFINE
```

```
    enStates_t17:=(ss.CS.in_waitForHeat);
    enEvents_t17:= 1;
    enCond_t17:=((ss.AV.waitedForWarm) < (5));
    ent17:=(enStates_t17)&(enEvents_t17)&(enCond_t17);

  --define enabled macros for transition t18
  DEFINE
    enStates_t18:=(ss.CS.in_waitForHeat);
    enEvents_t18:= 1;
    enCond_t18:=(((ss.AV.waitedForWarm) = (5)) & (!((ss.AV.valvePos) = (2))));
    ent18:=(enStates_t18)&(enEvents_t18)&(enCond_t18);


  --define enabled macros for transitions
    t15 := ent15;
    t16 := ent16;
    t17 := ent17;
    t18 := ent18;

  DEFINE
    any := t15|t16|t17|t18;

  --noHeatReq is inside interrupt operation, add priority macro
  --priority scheme is scopeOuter
  DEFINE
    pri_t15 := 3;
    pri_t16 := 3;
    pri_t17 := 3;
    pri_t18 := 3;
    pri:= case
      t15 : pri_t15;
      t16 : pri_t16;
      t17 : pri_t17;
      t18 : pri_t18;
      1 : 100;
    esac;


MODULE execute_noHeatReq(en)
  --transition declaration
  VAR
    tran : {t15_exe, t16_exe, t17_exe, t18_exe, noTran_exe};

  --execution macros for all transitions
  DEFINE
    t15 := (tran=t15_exe);
    t16 := (tran=t16_exe);
    t17 := (tran=t17_exe);
```

```
    t18 := (tran=t18_exe);
    any := !(tran=noTran_exe);

  INVAR
     (t15 -> en.t15)
    &(t16 -> en.t16)
    &(t17 -> en.t17)
    &(t18 -> en.t18)


--------------------------------------------------------------------------------

MODULE enabled_heatReq(ss)
--en_states: srcBlgssCS
--en_events: trigBlgssIEUssIa
--en_states: AVEntailCond
  --define enabled macros for transition t21
  DEFINE
    enStates_t21:=(ss.CS.in_idleHeat);
    enEvents_t21:= 1;
    enCond_t21:=(((ss.AV.actualTemp) - (ss.AV.setTemp)) > (2));
    ent21:=(enStates_t21)&(enEvents_t21)&(enCond_t21);

  --define enabled macros for transition t22
  DEFINE
    enStates_t22:=(ss.CS.in_waitForCool);
    enEvents_t22:= 1;
    enCond_t22:=(!(((ss.AV.actualTemp) - (ss.AV.setTemp)) > (2)));
    ent22:=(enStates_t22)&(enEvents_t22)&(enCond_t22);

  --define enabled macros for transition t23
  DEFINE
    enStates_t23:=(ss.CS.in_waitForCool);
    enEvents_t23:= 1;
    enCond_t23:=((ss.AV.waitedForCool) < (5));
    ent23:=(enStates_t23)&(enEvents_t23)&(enCond_t23);

  --define enabled macros for transition t24
  DEFINE
    enStates_t24:=(ss.CS.in_waitForCool);
    enEvents_t24:= 1;
    enCond_t24:=(((ss.AV.waitedForCool) = (5)) & (!((ss.AV.valvePos) = (0))));
    ent24:=(enStates_t24)&(enEvents_t24)&(enCond_t24);


  --define enabled macros for transitions
    t21 := ent21;
    t22 := ent22;
```

```
    t23 := ent23;
    t24 := ent24;

  DEFINE
    any := t21|t22|t23|t24;

  --heatReq is inside interrupt operation, add priority macro
  --priority scheme is scopeOuter
  DEFINE
    pri_t21 := 3;
    pri_t22 := 3;
    pri_t23 := 3;
    pri_t24 := 3;
    pri:= case
      t21 : pri_t21;
      t22 : pri_t22;
      t23 : pri_t23;
      t24 : pri_t24;
      1 : 100;
    esac;


MODULE execute_heatReq(en)
  --transition declaration
  VAR
    tran : {t21_exe, t22_exe, t23_exe, t24_exe, noTran_exe};

  --execution macros for all transitions
  DEFINE
    t21 := (tran=t21_exe);
    t22 := (tran=t22_exe);
    t23 := (tran=t23_exe);
    t24 := (tran=t24_exe);
    any := !(tran=noTran_exe);

  INVAR
     (t21 -> en.t21)
    &(t22 -> en.t22)
    &(t23 -> en.t23)
    &(t24 -> en.t24)


--------------------------------------------------------------------------------

MODULE enabled_controller(ss)
--en_states: srcBlgssCS
--en_events: trigBlgssIEUssIa
--en_states: AVEntailCond
```

```
--define enabled macros for transition t8
DEFINE
  enStates_t8:=(ss.CS.in_error);
  enEvents_t8:=(ss.Ia.userReset=1);
  enCond_t8 := 1;
  ent8:=(enStates_t8)&(enEvents_t8)&(enCond_t8);

--define enabled macros for transition t9
DEFINE
  enStates_t9:=(ss.CS.in_off);
  enEvents_t9:=(ss.Ia.heatSwitchOn=1);
  enCond_t9 := 1;
  ent9:=(enStates_t9)&(enEvents_t9)&(enCond_t9);

--define enabled macros for transition t10
DEFINE
  enStates_t10:=(ss.CS.in_controllerOn);
  enEvents_t10:=(ss.Ia.heatSwitchOff=1);
  enCond_t10 := 1;
  ent10:=(enStates_t10)&(enEvents_t10)&(enCond_t10);

--define enabled macros for transition t11
DEFINE
  enStates_t11:=(ss.CS.in_controllerOn);
  enEvents_t11:=(ss.Ia.furnaceFault=1);
  enCond_t11 := 1;
  ent11:=(enStates_t11)&(enEvents_t11)&(enCond_t11);

--define enabled macros for transition t12
DEFINE
  enStates_t12:=(ss.CS.in_idle);
  enEvents_t12:= 1;
  enCond_t12:=((ss.AV.requestHeat) = 1);
  ent12:=(enStates_t12)&(enEvents_t12)&(enCond_t12);

--define enabled macros for transition t13
DEFINE
  enStates_t13:=(ss.CS.in_actHeater);
  enEvents_t13:=(ss.IE.furnaceRunning=1);
  enCond_t13 := 1;
  ent13:=(enStates_t13)&(enEvents_t13)&(enCond_t13);

--define enabled macros for transition t14
DEFINE
  enStates_t14:=(ss.CS.in_heaterActive);
  enEvents_t14:= 1;
  enCond_t14:=((ss.AV.requestHeat) = 0);
  ent14:=(enStates_t14)&(enEvents_t14)&(enCond_t14);
```

```
  --define enabled macros for transitions
    t8 := ent8;
    t9 := ent9;
    t10 := ent10;
    t11 := ent11;
    t12 := ent12&!ent8&!ent9&!ent10&!ent11;
    t13 := ent13&!ent8&!ent9&!ent10&!ent11&!ent12&!ent14;
    t14 := ent14&!ent8&!ent9&!ent10&!ent11;

  DEFINE
    any := t8|t9|t10|t11|t12|t13|t14;

MODULE execute_controller(en)
  --transition declaration
  VAR
    tran : {t8_exe, t9_exe, t10_exe, t11_exe, t12_exe, t13_exe, t14_exe, noTran_exe};

  --execution macros for all transitions
  DEFINE
    t8 := (tran=t8_exe);
    t9 := (tran=t9_exe);
    t10 := (tran=t10_exe);
    t11 := (tran=t11_exe);
    t12 := (tran=t12_exe);
    t13 := (tran=t13_exe);
    t14 := (tran=t14_exe);
    any := !(tran=noTran_exe);

  INVAR
     (t8 -> en.t8)
    &(t9 -> en.t9)
    &(t10 -> en.t10)
    &(t11 -> en.t11)
    &(t12 -> en.t12)
    &(t13 -> en.t13)
    &(t14 -> en.t14)


--------------------------------------------------------------------------------

MODULE enabled_furnace(ss)
--en_states: srcBlgssCS
--en_events: trigBlgssIEUssIa
--en_states: AVEntailCond
  --define enabled macros for transition t1
  DEFINE
```

```
  enStates_t1:=(ss.CS.in_furnaceOff);
  enEvents_t1:=(ss.IE.activate=1);
  enCond_t1 := 1;
  ent1:=(enStates_t1)&(enEvents_t1)&(enCond_t1);

--define enabled macros for transition t2
DEFINE
  enStates_t2:=(ss.CS.in_furnaceAct);
  enEvents_t2:=(ss.IE.deactivate=1);
  enCond_t2 := 1;
  ent2:=(enStates_t2)&(enEvents_t2)&(enCond_t2);

--define enabled macros for transition t3
DEFINE
  enStates_t3:=(ss.CS.in_furnaceAct);
  enEvents_t3:= 1;
  enCond_t3:=((ss.AV.furnaceStartup) = (5));
  ent3:=(enStates_t3)&(enEvents_t3)&(enCond_t3);

--define enabled macros for transition t4
DEFINE
  enStates_t4:=(ss.CS.in_furnaceRun);
  enEvents_t4:=(ss.IE.deactivate=1);
  enCond_t4 := 1;
  ent4:=(enStates_t4)&(enEvents_t4)&(enCond_t4);

--define enabled macros for transition t5
DEFINE
  enStates_t5:=(ss.CS.in_furnaceAct);
  enEvents_t5:= 1;
  enCond_t5:=((ss.AV.furnaceStartup) < (5));
  ent5:=(enStates_t5)&(enEvents_t5)&(enCond_t5);

--define enabled macros for transition t6
DEFINE
  enStates_t6:=(ss.CS.in_furnaceErr);
  enEvents_t6:=(ss.IE.furnaceReset=1);
  enCond_t6 := 1;
  ent6:=(enStates_t6)&(enEvents_t6)&(enCond_t6);

--define enabled macros for transition t7
DEFINE
  enStates_t7:=(ss.CS.in_furnaceNormal);
  enEvents_t7:=(ss.Ia.furnaceFault=1);
  enCond_t7 := 1;
  ent7:=(enStates_t7)&(enEvents_t7)&(enCond_t7);
```

```
  --define enabled macros for transitions
    t1 := ent1&!ent6&!ent7;
    t2 := ent2&!ent6&!ent7;
    t3 := ent3&!ent6&!ent7;
    t4 := ent4&!ent6&!ent7;
    t5 := ent5&!ent6&!ent7;
    t6 := ent6;
    t7 := ent7;

  DEFINE
    any := t1|t2|t3|t4|t5|t6|t7;

MODULE execute_furnace(en)
  --transition declaration
  VAR
    tran : {t1_exe, t2_exe, t3_exe, t4_exe, t5_exe, t6_exe, t7_exe, noTran_exe};

  --execution macros for all transitions
  DEFINE
    t1 := (tran=t1_exe);
    t2 := (tran=t2_exe);
    t3 := (tran=t3_exe);
    t4 := (tran=t4_exe);
    t5 := (tran=t5_exe);
    t6 := (tran=t6_exe);
    t7 := (tran=t7_exe);
    any := !(tran=noTran_exe);

  INVAR
     (t1 -> en.t1)
    &(t2 -> en.t2)
    &(t3 -> en.t3)
    &(t4 -> en.t4)
    &(t5 -> en.t5)
    &(t6 -> en.t6)
    &(t7 -> en.t7)



--------------------------------------------------------------------------------


MODULE Parallel(enLeft,enRight,exeLeft,exeRight)
  DEFINE
    any:=exeLeft.any | exeRight.any ;
  INVAR
    (any -> ((enLeft.any -> exeLeft.any) & (enRight.any -> exeRight.any)))
```

```
MODULE enabled_roomIntrTrans(ss)
--en_states: srcBlgssCS
--en_events: trigBlgssIEUssIa
--en_states: AVEntailCond
  --define enabled macros for transition t19
  DEFINE
    enStates_t19:=(ss.CS.in_waitForCool);
    enEvents_t19:= 1;
    enCond_t19:=((((ss.AV.waitedForCool) = (5)) & ((ss.AV.valvePos) = (0)))
                 & (((ss.AV.actualTemp) - (ss.AV.setTemp)) > (2)));
    ent19:=(enStates_t19)&(enEvents_t19)&(enCond_t19);

  --define enabled macros for transition t20
  DEFINE
    enStates_t20:=(ss.CS.in_waitForHeat);
    enEvents_t20:= 1;
    enCond_t20:=((((ss.AV.waitedForWarm) = (5)) & ((ss.AV.valvePos) = (2)))
                & (((ss.AV.setTemp) - (ss.AV.actualTemp)) > (2)));
    ent20:=(enStates_t20)&(enEvents_t20)&(enCond_t20);


  --define enabled macros for transitions
    t19 := ent19;
    t20 := ent20;

  DEFINE
    any := t19|t20;

  --roomIntrTrans is inside interrupt operation, add priority macro
  --priority scheme is scopeOuter
  DEFINE
    pri_t19 := 2;
    pri_t20 := 2;
    pri:= case
      t19 : pri_t19;
      t20 : pri_t20;
      1 : 100;
    esac;


MODULE execute_roomIntrTrans(en)
  --transition declaration
  VAR
    tran : {t19_exe, t20_exe, noTran_exe};

  --execution macros for all transitions
  DEFINE
    t19 := (tran=t19_exe);
```

```
    t20 := (tran=t20_exe);
    any := !(tran=noTran_exe);

  INVAR
     (t19 -> en.t19)
    &(t20 -> en.t20)


MODULE Interrupt(enLeft,enRight,enIntrTran,exeLeft,exeRight,exeIntrTran)
 DEFINE
   any:=exeLeft.any | exeRight.any | exeIntrTran.any;
 INVAR
   (exeLeft.any -> (enLeft.pri <= enIntrTran.pri))
  &(exeRight.any -> (enRight.pri <= enIntrTran.pri))
  &(exeIntrTran.any -> ((enIntrTran.pri <= enLeft.pri) & (enIntrTran.pri <= enRight.pri)))
  &!(exeLeft.any & exeIntrTran.any)
  &!(exeRight.any & exeIntrTran.any)


--------------------------------------------------------------------------

MODULE reset(stable,ss,I)
  VAR
    CS : resetCS(stable,ss);
    AV : resetAV(stable,ss,I);
    IE : resetIE(stable,ss);
    OO : resetO(stable,ss);
    Ia : resetIa(stable,ss,I);

MODULE resetCS(stable,ss)
--in stable macro semantics
--resetCS : ssCS
  DEFINE
    noHeatReq_state := case
      stable : ss.CS.noHeatReq_state;
      1 : ss.CS.noHeatReq_state;
    esac;
  DEFINE
    heatReq_state := case
      stable : ss.CS.heatReq_state;
      1 : ss.CS.heatReq_state;
    esac;
  DEFINE
    controller_state := case
      stable : ss.CS.controller_state;
      1 : ss.CS.controller_state;
    esac;
  DEFINE
```

```
    furnace_state := case
      stable : ss.CS.furnace_state;
      1 : ss.CS.furnace_state;
    esac;
  --define macros for all states
  DEFINE
    in_heatingSystem := in_house | in_furnace;
    in_house := in_room | in_controller;
    in_room := in_noHeatReq | in_heatReq;
    in_noHeatReq := in_idleNoHeat | in_waitForHeat;
    in_idleNoHeat := noHeatReq_state=idleNoHeat;
    in_waitForHeat := noHeatReq_state=waitForHeat;
    in_heatReq := in_idleHeat | in_waitForCool;
    in_idleHeat := heatReq_state=idleHeat;
    in_waitForCool := heatReq_state=waitForCool;
    in_controller := in_off | in_error | in_controllerOn;
    in_off := controller_state=off;
    in_error := controller_state=error;
    in_controllerOn := in_idle | in_heaterActive;
    in_idle := controller_state=idle;
    in_heaterActive := in_actHeater | in_heaterRun;
    in_actHeater := controller_state=actHeater;
    in_heaterRun := controller_state=heaterRun;
    in_furnace := in_furnaceNormal | in_furnaceErr;
    in_furnaceNormal := in_furnaceOff | in_furnaceAct | in_furnaceRun;
    in_furnaceOff := furnace_state=furnaceOff;
    in_furnaceAct := furnace_state=furnaceAct;
    in_furnaceRun := furnace_state=furnaceRun;
    in_furnaceErr := furnace_state=furnaceErr;

MODULE resetAV(stable,ss,I)
--resetAV : ssAV
  DEFINE
    valvePos := case
      stable : ss.AV.valvePos;
      1 : ss.AV.valvePos;
    esac;

  DEFINE
    setTemp := case
      stable : I.var.setTemp;
      1 : ss.AV.setTemp;
    esac;

  DEFINE
    actualTemp := case
      stable : I.var.actualTemp;
      1 : ss.AV.actualTemp;
```

```
    esac;

  DEFINE
    furnaceStartup := case
      stable : ss.AV.furnaceStartup;
      1 : ss.AV.furnaceStartup;
    esac;

  DEFINE
    waitedForWarm := case
      stable : ss.AV.waitedForWarm;
      1 : ss.AV.waitedForWarm;
    esac;

  DEFINE
    waitedForCool := case
      stable : ss.AV.waitedForCool;
      1 : ss.AV.waitedForCool;
    esac;

  DEFINE
    requestHeat := case
      stable : ss.AV.requestHeat;
      1 : ss.AV.requestHeat;
    esac;

DEFINE error_variables := ss.AV.error_variables;

MODULE resetIE(stable,ss)
--resetIE : resetIEEPT
  DEFINE
    activate := case
      stable : 0;
      1 : ss.IE.activate;
    esac;

  DEFINE
    deactivate := case
      stable : 0;
      1 : ss.IE.deactivate;
    esac;

  DEFINE
    furnaceReset := case
      stable : 0;
      1 : ss.IE.furnaceReset;
    esac;
```

```
  DEFINE
    furnaceRunning := case
      stable : 0;
      1 : ss.IE.furnaceRunning;
    esac;


MODULE resetO(stable,ss)
--resetO : resetOEPT
  DEFINE
    activate := case
      stable : 0;
      1 : ss.OO.activate;
    esac;

  DEFINE
    deactivate := case
      stable : 0;
      1 : ss.OO.deactivate;
    esac;

  DEFINE
    furnaceReset := case
      stable : 0;
      1 : ss.OO.furnaceReset;
    esac;

  DEFINE
    furnaceRunning := case
      stable : 0;
      1 : ss.OO.furnaceRunning;
    esac;


MODULE resetIa(stable,ss,I)
--resetIa : resetIaI
  DEFINE
    heatSwitchOn := case
      stable : I.ev.heatSwitchOn;
      1 : ss.Ia.heatSwitchOn;
    esac;

  DEFINE
    heatSwitchOff := case
      stable : I.ev.heatSwitchOff;
      1 : ss.Ia.heatSwitchOff;
    esac;
```

```
  DEFINE
    userReset := case
      stable : I.ev.userReset;
      1 : ss.Ia.userReset;
    esac;

  DEFINE
    furnaceFault := case
      stable : I.ev.furnaceFault;
      1 : ss.Ia.furnaceFault;
    esac;


-------------------------------------------------------------------------

MODULE enabled(ss)
  VAR
    noHeatReq : enabled_noHeatReq(ss);
    heatReq : enabled_heatReq(ss);
    controller : enabled_controller(ss);
    furnace : enabled_furnace(ss);

    roomIntrTrans : enabled_roomIntrTrans(ss);
  --define enabled macros for each composite HTSs
  DEFINE
  --define en.heatingSystem.any that use composition Parallel
    heatingSystem.any := house.any | furnace.any;

  --define en.house.any that use composition Parallel
    house.any := room.any | controller.any;

  --define en.room.any that use composition Interrupt
    room.any := noHeatReq.any | heatReq.any | roomIntrTrans.any;


 DEFINE
  stable := !(heatingSystem.any);
-------------------------------------------------------------------------

MODULE execute(en)
  VAR
    noHeatReq : execute_noHeatReq(en.noHeatReq);
    heatReq : execute_heatReq(en.heatReq);
    controller : execute_controller(en.controller);
    furnace : execute_furnace(en.furnace);
    roomIntrTrans : execute_roomIntrTrans(en.roomIntrTrans);
    heatingSystem : Parallel(en.house,en.furnace,
                         house,furnace);
```

```
    house : Parallel(en.room,en.controller,
                        room,controller);
    room : Interrupt(en.noHeatReq,en.heatReq,en.roomIntrTrans,
                          noHeatReq,heatReq,roomIntrTrans);

  INVAR
    (en.heatingSystem.any -> heatingSystem.any)



--------------------------------------------------------------------------

MODULE apply(pss,iss,exe)
  VAR
    nextCS : nextCS(pss,iss,exe);
    nextAV : nextAV(pss,iss,exe);
    nextIE : nextIE(pss,iss,exe);
    nextO : nextO(pss,iss,exe);
    nextIa : nextIa(pss,iss,exe);

MODULE nextCS(pss,iss,exe)
--nextCS: enterDestT
  ASSIGN
    next(pss.CS.noHeatReq_state):=case
      exe.noHeatReq.t15 : waitForHeat;
      exe.noHeatReq.t16 : idleNoHeat;
      exe.noHeatReq.t17 : waitForHeat;
      exe.noHeatReq.t18 : waitForHeat;
      exe.roomIntrTrans.t19 : idleNoHeat;
      exe.roomIntrTrans.t20 : noState;
      1 : iss.CS.noHeatReq_state;
    esac;

  ASSIGN
    next(pss.CS.heatReq_state):=case
      exe.heatReq.t21 : waitForCool;
      exe.heatReq.t22 : idleHeat;
      exe.heatReq.t23 : waitForCool;
      exe.heatReq.t24 : waitForCool;
      exe.roomIntrTrans.t19 : noState;
      exe.roomIntrTrans.t20 : idleHeat;
      1 : iss.CS.heatReq_state;
    esac;

  ASSIGN
    next(pss.CS.controller_state):=case
      exe.controller.t8 : off;
      exe.controller.t9 : idle;
```

```
      exe.controller.t10 : off;
      exe.controller.t11 : error;
      exe.controller.t12 : actHeater;
      exe.controller.t13 : heaterRun;
      exe.controller.t14 : idle;
      1 : iss.CS.controller_state;
    esac;

  ASSIGN
    next(pss.CS.furnace_state):=case
      exe.furnace.t1 : furnaceAct;
      exe.furnace.t2 : furnaceOff;
      exe.furnace.t3 : furnaceRun;
      exe.furnace.t4 : furnaceOff;
      exe.furnace.t5 : furnaceAct;
      exe.furnace.t6 : furnaceOff;
      exe.furnace.t7 : furnaceErr;
      1 : iss.CS.furnace_state;
    esac;


MODULE nextAV(pss,iss,exe)
--nextAV : evalLastAsn
  --next state relation for  valvePos
  DEFINE valvePost15 := case
  ((((iss.AV.valvePos) + (1)))>=0)&((((iss.AV.valvePos) + (1)))<=2)
               : ((iss.AV.valvePos) + (1));
  1 : iss.AV.valvePos;
 esac;
  DEFINE valvePost15error := exe.noHeatReq.t15
              & ((((iss.AV.valvePos) + (1)))<0)|((((iss.AV.valvePos) + (1)))>2);
  DEFINE valvePost18 := case
  ((((iss.AV.valvePos) + (1)))>=0)&((((iss.AV.valvePos) + (1)))<=2)
               : ((iss.AV.valvePos) + (1));
  1 : iss.AV.valvePos;
 esac;
  DEFINE valvePost18error := exe.noHeatReq.t18
              & ((((iss.AV.valvePos) + (1)))<0)|((((iss.AV.valvePos) + (1)))>2);
  DEFINE valvePost21 := case
  ((((iss.AV.valvePos) - (1)))>=0)&((((iss.AV.valvePos) - (1)))<=2)
               : ((iss.AV.valvePos) - (1));
  1 : iss.AV.valvePos;
 esac;
  DEFINE valvePost21error := exe.heatReq.t21
              & ((((iss.AV.valvePos) - (1)))<0)|((((iss.AV.valvePos) - (1)))>2);
  DEFINE valvePost24 := case
  ((((iss.AV.valvePos) - (1)))>=0)&((((iss.AV.valvePos) - (1)))<=2)
               : ((iss.AV.valvePos) - (1));
```

```
 1 : iss.AV.valvePos;
esac;
 DEFINE valvePost24error := exe.heatReq.t24
            & ((((iss.AV.valvePos) - (1)))<0)|((((iss.AV.valvePos) - (1)))>2);
 ASSIGN
   next(pss.AV.valvePos):=case
     exe.noHeatReq.t15 : valvePost15;
     exe.noHeatReq.t18 : valvePost18;
     exe.heatReq.t21 : valvePost21;
     exe.heatReq.t24 : valvePost24;
     1 : iss.AV.valvePos;
   esac;

 --next state relation for  furnaceStartup
 DEFINE furnaceStartupt1 := case
 (((0))>=0)&(((0))<=5) : (0);
 1 : iss.AV.furnaceStartup;
esac;
 DEFINE furnaceStartupt1error := exe.furnace.t1 & (((0))<0)|(((0))>5);
 DEFINE furnaceStartupt5 := case
 ((((iss.AV.furnaceStartup) + (1)))>=0)&((((iss.AV.furnaceStartup) + (1)))<=5)
             : ((iss.AV.furnaceStartup) + (1));
 1 : iss.AV.furnaceStartup;
esac;
 DEFINE furnaceStartupt5error := exe.furnace.t5
       & ((((iss.AV.furnaceStartup) + (1)))<0)|((((iss.AV.furnaceStartup) + (1)))>5);
 ASSIGN
   next(pss.AV.furnaceStartup):=case
     exe.furnace.t1 : furnaceStartupt1;
     exe.furnace.t5 : furnaceStartupt5;
     1 : iss.AV.furnaceStartup;
   esac;

 --next state relation for  waitedForWarm
 DEFINE waitedForWarmt15 := case
 (((0))>=0)&(((0))<=5) : (0);
 1 : iss.AV.waitedForWarm;
esac;
 DEFINE waitedForWarmt15error := exe.noHeatReq.t15 & (((0))<0)|(((0))>5);
 DEFINE waitedForWarmt17 := case
 ((((iss.AV.waitedForWarm) + (1)))>=0)&((((iss.AV.waitedForWarm) + (1)))<=5)
             : ((iss.AV.waitedForWarm) + (1));
 1 : iss.AV.waitedForWarm;
esac;
 DEFINE waitedForWarmt17error := exe.noHeatReq.t17
          & ((((iss.AV.waitedForWarm) + (1)))<0)|((((iss.AV.waitedForWarm) + (1)))>5);
 DEFINE waitedForWarmt18 := case
 (((0))>=0)&(((0))<=5) : (0);
```

```
 1 : iss.AV.waitedForWarm;
esac;
 DEFINE waitedForWarmt18error := exe.noHeatReq.t18 & (((0))<0)|(((0))>5);
 ASSIGN
   next(pss.AV.waitedForWarm):=case
     exe.noHeatReq.t15 : waitedForWarmt15;
     exe.noHeatReq.t17 : waitedForWarmt17;
     exe.noHeatReq.t18 : waitedForWarmt18;
     1 : iss.AV.waitedForWarm;
   esac;

 --next state relation for  waitedForCool
 DEFINE waitedForCoolt21 := case
 (((0))>=0)&(((0))<=5) : (0);
 1 : iss.AV.waitedForCool;
esac;
 DEFINE waitedForCoolt21error := exe.heatReq.t21 & (((0))<0)|(((0))>5);
 DEFINE waitedForCoolt23 := case
 ((((iss.AV.waitedForCool) + (1)))>=0)&((((iss.AV.waitedForCool) + (1)))<=5)
             : ((iss.AV.waitedForCool) + (1));
 1 : iss.AV.waitedForCool;
esac;
 DEFINE waitedForCoolt23error := exe.heatReq.t23
          & ((((iss.AV.waitedForCool) + (1)))<0)|((((iss.AV.waitedForCool) + (1)))>5);
 DEFINE waitedForCoolt24 := case
 (((0))>=0)&(((0))<=5) : (0);
 1 : iss.AV.waitedForCool;
esac;
 DEFINE waitedForCoolt24error := exe.heatReq.t24 & (((0))<0)|(((0))>5);
 ASSIGN
   next(pss.AV.waitedForCool):=case
     exe.heatReq.t21 : waitedForCoolt21;
     exe.heatReq.t23 : waitedForCoolt23;
     exe.heatReq.t24 : waitedForCoolt24;
     1 : iss.AV.waitedForCool;
   esac;

 --next state relation for  requestHeat
 ASSIGN
   next(pss.AV.requestHeat):=case
     1 : iss.AV.requestHeat;
   esac;


 --next state relation for variable overflow and underflow
 ASSIGN
   next(pss.AV.error_variables):=
      iss.AV.error_variables
```

```
      |valvePost15error
      |valvePost18error
      |valvePost21error
      |valvePost24error
      |furnaceStartupt1error
      |furnaceStartupt5error
      |waitedForWarmt15error
      |waitedForWarmt17error
      |waitedForWarmt18error
      |waitedForCoolt21error
      |waitedForCoolt23error
      |waitedForCoolt24error
       ;

MODULE nextIE(pss,iss,exe)
--nextIE : Gen

  --next state relation for  activate
  ASSIGN
    next(pss.IE.activate):=case
      exe.controller.t12 : 1;
      1 : 0;
    esac;

  --next state relation for  deactivate
  ASSIGN
    next(pss.IE.deactivate):=case
      exe.controller.t10 : 1;
      exe.controller.t14 : 1;
      1 : 0;
    esac;

  --next state relation for  furnaceReset
  ASSIGN
    next(pss.IE.furnaceReset):=case
      exe.controller.t8 : 1;
      1 : 0;
    esac;

  --next state relation for  furnaceRunning
  ASSIGN
    next(pss.IE.furnaceRunning):=case
      exe.furnace.t3 : 1;
      1 : 0;
    esac;
```

```
MODULE nextO(pss,iss,exe)
--nextO : OGen

  --next state relation for output activate
  ASSIGN
    next(pss.OO.activate):=case
      exe.controller.t12 : 1;
      1 : 0;
    esac;

  --next state relation for output deactivate
  ASSIGN
    next(pss.OO.deactivate):=case
      exe.controller.t10 : 1;
      exe.controller.t14 : 1;
      1 : 0;
    esac;

  --next state relation for output furnaceReset
  ASSIGN
    next(pss.OO.furnaceReset):=case
      exe.controller.t8 : 1;
      1 : 0;
    esac;

  --next state relation for output furnaceRunning
  ASSIGN
    next(pss.OO.furnaceRunning):=case
      exe.furnace.t3 : 1;
      1 : 0;
    esac;



MODULE nextIa(pss,iss,exe)
--nextIa : nextIaEPT
  --next state relation for heatSwitchOn
  ASSIGN
    next(pss.Ia.heatSwitchOn) := 0;

  --next state relation for heatSwitchOff
  ASSIGN
    next(pss.Ia.heatSwitchOff) := 0;

  --next state relation for userReset
  ASSIGN
    next(pss.Ia.userReset) := 0;
```

```
  --next state relation for furnaceFault
  ASSIGN
    next(pss.Ia.furnaceFault) := 0;


-------------------------------------------------------------------------

MODULE initss(pss)
  ASSIGN
    init(pss.CS.noHeatReq_state):=idleNoHeat;
    init(pss.CS.heatReq_state):=idleHeat;
    init(pss.CS.controller_state):=off;
    init(pss.CS.furnace_state):=furnaceOff;
    init(pss.AV.valvePos):={0};
    init(pss.AV.setTemp):={20};
    init(pss.AV.actualTemp):={16};
    init(pss.AV.furnaceStartup):={0};
    init(pss.AV.waitedForWarm):={0};
    init(pss.AV.waitedForCool):={0};
    init(pss.AV.requestHeat):={0};
    init(pss.AV.error_variables):=0;
    init(pss.IE.activate):=0;
    init(pss.IE.deactivate):=0;
    init(pss.IE.furnaceReset):=0;
    init(pss.IE.furnaceRunning):=0;
    init(pss.Ia.heatSwitchOn):=0;
    init(pss.Ia.heatSwitchOff):=0;
    init(pss.Ia.userReset):=0;
    init(pss.Ia.furnaceFault):=0;
MODULE main
  VAR
    --pss is a set of variables storing snapshot elements
    pss : snapshot;
    --I is a set of inputs
    I : inputs;
    --pss_en is a set of macros identifying enabled entities in pss
    pss_en: enabled(pss);
    --iss is a set of macros of type snapshot
    iss : reset(pss_en.stable,pss,I);
    --iss_en is a set of macros identifying enabled entities in iss
    iss_en: enabled(iss);
    --iss_exe is a set of macros identifying executing entities
    iss_exe: execute(iss_en);
    --initss is a module containing initlialization statements
    _initss: initss(pss);
    --apply is a module containing next statements
    _apply : apply(pss,iss,iss_exe);
```

Appendix B

Generated SMV Model for Single Lane Bridge

The single lane bridge specification was given in page 100.

```
MODULE states
  VAR
    redAHts_state : {waitRedA,moveOnRedA,moveOutRedA,onRedA,noState};
    redBHts_state : {waitRedB,moveOnRedB,moveOutRedB,onRedB,noState};
    blueAHts_state : {waitBlueA,moveOnBlueA,moveOutBlueA,onBlueA,noState};
    blueBHts_state : {waitBlueB,moveOnBlueB,moveOutBlueB,onBlueB,noState};
    bridgeStatusHts_state : {empty,oneRed,twoRed,oneBlue,twoBlue,noState};
    redCoordEntHts_state : {coordEntRedA,coordEntRedB,noState};
    redCoordExitHts_state : {coordExitRedA,coordExitRedB,noState};
    blueCoordEntHts_state : {coordEntBlueA,coordEntBlueB,noState};
    blueCoordExitHts_state : {coordExitBlueA,coordExitBlueB,noState};

  --define macros for all states
  DEFINE
    in_singleLaneBridge := in_bridge | in_coord;
    in_bridge := in_car | in_bridgeStatus;
    in_car := in_redCar | in_blueCar;
    in_redCar := in_redA | in_redB;
    in_redA := in_waitRedA | in_moveOnRedA | in_moveOutRedA | in_onRedA;
    in_waitRedA := redAHts_state=waitRedA;
    in_moveOnRedA := redAHts_state=moveOnRedA;
    in_moveOutRedA := redAHts_state=moveOutRedA;
    in_onRedA := redAHts_state=onRedA;
    in_redB := in_waitRedB | in_moveOnRedB | in_moveOutRedB | in_onRedB;
    in_waitRedB := redBHts_state=waitRedB;
    in_moveOnRedB := redBHts_state=moveOnRedB;
```

```
    in_moveOutRedB := redBHts_state=moveOutRedB;
    in_onRedB := redBHts_state=onRedB;
    in_blueCar := in_blueA | in_blueB;
    in_blueA := in_waitBlueA | in_moveOnBlueA | in_moveOutBlueA | in_onBlueA;
    in_waitBlueA := blueAHts_state=waitBlueA;
    in_moveOnBlueA := blueAHts_state=moveOnBlueA;
    in_moveOutBlueA := blueAHts_state=moveOutBlueA;
    in_onBlueA := blueAHts_state=onBlueA;
    in_blueB := in_waitBlueB | in_moveOnBlueB | in_moveOutBlueB | in_onBlueB;
    in_waitBlueB := blueBHts_state=waitBlueB;
    in_moveOnBlueB := blueBHts_state=moveOnBlueB;
    in_moveOutBlueB := blueBHts_state=moveOutBlueB;
    in_onBlueB := blueBHts_state=onBlueB;
    in_bridgeStatus := in_empty | in_oneRed | in_twoRed | in_oneBlue | in_twoBlue;
    in_empty := bridgeStatusHts_state=empty;
    in_oneRed := bridgeStatusHts_state=oneRed;
    in_twoRed := bridgeStatusHts_state=twoRed;
    in_oneBlue := bridgeStatusHts_state=oneBlue;
    in_twoBlue := bridgeStatusHts_state=twoBlue;
    in_coord := in_coordRed | in_coordBlue;
    in_coordRed := in_redCoordEnt | in_redCoordExit;
    in_redCoordEnt := in_coordEntRedA | in_coordEntRedB;
    in_coordEntRedA := redCoordEntHts_state=coordEntRedA;
    in_coordEntRedB := redCoordEntHts_state=coordEntRedB;
    in_redCoordExit := in_coordExitRedA | in_coordExitRedB;
    in_coordExitRedA := redCoordExitHts_state=coordExitRedA;
    in_coordExitRedB := redCoordExitHts_state=coordExitRedB;
    in_coordBlue := in_blueCoordEnt | in_blueCoordExit;
    in_blueCoordEnt := in_coordEntBlueA | in_coordEntBlueB;
    in_coordEntBlueA := blueCoordEntHts_state=coordEntBlueA;
    in_coordEntBlueB := blueCoordEntHts_state=coordEntBlueB;
    in_blueCoordExit := in_coordExitBlueA | in_coordExitBlueB;
    in_coordExitBlueA := blueCoordExitHts_state=coordExitBlueA;
    in_coordExitBlueB := blueCoordExitHts_state=coordExitBlueB;

MODULE envEvents
  VAR
    entRedA : boolean;
    exitRedA : boolean;
    entRedB : boolean;
    exitRedB : boolean;
    entBlueA : boolean;
    exitBlueA : boolean;
    entBlueB : boolean;
    exitBlueB : boolean;

MODULE IaEvents
  VAR
```

```
    entRedA : boolean;
    exitRedA : boolean;
    entRedB : boolean;
    exitRedB : boolean;
    entBlueA : boolean;
    exitBlueA : boolean;
    entBlueB : boolean;
    exitBlueB : boolean;

MODULE variables
  VAR
    numRed : 0..2;
    numBlue : 0..2;
    error_variables : boolean;

MODULE envVars
  VAR

MODULE inputs
  VAR
    ev : envEvents;
    var : envVars;

MODULE outputs
  VAR
    inRed : boolean;
    outRed : boolean;
    inBlue : boolean;
    outBlue : boolean;

MODULE snapshot
  VAR
    CS : states;
    AV : variables;
    OO : outputs;
    Ia : IaEvents;

-------------------------------------------------------------------------

MODULE enabled_redAHts(ss,rend_events)
--en_states: srcBlgssCS
--en_events: trigBlgssIa
--en_states: AVEntailCond
  --define enabled macros for transition t1
  DEFINE
    enStates_t1:=(ss.CS.in_waitRedA);
    enEvents_t1:=(ss.Ia.entRedA=1);
    enCond_t1:=(((ss.AV.numRed) < (2)) & ((ss.AV.numBlue) = (0)));
```

```
  ent1:=(enStates_t1)&(enEvents_t1)&(enCond_t1);

--define enabled macros for transition t2
DEFINE
  enStates_t2:=(ss.CS.in_onRedA);
  enEvents_t2:=(ss.Ia.exitRedA=1);
  enCond_t2 := 1;
  ent2:=(enStates_t2)&(enEvents_t2)&(enCond_t2);

--define enabled macros for transition t25
DEFINE
  enStates_t25:=(ss.CS.in_moveOnRedA);
  enEvents_t25:= 1;
  enCond_t25 := 1;
  ent25:=(enStates_t25)&(enEvents_t25)&(enCond_t25);

--define enabled macros for transition t26
DEFINE
  enStates_t26:=(ss.CS.in_moveOutRedA);
  enEvents_t26:= 1;
  enCond_t26 := 1;
  ent26:=(enStates_t26)&(enEvents_t26)&(enCond_t26);


--define enabled macros for transitions
  t1 := ent1;
  t2 := ent2;
  t25 := ent25;
  t26 := ent26;

DEFINE
  any := t1|t2|t25|t26;
--define the env sync event triggering macros for simple Hts redAHts
DEFINE
  entRedA_trig:=t1;
  exitRedA_trig:=t2;
  entRedB_trig:=0;
  exitRedB_trig:=0;
  entBlueA_trig:=0;
  exitBlueA_trig:=0;
  entBlueB_trig:=0;
  exitBlueB_trig:=0;

--define the other triggering macro
--for the env sync events for simple Hts redAHts
DEFINE
  env_other_trig:=t25|t26;
```

```
  --define the rend sync event triggering macros for simple Hts redAHts
  DEFINE
    inRed_trig:=0;
    outRed_trig:=0;
    inBlue_trig:=0;
    outBlue_trig:=0;

  --define the rend sync event generating macros for simple Hts redAHts
  DEFINE
    inRed_gen:=t25;
    outRed_gen:=t26;
    inBlue_gen:=0;
    outBlue_gen:=0;

  --define the other triggering macro
  --for the rend sync event for simple Hts redAHts
  DEFINE
    rend_other_trig:=t1|t2;


MODULE execute_redAHts(en)
  --transition declaration
  VAR
    tran : {t1_exe, t2_exe, t25_exe, t26_exe, noTran_exe};

  --execution macros for all transitions
  DEFINE
    t1 := (tran=t1_exe);
    t2 := (tran=t2_exe);
    t25 := (tran=t25_exe);
    t26 := (tran=t26_exe);
    any := !(tran=noTran_exe);

  INVAR
     (t1 -> en.t1)
    &(t2 -> en.t2)
    &(t25 -> en.t25)
    &(t26 -> en.t26)
    &(any -> en.any)

  --define the env sync event triggering macros for simple Hts redAHts
  DEFINE
    entRedA_trig:=t1;
    exitRedA_trig:=t2;
    entRedB_trig:=0;
    exitRedB_trig:=0;
    entBlueA_trig:=0;
    exitBlueA_trig:=0;
```

```
    entBlueB_trig:=0;
    exitBlueB_trig:=0;

  --define the other triggering macro
  --for the env sync events for simple Hts redAHts
  DEFINE
    env_other_trig:=t25|t26;

  --redAHts is inside rend sync operation,
  --define flag to check whether more than 1 transition are to execute
  DEFINE
    more_than_one:=0;

  --define the rend sync event triggering macros for simple Hts redAHts
  DEFINE
    inRed_trig:=0;
    outRed_trig:=0;
    inBlue_trig:=0;
    outBlue_trig:=0;

  --define the rend sync event generating macros for simple Hts redAHts
  DEFINE
    inRed_gen:=t25;
    outRed_gen:=t26;
    inBlue_gen:=0;
    outBlue_gen:=0;

  --define the other triggering macro
  --for the rend sync event for simple Hts redAHts
  DEFINE
    rend_other_trig:=t1|t2;


--------------------------------------------------------------------------------

MODULE enabled_redBHts(ss,rend_events)
--en_states: srcBlgssCS
--en_events: trigBlgssIa
--en_states: AVEntailCond
  --define enabled macros for transition t3
  DEFINE
    enStates_t3:=(ss.CS.in_waitRedB);
    enEvents_t3:=(ss.Ia.entRedB=1);
    enCond_t3:=(((ss.AV.numRed) < (2)) & ((ss.AV.numBlue) = (0)));
    ent3:=(enStates_t3)&(enEvents_t3)&(enCond_t3);

  --define enabled macros for transition t4
  DEFINE
```

```
  enStates_t4:=(ss.CS.in_onRedB);
  enEvents_t4:=(ss.Ia.exitRedB=1);
  enCond_t4 := 1;
  ent4:=(enStates_t4)&(enEvents_t4)&(enCond_t4);

--define enabled macros for transition t27
DEFINE
  enStates_t27:=(ss.CS.in_moveOnRedB);
  enEvents_t27:= 1;
  enCond_t27 := 1;
  ent27:=(enStates_t27)&(enEvents_t27)&(enCond_t27);

--define enabled macros for transition t28
DEFINE
  enStates_t28:=(ss.CS.in_moveOutRedB);
  enEvents_t28:= 1;
  enCond_t28 := 1;
  ent28:=(enStates_t28)&(enEvents_t28)&(enCond_t28);


--define enabled macros for transitions
  t3 := ent3;
  t4 := ent4;
  t27 := ent27;
  t28 := ent28;

DEFINE
  any := t3|t4|t27|t28;
--define the env sync event triggering macros for simple Hts redBHts
DEFINE
  entRedA_trig:=0;
  exitRedA_trig:=0;
  entRedB_trig:=t3;
  exitRedB_trig:=t4;
  entBlueA_trig:=0;
  exitBlueA_trig:=0;
  entBlueB_trig:=0;
  exitBlueB_trig:=0;

--define the other triggering macro
--for the env sync events for simple Hts redBHts
DEFINE
  env_other_trig:=t27|t28;

--define the rend sync event triggering macros for simple Hts redBHts
DEFINE
  inRed_trig:=0;
  outRed_trig:=0;
```

```
    inBlue_trig:=0;
    outBlue_trig:=0;

  --define the rend sync event generating macros for simple Hts redBHts
  DEFINE
    inRed_gen:=0;
    outRed_gen:=0;
    inBlue_gen:=t27;
    outBlue_gen:=t28;

  --define the other triggering macro
  --for the rend sync event for simple Hts redBHts
  DEFINE
    rend_other_trig:=t3|t4;


MODULE execute_redBHts(en)
  --transition declaration
  VAR
    tran : {t3_exe, t4_exe, t27_exe, t28_exe, noTran_exe};

  --execution macros for all transitions
  DEFINE
    t3 := (tran=t3_exe);
    t4 := (tran=t4_exe);
    t27 := (tran=t27_exe);
    t28 := (tran=t28_exe);
    any := !(tran=noTran_exe);

  INVAR
     (t3 -> en.t3)
    &(t4 -> en.t4)
    &(t27 -> en.t27)
    &(t28 -> en.t28)
    &(any -> en.any)

  --define the env sync event triggering macros for simple Hts redBHts
  DEFINE
    entRedA_trig:=0;
    exitRedA_trig:=0;
    entRedB_trig:=t3;
    exitRedB_trig:=t4;
    entBlueA_trig:=0;
    exitBlueA_trig:=0;
    entBlueB_trig:=0;
    exitBlueB_trig:=0;

  --define the other triggering macro
```

```
  --for the env sync events for simple Hts redBHts
  DEFINE
    env_other_trig:=t27|t28;

  --redBHts is inside rend sync operation,
  --define flag to check whether more than 1 transition are to execute
  DEFINE
    more_than_one:=0;

  --define the rend sync event triggering macros for simple Hts redBHts
  DEFINE
    inRed_trig:=0;
    outRed_trig:=0;
    inBlue_trig:=0;
    outBlue_trig:=0;

  --define the rend sync event generating macros for simple Hts redBHts
  DEFINE
    inRed_gen:=0;
    outRed_gen:=0;
    inBlue_gen:=t27;
    outBlue_gen:=t28;

  --define the other triggering macro
  --for the rend sync event for simple Hts redBHts
  DEFINE
    rend_other_trig:=t3|t4;


-------------------------------------------------------------------------------

MODULE enabled_blueAHts(ss,rend_events)
--en_states: srcBlgssCS
--en_events: trigBlgssIa
--en_states: AVEntailCond
  --define enabled macros for transition t9
  DEFINE
    enStates_t9:=(ss.CS.in_waitBlueA);
    enEvents_t9:=(ss.Ia.entBlueA=1);
    enCond_t9:=(((ss.AV.numBlue) < (2)) & ((ss.AV.numRed) = (0)));
    ent9:=(enStates_t9)&(enEvents_t9)&(enCond_t9);

  --define enabled macros for transition t10
  DEFINE
    enStates_t10:=(ss.CS.in_onBlueA);
    enEvents_t10:=(ss.Ia.exitBlueA=1);
    enCond_t10 := 1;
    ent10:=(enStates_t10)&(enEvents_t10)&(enCond_t10);
```

```
--define enabled macros for transition t29
DEFINE
  enStates_t29:=(ss.CS.in_moveOnBlueA);
  enEvents_t29:= 1;
  enCond_t29 := 1;
  ent29:=(enStates_t29)&(enEvents_t29)&(enCond_t29);

--define enabled macros for transition t30
DEFINE
  enStates_t30:=(ss.CS.in_moveOutBlueA);
  enEvents_t30:= 1;
  enCond_t30 := 1;
  ent30:=(enStates_t30)&(enEvents_t30)&(enCond_t30);


--define enabled macros for transitions
  t9 := ent9;
  t10 := ent10;
  t29 := ent29;
  t30 := ent30;

DEFINE
  any := t9|t10|t29|t30;
--define the env sync event triggering macros for simple Hts blueAHts
DEFINE
  entRedA_trig:=0;
  exitRedA_trig:=0;
  entRedB_trig:=0;
  exitRedB_trig:=0;
  entBlueA_trig:=t9;
  exitBlueA_trig:=t10;
  entBlueB_trig:=0;
  exitBlueB_trig:=0;

--define the other triggering macro
--for the env sync events for simple Hts blueAHts
DEFINE
  env_other_trig:=t29|t30;

--define the rend sync event triggering macros for simple Hts blueAHts
DEFINE
  inRed_trig:=0;
  outRed_trig:=0;
  inBlue_trig:=0;
  outBlue_trig:=0;

--define the rend sync event generating macros for simple Hts blueAHts
```

```
  DEFINE
    inRed_gen:=0;
    outRed_gen:=0;
    inBlue_gen:=t29;
    outBlue_gen:=t30;

  --define the other triggering macro
  --for the rend sync event for simple Hts blueAHts
  DEFINE
    rend_other_trig:=t9|t10;


MODULE execute_blueAHts(en)
  --transition declaration
  VAR
    tran : {t9_exe, t10_exe, t29_exe, t30_exe, noTran_exe};

  --execution macros for all transitions
  DEFINE
    t9 := (tran=t9_exe);
    t10 := (tran=t10_exe);
    t29 := (tran=t29_exe);
    t30 := (tran=t30_exe);
    any := !(tran=noTran_exe);

  INVAR
     (t9 -> en.t9)
    &(t10 -> en.t10)
    &(t29 -> en.t29)
    &(t30 -> en.t30)
    &(any -> en.any)

  --define the env sync event triggering macros for simple Hts blueAHts
  DEFINE
    entRedA_trig:=0;
    exitRedA_trig:=0;
    entRedB_trig:=0;
    exitRedB_trig:=0;
    entBlueA_trig:=t9;
    exitBlueA_trig:=t10;
    entBlueB_trig:=0;
    exitBlueB_trig:=0;

  --define the other triggering macro
  --for the env sync events for simple Hts blueAHts
  DEFINE
    env_other_trig:=t29|t30;
```

```
  --blueAHts is inside rend sync operation,
  --define flag to check whether more than 1 transition are to execute
  DEFINE
    more_than_one:=0;

  --define the rend sync event triggering macros for simple Hts blueAHts
  DEFINE
    inRed_trig:=0;
    outRed_trig:=0;
    inBlue_trig:=0;
    outBlue_trig:=0;

  --define the rend sync event generating macros for simple Hts blueAHts
  DEFINE
    inRed_gen:=0;
    outRed_gen:=0;
    inBlue_gen:=t29;
    outBlue_gen:=t30;

  --define the other triggering macro
  --for the rend sync event for simple Hts blueAHts
  DEFINE
    rend_other_trig:=t9|t10;


-------------------------------------------------------------------------------

MODULE enabled_blueBHts(ss,rend_events)
--en_states: srcBlgssCS
--en_events: trigBlgssIa
--en_states: AVEntailCond
  --define enabled macros for transition t11
  DEFINE
    enStates_t11:=(ss.CS.in_waitBlueB);
    enEvents_t11:=(ss.Ia.entBlueB=1);
    enCond_t11:=(((ss.AV.numBlue) < (2)) & ((ss.AV.numRed) = (0)));
    ent11:=(enStates_t11)&(enEvents_t11)&(enCond_t11);

  --define enabled macros for transition t12
  DEFINE
    enStates_t12:=(ss.CS.in_onBlueB);
    enEvents_t12:=(ss.Ia.exitBlueB=1);
    enCond_t12 := 1;
    ent12:=(enStates_t12)&(enEvents_t12)&(enCond_t12);

  --define enabled macros for transition t31
  DEFINE
    enStates_t31:=(ss.CS.in_moveOnBlueB);
```

```
  enEvents_t31:= 1;
  enCond_t31 := 1;
  ent31:=(enStates_t31)&(enEvents_t31)&(enCond_t31);

--define enabled macros for transition t32
DEFINE
  enStates_t32:=(ss.CS.in_moveOutBlueB);
  enEvents_t32:= 1;
  enCond_t32 := 1;
  ent32:=(enStates_t32)&(enEvents_t32)&(enCond_t32);


--define enabled macros for transitions
  t11 := ent11;
  t12 := ent12;
  t31 := ent31;
  t32 := ent32;

DEFINE
  any := t11|t12|t31|t32;
--define the env sync event triggering macros for simple Hts blueBHts
DEFINE
  entRedA_trig:=0;
  exitRedA_trig:=0;
  entRedB_trig:=0;
  exitRedB_trig:=0;
  entBlueA_trig:=0;
  exitBlueA_trig:=0;
  entBlueB_trig:=t11;
  exitBlueB_trig:=t12;

--define the other triggering macro
--for the env sync events for simple Hts blueBHts
DEFINE
  env_other_trig:=t31|t32;

--define the rend sync event triggering macros for simple Hts blueBHts
DEFINE
  inRed_trig:=0;
  outRed_trig:=0;
  inBlue_trig:=0;
  outBlue_trig:=0;

--define the rend sync event generating macros for simple Hts blueBHts
DEFINE
  inRed_gen:=0;
  outRed_gen:=0;
  inBlue_gen:=t31;
```

```
    outBlue_gen:=t32;

  --define the other triggering macro
  --for the rend sync event for simple Hts blueBHts
  DEFINE
    rend_other_trig:=t11|t12;


MODULE execute_blueBHts(en)
  --transition declaration
  VAR
    tran : {t11_exe, t12_exe, t31_exe, t32_exe, noTran_exe};

  --execution macros for all transitions
  DEFINE
    t11 := (tran=t11_exe);
    t12 := (tran=t12_exe);
    t31 := (tran=t31_exe);
    t32 := (tran=t32_exe);
    any := !(tran=noTran_exe);

  INVAR
     (t11 -> en.t11)
    &(t12 -> en.t12)
    &(t31 -> en.t31)
    &(t32 -> en.t32)
    &(any -> en.any)

  --define the env sync event triggering macros for simple Hts blueBHts
  DEFINE
    entRedA_trig:=0;
    exitRedA_trig:=0;
    entRedB_trig:=0;
    exitRedB_trig:=0;
    entBlueA_trig:=0;
    exitBlueA_trig:=0;
    entBlueB_trig:=t11;
    exitBlueB_trig:=t12;

  --define the other triggering macro
  --for the env sync events for simple Hts blueBHts
  DEFINE
    env_other_trig:=t31|t32;

  --blueBHts is inside rend sync operation,
  --define flag to check whether more than 1 transition are to execute
  DEFINE
    more_than_one:=0;
```

```
  --define the rend sync event triggering macros for simple Hts blueBHts
  DEFINE
    inRed_trig:=0;
    outRed_trig:=0;
    inBlue_trig:=0;
    outBlue_trig:=0;

  --define the rend sync event generating macros for simple Hts blueBHts
  DEFINE
    inRed_gen:=0;
    outRed_gen:=0;
    inBlue_gen:=t31;
    outBlue_gen:=t32;

  --define the other triggering macro
  --for the rend sync event for simple Hts blueBHts
  DEFINE
    rend_other_trig:=t11|t12;


--------------------------------------------------------------------------------

MODULE enabled_bridgeStatusHts(ss,rend_events)
--en_states: srcBlgssCS
--en_events: trigBlgssIa
--en_states: AVEntailCond
  --define enabled macros for transition t17
  DEFINE
    enStates_t17:=(ss.CS.in_empty);
    enEvents_t17:= 1;
    enCond_t17 := 1;
    ent17:=(enStates_t17)&(enEvents_t17)&(enCond_t17);

  --define enabled macros for transition t18
  DEFINE
    enStates_t18:=(ss.CS.in_oneRed);
    enEvents_t18:= 1;
    enCond_t18 := 1;
    ent18:=(enStates_t18)&(enEvents_t18)&(enCond_t18);

  --define enabled macros for transition t19
  DEFINE
    enStates_t19:=(ss.CS.in_twoRed);
    enEvents_t19:= 1;
    enCond_t19 := 1;
    ent19:=(enStates_t19)&(enEvents_t19)&(enCond_t19);
```

```
--define enabled macros for transition t20
DEFINE
  enStates_t20:=(ss.CS.in_oneRed);
  enEvents_t20:= 1;
  enCond_t20 := 1;
  ent20:=(enStates_t20)&(enEvents_t20)&(enCond_t20);

--define enabled macros for transition t21
DEFINE
  enStates_t21:=(ss.CS.in_empty);
  enEvents_t21:= 1;
  enCond_t21 := 1;
  ent21:=(enStates_t21)&(enEvents_t21)&(enCond_t21);

--define enabled macros for transition t22
DEFINE
  enStates_t22:=(ss.CS.in_oneBlue);
  enEvents_t22:= 1;
  enCond_t22 := 1;
  ent22:=(enStates_t22)&(enEvents_t22)&(enCond_t22);

--define enabled macros for transition t23
DEFINE
  enStates_t23:=(ss.CS.in_twoBlue);
  enEvents_t23:= 1;
  enCond_t23 := 1;
  ent23:=(enStates_t23)&(enEvents_t23)&(enCond_t23);

--define enabled macros for transition t24
DEFINE
  enStates_t24:=(ss.CS.in_oneBlue);
  enEvents_t24:= 1;
  enCond_t24 := 1;
  ent24:=(enStates_t24)&(enEvents_t24)&(enCond_t24);


--define enabled macros for transitions
  t17 := ent17;
  t18 := ent18;
  t19 := ent19;
  t20 := ent20;
  t21 := ent21;
  t22 := ent22;
  t23 := ent23;
  t24 := ent24;

DEFINE
  any := t17|t18|t19|t20|t21|t22|t23|t24;
```

```
  --define the env sync event triggering macros for simple Hts bridgeStatusHts
  DEFINE
    entRedA_trig:=0;
    exitRedA_trig:=0;
    entRedB_trig:=0;
    exitRedB_trig:=0;
    entBlueA_trig:=0;
    exitBlueA_trig:=0;
    entBlueB_trig:=0;
    exitBlueB_trig:=0;

  --define the other triggering macro
  --for the env sync events for simple Hts bridgeStatusHts
  DEFINE
    env_other_trig:=t17|t18|t19|t20|t21|t22|t23|t24;

  --define the rend sync event triggering macros for simple Hts bridgeStatusHts
  DEFINE
    inRed_trig:=t17|t18;
    outRed_trig:=t19|t20;
    inBlue_trig:=t21|t22;
    outBlue_trig:=t23|t24;

  --define the rend sync event generating macros for simple Hts bridgeStatusHts
  DEFINE
    inRed_gen:=0;
    outRed_gen:=0;
    inBlue_gen:=0;
    outBlue_gen:=0;

  --define the other triggering macro
  --for the rend sync event for simple Hts bridgeStatusHts
  DEFINE
    rend_other_trig:=0;


MODULE execute_bridgeStatusHts(en)
  --transition declaration
  VAR
    tran : {t17_exe, t18_exe, t19_exe, t20_exe,
            t21_exe, t22_exe, t23_exe, t24_exe, noTran_exe};

  --execution macros for all transitions
  DEFINE
    t17 := (tran=t17_exe);
    t18 := (tran=t18_exe);
    t19 := (tran=t19_exe);
    t20 := (tran=t20_exe);
```

```
  t21 := (tran=t21_exe);
  t22 := (tran=t22_exe);
  t23 := (tran=t23_exe);
  t24 := (tran=t24_exe);
  any := !(tran=noTran_exe);

INVAR
   (t17 -> en.t17)
  &(t18 -> en.t18)
  &(t19 -> en.t19)
  &(t20 -> en.t20)
  &(t21 -> en.t21)
  &(t22 -> en.t22)
  &(t23 -> en.t23)
  &(t24 -> en.t24)
  &(any -> en.any)

--define the env sync event triggering macros for simple Hts bridgeStatusHts
DEFINE
  entRedA_trig:=0;
  exitRedA_trig:=0;
  entRedB_trig:=0;
  exitRedB_trig:=0;
  entBlueA_trig:=0;
  exitBlueA_trig:=0;
  entBlueB_trig:=0;
  exitBlueB_trig:=0;

--define the other triggering macro
--for the env sync events for simple Hts bridgeStatusHts
DEFINE
  env_other_trig:=t17|t18|t19|t20|t21|t22|t23|t24;

--bridgeStatusHts is inside rend sync operation,
--define flag to check whether more than 1 transition are to execute
DEFINE
  more_than_one:=0;

--define the rend sync event triggering macros for simple Hts bridgeStatusHts
DEFINE
  inRed_trig:=t17|t18;
  outRed_trig:=t19|t20;
  inBlue_trig:=t21|t22;
  outBlue_trig:=t23|t24;

--define the rend sync event generating macros for simple Hts bridgeStatusHts
DEFINE
  inRed_gen:=0;
```

```
    outRed_gen:=0;
    inBlue_gen:=0;
    outBlue_gen:=0;

  --define the other triggering macro
  --for the rend sync event for simple Hts bridgeStatusHts
  DEFINE
    rend_other_trig:=0;


--------------------------------------------------------------------------------

MODULE enabled_redCoordEntHts(ss)
--en_states: srcBlgssCS
--en_events: trigBlgssIa
--en_states: AVEntailCond
  --define enabled macros for transition t5
  DEFINE
    enStates_t5:=(ss.CS.in_coordEntRedA);
    enEvents_t5:=(ss.Ia.entRedA=1);
    enCond_t5 := 1;
    ent5:=(enStates_t5)&(enEvents_t5)&(enCond_t5);

  --define enabled macros for transition t6
  DEFINE
    enStates_t6:=(ss.CS.in_coordEntRedB);
    enEvents_t6:=(ss.Ia.entRedB=1);
    enCond_t6 := 1;
    ent6:=(enStates_t6)&(enEvents_t6)&(enCond_t6);


  --define enabled macros for transitions
    t5 := ent5;
    t6 := ent6;

  DEFINE
    any := t5|t6;
  --define the env sync event triggering macros for simple Hts redCoordEntHts
  DEFINE
    entRedA_trig:=t5;
    exitRedA_trig:=0;
    entRedB_trig:=t6;
    exitRedB_trig:=0;
    entBlueA_trig:=0;
    exitBlueA_trig:=0;
    entBlueB_trig:=0;
    exitBlueB_trig:=0;
```

```
  --define the other triggering macro
  --for the env sync events for simple Hts redCoordEntHts
  DEFINE
    env_other_trig:=0;


MODULE execute_redCoordEntHts(en)
  --transition declaration
  VAR
    tran : {t5_exe, t6_exe, noTran_exe};

  --execution macros for all transitions
  DEFINE
    t5 := (tran=t5_exe);
    t6 := (tran=t6_exe);
    any := !(tran=noTran_exe);

  INVAR
     (t5 -> en.t5)
    &(t6 -> en.t6)
    &(any -> en.any)

  --define the env sync event triggering macros for simple Hts redCoordEntHts
  DEFINE
    entRedA_trig:=t5;
    exitRedA_trig:=0;
    entRedB_trig:=t6;
    exitRedB_trig:=0;
    entBlueA_trig:=0;
    exitBlueA_trig:=0;
    entBlueB_trig:=0;
    exitBlueB_trig:=0;

  --define the other triggering macro
  --for the env sync events for simple Hts redCoordEntHts
  DEFINE
    env_other_trig:=0;


--------------------------------------------------------------------------------

MODULE enabled_redCoordExitHts(ss)
--en_states: srcBlgssCS
--en_events: trigBlgssIa
--en_states: AVEntailCond
  --define enabled macros for transition t7
  DEFINE
    enStates_t7:=(ss.CS.in_coordExitRedA);
```

```
    enEvents_t7:=(ss.Ia.exitRedA=1);
    enCond_t7 := 1;
    ent7:=(enStates_t7)&(enEvents_t7)&(enCond_t7);

  --define enabled macros for transition t8
  DEFINE
    enStates_t8:=(ss.CS.in_coordExitRedB);
    enEvents_t8:=(ss.Ia.exitRedB=1);
    enCond_t8 := 1;
    ent8:=(enStates_t8)&(enEvents_t8)&(enCond_t8);


  --define enabled macros for transitions
    t7 := ent7;
    t8 := ent8;

  DEFINE
    any := t7|t8;
  --define the env sync event triggering macros for simple Hts redCoordExitHts
  DEFINE
    entRedA_trig:=0;
    exitRedA_trig:=t7;
    entRedB_trig:=0;
    exitRedB_trig:=t8;
    entBlueA_trig:=0;
    exitBlueA_trig:=0;
    entBlueB_trig:=0;
    exitBlueB_trig:=0;

  --define the other triggering macro
  --for the env sync events for simple Hts redCoordExitHts
  DEFINE
    env_other_trig:=0;


MODULE execute_redCoordExitHts(en)
  --transition declaration
  VAR
    tran : {t7_exe, t8_exe, noTran_exe};

  --execution macros for all transitions
  DEFINE
    t7 := (tran=t7_exe);
    t8 := (tran=t8_exe);
    any := !(tran=noTran_exe);

  INVAR
     (t7 -> en.t7)
```

```
    &(t8 -> en.t8)
    &(any -> en.any)

  --define the env sync event triggering macros for simple Hts redCoordExitHts
  DEFINE
    entRedA_trig:=0;
    exitRedA_trig:=t7;
    entRedB_trig:=0;
    exitRedB_trig:=t8;
    entBlueA_trig:=0;
    exitBlueA_trig:=0;
    entBlueB_trig:=0;
    exitBlueB_trig:=0;

  --define the other triggering macro
  --for the env sync events for simple Hts redCoordExitHts
  DEFINE
    env_other_trig:=0;


--------------------------------------------------------------------------------

MODULE enabled_blueCoordEntHts(ss)
--en_states: srcBlgssCS
--en_events: trigBlgssIa
--en_states: AVEntailCond
  --define enabled macros for transition t13
  DEFINE
    enStates_t13:=(ss.CS.in_coordEntBlueA);
    enEvents_t13:=(ss.Ia.entBlueA=1);
    enCond_t13 := 1;
    ent13:=(enStates_t13)&(enEvents_t13)&(enCond_t13);

  --define enabled macros for transition t14
  DEFINE
    enStates_t14:=(ss.CS.in_coordEntBlueB);
    enEvents_t14:=(ss.Ia.entBlueB=1);
    enCond_t14 := 1;
    ent14:=(enStates_t14)&(enEvents_t14)&(enCond_t14);


  --define enabled macros for transitions
    t13 := ent13;
    t14 := ent14;

  DEFINE
    any := t13|t14;
  --define the env sync event triggering macros for simple Hts blueCoordEntHts
```

```
  DEFINE
    entRedA_trig:=0;
    exitRedA_trig:=0;
    entRedB_trig:=0;
    exitRedB_trig:=0;
    entBlueA_trig:=t13;
    exitBlueA_trig:=0;
    entBlueB_trig:=t14;
    exitBlueB_trig:=0;

  --define the other triggering macro
  --for the env sync events for simple Hts blueCoordEntHts
  DEFINE
    env_other_trig:=0;


MODULE execute_blueCoordEntHts(en)
  --transition declaration
  VAR
    tran : {t13_exe, t14_exe, noTran_exe};

  --execution macros for all transitions
  DEFINE
    t13 := (tran=t13_exe);
    t14 := (tran=t14_exe);
    any := !(tran=noTran_exe);

  INVAR
     (t13 -> en.t13)
    &(t14 -> en.t14)
    &(any -> en.any)

  --define the env sync event triggering macros for simple Hts blueCoordEntHts
  DEFINE
    entRedA_trig:=0;
    exitRedA_trig:=0;
    entRedB_trig:=0;
    exitRedB_trig:=0;
    entBlueA_trig:=t13;
    exitBlueA_trig:=0;
    entBlueB_trig:=t14;
    exitBlueB_trig:=0;

  --define the other triggering macro
  --for the env sync events for simple Hts blueCoordEntHts
  DEFINE
    env_other_trig:=0;
```

```
-------------------------------------------------------------------------------

MODULE enabled_blueCoordExitHts(ss)
--en_states: srcBlgssCS
--en_events: trigBlgssIa
--en_states: AVEntailCond
  --define enabled macros for transition t15
  DEFINE
    enStates_t15:=(ss.CS.in_coordExitBlueA);
    enEvents_t15:=(ss.Ia.exitBlueA=1);
    enCond_t15 := 1;
    ent15:=(enStates_t15)&(enEvents_t15)&(enCond_t15);

  --define enabled macros for transition t16
  DEFINE
    enStates_t16:=(ss.CS.in_coordExitBlueB);
    enEvents_t16:=(ss.Ia.exitBlueB=1);
    enCond_t16 := 1;
    ent16:=(enStates_t16)&(enEvents_t16)&(enCond_t16);


  --define enabled macros for transitions
    t15 := ent15;
    t16 := ent16;

  DEFINE
    any := t15|t16;
  --define the env sync event triggering macros for simple Hts blueCoordExitHts
  DEFINE
    entRedA_trig:=0;
    exitRedA_trig:=0;
    entRedB_trig:=0;
    exitRedB_trig:=0;
    entBlueA_trig:=0;
    exitBlueA_trig:=t15;
    entBlueB_trig:=0;
    exitBlueB_trig:=t16;

  --define the other triggering macro
  --for the env sync events for simple Hts blueCoordExitHts
  DEFINE
    env_other_trig:=0;


MODULE execute_blueCoordExitHts(en)
  --transition declaration
  VAR
```

```
    tran : {t15_exe, t16_exe, noTran_exe};

  --execution macros for all transitions
  DEFINE
    t15 := (tran=t15_exe);
    t16 := (tran=t16_exe);
    any := !(tran=noTran_exe);

  INVAR
     (t15 -> en.t15)
    &(t16 -> en.t16)
    &(any -> en.any)

  --define the env sync event triggering macros for simple Hts blueCoordExitHts
  DEFINE
    entRedA_trig:=0;
    exitRedA_trig:=0;
    entRedB_trig:=0;
    exitRedB_trig:=0;
    entBlueA_trig:=0;
    exitBlueA_trig:=t15;
    entBlueB_trig:=0;
    exitBlueB_trig:=t16;

  --define the other triggering macro
  --for the env sync events for simple Hts blueCoordExitHts
  DEFINE
    env_other_trig:=0;


--------------------------------------------------------------------------------

--------------------------------------------------------------------------

MODULE envsync_entRedA_exitRedA_entRedB_exitRedB
        _entBlueA_exitBlueA_entBlueB_exitBlueB(enLeft,enRight,exeLeft,exeRight)
  DEFINE
    any:=exeLeft.any | exeRight.any ;
    entRedA_trig := exeLeft.entRedA_trig & exeRight.entRedA_trig;
    exitRedA_trig := exeLeft.exitRedA_trig & exeRight.exitRedA_trig;
    entRedB_trig := exeLeft.entRedB_trig & exeRight.entRedB_trig;
    exitRedB_trig := exeLeft.exitRedB_trig & exeRight.exitRedB_trig;
    entBlueA_trig := exeLeft.entBlueA_trig & exeRight.entBlueA_trig;
    exitBlueA_trig := exeLeft.exitBlueA_trig & exeRight.exitBlueA_trig;
    entBlueB_trig := exeLeft.entBlueB_trig & exeRight.entBlueB_trig;
    exitBlueB_trig := exeLeft.exitBlueB_trig & exeRight.exitBlueB_trig;
    env_other_trig := exeLeft.env_other_trig | exeRight.env_other_trig;
  INVAR
```

```
     (!(entRedA_trig|exitRedA_trig|entRedB_trig|exitRedB_trig
        |entBlueA_trig|exitBlueA_trig|entBlueB_trig|exitBlueB_trig)
       -> !(exeLeft.any & exeRight.any))
    &((entRedA_trig|exitRedA_trig|entRedB_trig|exitRedB_trig
        |entBlueA_trig|exitBlueA_trig|entBlueB_trig|exitBlueB_trig)
      -> !env_other_trig)
    &(exeLeft.entRedA_trig <-> exeRight.entRedA_trig)
    &(exeLeft.exitRedA_trig <-> exeRight.exitRedA_trig)
    &(exeLeft.entRedB_trig <-> exeRight.entRedB_trig)
    &(exeLeft.exitRedB_trig <-> exeRight.exitRedB_trig)
    &(exeLeft.entBlueA_trig <-> exeRight.entBlueA_trig)
    &(exeLeft.exitBlueA_trig <-> exeRight.exitBlueA_trig)
    &(exeLeft.entBlueB_trig <-> exeRight.entBlueB_trig)
    &(exeLeft.exitBlueB_trig <-> exeRight.exitBlueB_trig)
    &!(exeLeft.entRedA_trig & exeLeft.exitRedA_trig)
    &!(exeLeft.entRedA_trig & exeLeft.entRedB_trig)
    &!(exeLeft.entRedA_trig & exeLeft.exitRedB_trig)
    &!(exeLeft.entRedA_trig & exeLeft.entBlueA_trig)
    &!(exeLeft.entRedA_trig & exeLeft.exitBlueA_trig)
    &!(exeLeft.entRedA_trig & exeLeft.entBlueB_trig)
    &!(exeLeft.entRedA_trig & exeLeft.exitBlueB_trig)
    &!(exeLeft.exitRedA_trig & exeLeft.entRedB_trig)
    &!(exeLeft.exitRedA_trig & exeLeft.exitRedB_trig)
    &!(exeLeft.exitRedA_trig & exeLeft.entBlueA_trig)
    &!(exeLeft.exitRedA_trig & exeLeft.exitBlueA_trig)
    &!(exeLeft.exitRedA_trig & exeLeft.entBlueB_trig)
    &!(exeLeft.exitRedA_trig & exeLeft.exitBlueB_trig)
    &!(exeLeft.entRedB_trig & exeLeft.exitRedB_trig)
    &!(exeLeft.entRedB_trig & exeLeft.entBlueA_trig)
    &!(exeLeft.entRedB_trig & exeLeft.exitBlueA_trig)
    &!(exeLeft.entRedB_trig & exeLeft.entBlueB_trig)
    &!(exeLeft.entRedB_trig & exeLeft.exitBlueB_trig)
    &!(exeLeft.exitRedB_trig & exeLeft.entBlueA_trig)
    &!(exeLeft.exitRedB_trig & exeLeft.exitBlueA_trig)
    &!(exeLeft.exitRedB_trig & exeLeft.entBlueB_trig)
    &!(exeLeft.exitRedB_trig & exeLeft.exitBlueB_trig)
    &!(exeLeft.entBlueA_trig & exeLeft.exitBlueA_trig)
    &!(exeLeft.entBlueA_trig & exeLeft.entBlueB_trig)
    &!(exeLeft.entBlueA_trig & exeLeft.exitBlueB_trig)
    &!(exeLeft.exitBlueA_trig & exeLeft.entBlueB_trig)
    &!(exeLeft.exitBlueA_trig & exeLeft.exitBlueB_trig)
    &!(exeLeft.entBlueB_trig & exeLeft.exitBlueB_trig)

MODULE rendezvous_inRed_outRed_inBlue_outBlue(enLeft,enRight,exeLeft,exeRight)
  DEFINE
    any:=exeLeft.any | exeRight.any ;
    inRed_rend:=  (exeLeft.inRed_trig&exeRight.inRed_gen)
                 |(exeLeft.inRed_gen&exeRight.inRed_trig);
```

```
    outRed_rend:= (exeLeft.outRed_trig&exeRight.outRed_gen)
                 |(exeLeft.outRed_gen&exeRight.outRed_trig);
    inBlue_rend:= (exeLeft.inBlue_trig&exeRight.inBlue_gen)
                 |(exeLeft.inBlue_gen&exeRight.inBlue_trig);
    outBlue_rend:=(exeLeft.outBlue_trig&exeRight.outBlue_gen)
                 |(exeLeft.outBlue_gen&exeRight.outBlue_trig);
  INVAR
     (! (inRed_rend|outRed_rend|inBlue_rend|outBlue_rend)
       -> !(exeLeft.any & exeRight.any))
    &((inRed_rend|outRed_rend|inBlue_rend|outBlue_rend)
      -> !(exeLeft.more_than_one | exeRight.more_than_one))
    &(exeLeft.inRed_trig <-> exeRight.inRed_gen)
    &(exeLeft.inRed_gen <-> exeRight.inRed_trig)
    &(exeLeft.outRed_trig <-> exeRight.outRed_gen)
    &(exeLeft.outRed_gen <-> exeRight.outRed_trig)
    &(exeLeft.inBlue_trig <-> exeRight.inBlue_gen)
    &(exeLeft.inBlue_gen <-> exeRight.inBlue_trig)
    &(exeLeft.outBlue_trig <-> exeRight.outBlue_gen)
    &(exeLeft.outBlue_gen <-> exeRight.outBlue_trig)

MODULE interleaving(enLeft,enRight,exeLeft,exeRight)
  DEFINE
    any:=exeLeft.any | exeRight.any ;
  INVAR
    !(exeLeft.any & exeRight.any)

-----------------------------------------------------------------------------

MODULE reset(ss,I)
  VAR
    CS : resetCS(ss);
    AV : resetAV(ss,I);
    OO : resetO(ss);
    Ia : resetIa(ss,I);

MODULE resetCS(ss)
--in simple macro semantics
--resetCS : ssCS
  DEFINE redAHts_state := ss.CS.redAHts_state;
  DEFINE redBHts_state := ss.CS.redBHts_state;
  DEFINE blueAHts_state := ss.CS.blueAHts_state;
  DEFINE blueBHts_state := ss.CS.blueBHts_state;
  DEFINE bridgeStatusHts_state := ss.CS.bridgeStatusHts_state;
  DEFINE redCoordEntHts_state := ss.CS.redCoordEntHts_state;
  DEFINE redCoordExitHts_state := ss.CS.redCoordExitHts_state;
  DEFINE blueCoordEntHts_state := ss.CS.blueCoordEntHts_state;
  DEFINE blueCoordExitHts_state := ss.CS.blueCoordExitHts_state;
  --define macros for all states
```

```
DEFINE
  in_singleLaneBridge := in_bridge | in_coord;
  in_bridge := in_car | in_bridgeStatus;
  in_car := in_redCar | in_blueCar;
  in_redCar := in_redA | in_redB;
  in_redA := in_waitRedA | in_moveOnRedA | in_moveOutRedA | in_onRedA;
  in_waitRedA := redAHts_state=waitRedA;
  in_moveOnRedA := redAHts_state=moveOnRedA;
  in_moveOutRedA := redAHts_state=moveOutRedA;
  in_onRedA := redAHts_state=onRedA;
  in_redB := in_waitRedB | in_moveOnRedB | in_moveOutRedB | in_onRedB;
  in_waitRedB := redBHts_state=waitRedB;
  in_moveOnRedB := redBHts_state=moveOnRedB;
  in_moveOutRedB := redBHts_state=moveOutRedB;
  in_onRedB := redBHts_state=onRedB;
  in_blueCar := in_blueA | in_blueB;
  in_blueA := in_waitBlueA | in_moveOnBlueA | in_moveOutBlueA | in_onBlueA;
  in_waitBlueA := blueAHts_state=waitBlueA;
  in_moveOnBlueA := blueAHts_state=moveOnBlueA;
  in_moveOutBlueA := blueAHts_state=moveOutBlueA;
  in_onBlueA := blueAHts_state=onBlueA;
  in_blueB := in_waitBlueB | in_moveOnBlueB | in_moveOutBlueB | in_onBlueB;
  in_waitBlueB := blueBHts_state=waitBlueB;
  in_moveOnBlueB := blueBHts_state=moveOnBlueB;
  in_moveOutBlueB := blueBHts_state=moveOutBlueB;
  in_onBlueB := blueBHts_state=onBlueB;
  in_bridgeStatus := in_empty | in_oneRed | in_twoRed | in_oneBlue | in_twoBlue;
  in_empty := bridgeStatusHts_state=empty;
  in_oneRed := bridgeStatusHts_state=oneRed;
  in_twoRed := bridgeStatusHts_state=twoRed;
  in_oneBlue := bridgeStatusHts_state=oneBlue;
  in_twoBlue := bridgeStatusHts_state=twoBlue;
  in_coord := in_coordRed | in_coordBlue;
  in_coordRed := in_redCoordEnt | in_redCoordExit;
  in_redCoordEnt := in_coordEntRedA | in_coordEntRedB;
  in_coordEntRedA := redCoordEntHts_state=coordEntRedA;
  in_coordEntRedB := redCoordEntHts_state=coordEntRedB;
  in_redCoordExit := in_coordExitRedA | in_coordExitRedB;
  in_coordExitRedA := redCoordExitHts_state=coordExitRedA;
  in_coordExitRedB := redCoordExitHts_state=coordExitRedB;
  in_coordBlue := in_blueCoordEnt | in_blueCoordExit;
  in_blueCoordEnt := in_coordEntBlueA | in_coordEntBlueB;
  in_coordEntBlueA := blueCoordEntHts_state=coordEntBlueA;
  in_coordEntBlueB := blueCoordEntHts_state=coordEntBlueB;
  in_blueCoordExit := in_coordExitBlueA | in_coordExitBlueB;
  in_coordExitBlueA := blueCoordExitHts_state=coordExitBlueA;
  in_coordExitBlueB := blueCoordExitHts_state=coordExitBlueB;
```

```
MODULE resetAV(ss,I)
--resetAV : ssAV
DEFINE numRed := ss.AV.numRed;
DEFINE numBlue := ss.AV.numBlue;
DEFINE error_variables := ss.AV.error_variables;

MODULE resetO(ss)
--resetO : resetOEPT
  DEFINE inRed := 0;
  DEFINE outRed := 0;
  DEFINE inBlue := 0;
  DEFINE outBlue := 0;

MODULE resetIa(ss,I)
--resetIa : resetIaI
DEFINE entRedA := I.ev.entRedA;
DEFINE exitRedA := I.ev.exitRedA;
DEFINE entRedB := I.ev.entRedB;
DEFINE exitRedB := I.ev.exitRedB;
DEFINE entBlueA := I.ev.entBlueA;
DEFINE exitBlueA := I.ev.exitBlueA;
DEFINE entBlueB := I.ev.entBlueB;
DEFINE exitBlueB := I.ev.exitBlueB;



--------------------------------------------------------------------------

MODULE enabled(ss)
  VAR
    redAHts : enabled_redAHts(ss,rend_events);
    redBHts : enabled_redBHts(ss,rend_events);
    blueAHts : enabled_blueAHts(ss,rend_events);
    blueBHts : enabled_blueBHts(ss,rend_events);
    bridgeStatusHts : enabled_bridgeStatusHts(ss,rend_events);
    redCoordEntHts : enabled_redCoordEntHts(ss);
    redCoordExitHts : enabled_redCoordExitHts(ss);
    blueCoordEntHts : enabled_blueCoordEntHts(ss);
    blueCoordExitHts : enabled_blueCoordExitHts(ss);

  DEFINE
  --extra macro for rendzevous event inRed
    rend_events.inRed := redAHts.t25;
  --extra macro for rendzevous event outRed
    rend_events.outRed := redAHts.t26;
  --extra macro for rendzevous event inBlue
    rend_events.inBlue := redBHts.t27 | blueAHts.t29 | blueBHts.t31;
  --extra macro for rendzevous event outBlue
```

```
    rend_events.outBlue := redBHts.t28 | blueAHts.t30 | blueBHts.t32;

  --define enabled macros for each composite HTSs
  DEFINE
  --define en.singleLaneBridgeHts.any that use composition envsync
    singleLaneBridgeHts.any := bridgeHts.entRedA_trig&coordHts.entRedA_trig
 | bridgeHts.exitRedA_trig&coordHts.exitRedA_trig
 | bridgeHts.entRedB_trig&coordHts.entRedB_trig
 | bridgeHts.exitRedB_trig&coordHts.exitRedB_trig
 | bridgeHts.entBlueA_trig&coordHts.entBlueA_trig
 | bridgeHts.exitBlueA_trig&coordHts.exitBlueA_trig
 | bridgeHts.entBlueB_trig&coordHts.entBlueB_trig
 | bridgeHts.exitBlueB_trig&coordHts.exitBlueB_trig
 | bridgeHts.env_other_trig
 | coordHts.env_other_trig;

  --define en.bridgeHts.any that use composition rendezvous
    bridgeHts.any :=  carHts.inRed_trig&bridgeStatusHts.inRed_gen
                    | carHts.inRed_gen&bridgeStatusHts.inRed_trig
            | carHts.outRed_trig&bridgeStatusHts.outRed_gen
                    | carHts.outRed_gen&bridgeStatusHts.outRed_trig
            | carHts.inBlue_trig&bridgeStatusHts.inBlue_gen
                    | carHts.inBlue_gen&bridgeStatusHts.inBlue_trig
            | carHts.outBlue_trig&bridgeStatusHts.outBlue_gen
                    | carHts.outBlue_gen&bridgeStatusHts.outBlue_trig
            | carHts.rend_other_trig
            | bridgeStatusHts.rend_other_trig;
  --defining macro flag for environmental sync events in bridgeHts
  DEFINE
    bridgeHts.env_other_trig :=  carHts.env_other_trig
                               | bridgeStatusHts.env_other_trig;
    bridgeHts.entRedA_trig :=  carHts.entRedA_trig
                             | bridgeStatusHts.entRedA_trig;
    bridgeHts.exitRedA_trig := carHts.exitRedA_trig
                             | bridgeStatusHts.exitRedA_trig;
    bridgeHts.entRedB_trig := carHts.entRedB_trig
                            | bridgeStatusHts.entRedB_trig;
    bridgeHts.exitRedB_trig := carHts.exitRedB_trig
                             | bridgeStatusHts.exitRedB_trig;
    bridgeHts.entBlueA_trig := carHts.entBlueA_trig
                             | bridgeStatusHts.entBlueA_trig;
    bridgeHts.exitBlueA_trig := carHts.exitBlueA_trig
                              | bridgeStatusHts.exitBlueA_trig;
    bridgeHts.entBlueB_trig := carHts.entBlueB_trig
                             | bridgeStatusHts.entBlueB_trig;
    bridgeHts.exitBlueB_trig := carHts.exitBlueB_trig
                              | bridgeStatusHts.exitBlueB_trig;
```

```
--define en.carHts.any that use composition interleaving
  carHts.any := redCarHts.any | blueCarHts.any;
--defining macro flag for environmental sync events in carHts
DEFINE
  carHts.env_other_trig := redCarHts.env_other_trig | blueCarHts.env_other_trig;
  carHts.entRedA_trig := redCarHts.entRedA_trig | blueCarHts.entRedA_trig;
  carHts.exitRedA_trig := redCarHts.exitRedA_trig | blueCarHts.exitRedA_trig;
  carHts.entRedB_trig := redCarHts.entRedB_trig | blueCarHts.entRedB_trig;
  carHts.exitRedB_trig := redCarHts.exitRedB_trig | blueCarHts.exitRedB_trig;
  carHts.entBlueA_trig := redCarHts.entBlueA_trig | blueCarHts.entBlueA_trig;
  carHts.exitBlueA_trig := redCarHts.exitBlueA_trig | blueCarHts.exitBlueA_trig;
  carHts.entBlueB_trig := redCarHts.entBlueB_trig | blueCarHts.entBlueB_trig;
  carHts.exitBlueB_trig := redCarHts.exitBlueB_trig | blueCarHts.exitBlueB_trig;
--defining macro flag for rendezvous events in carHts
DEFINE
  carHts.rend_other_trig := redCarHts.rend_other_trig | blueCarHts.rend_other_trig;
  carHts.inRed_trig := redCarHts.inRed_trig | blueCarHts.inRed_trig;
  carHts.inRed_gen := redCarHts.inRed_gen | blueCarHts.inRed_gen;
  carHts.outRed_trig := redCarHts.outRed_trig | blueCarHts.outRed_trig;
  carHts.outRed_gen := redCarHts.outRed_gen | blueCarHts.outRed_gen;
  carHts.inBlue_trig := redCarHts.inBlue_trig | blueCarHts.inBlue_trig;
  carHts.inBlue_gen := redCarHts.inBlue_gen | blueCarHts.inBlue_gen;
  carHts.outBlue_trig := redCarHts.outBlue_trig | blueCarHts.outBlue_trig;
  carHts.outBlue_gen := redCarHts.outBlue_gen | blueCarHts.outBlue_gen;

--define en.redCarHts.any that use composition interleaving
  redCarHts.any := redAHts.any | redBHts.any;
--defining macro flag for environmental sync events in redCarHts
DEFINE
  redCarHts.env_other_trig := redAHts.env_other_trig | redBHts.env_other_trig;
  redCarHts.entRedA_trig := redAHts.entRedA_trig | redBHts.entRedA_trig;
  redCarHts.exitRedA_trig := redAHts.exitRedA_trig | redBHts.exitRedA_trig;
  redCarHts.entRedB_trig := redAHts.entRedB_trig | redBHts.entRedB_trig;
  redCarHts.exitRedB_trig := redAHts.exitRedB_trig | redBHts.exitRedB_trig;
  redCarHts.entBlueA_trig := redAHts.entBlueA_trig | redBHts.entBlueA_trig;
  redCarHts.exitBlueA_trig := redAHts.exitBlueA_trig | redBHts.exitBlueA_trig;
  redCarHts.entBlueB_trig := redAHts.entBlueB_trig | redBHts.entBlueB_trig;
  redCarHts.exitBlueB_trig := redAHts.exitBlueB_trig | redBHts.exitBlueB_trig;
--defining macro flag for rendezvous events in redCarHts
DEFINE
  redCarHts.rend_other_trig := redAHts.rend_other_trig | redBHts.rend_other_trig;
  redCarHts.inRed_trig := redAHts.inRed_trig | redBHts.inRed_trig;
  redCarHts.inRed_gen := redAHts.inRed_gen | redBHts.inRed_gen;
  redCarHts.outRed_trig := redAHts.outRed_trig | redBHts.outRed_trig;
  redCarHts.outRed_gen := redAHts.outRed_gen | redBHts.outRed_gen;
  redCarHts.inBlue_trig := redAHts.inBlue_trig | redBHts.inBlue_trig;
  redCarHts.inBlue_gen := redAHts.inBlue_gen | redBHts.inBlue_gen;
  redCarHts.outBlue_trig := redAHts.outBlue_trig | redBHts.outBlue_trig;
```

```
  redCarHts.outBlue_gen := redAHts.outBlue_gen | redBHts.outBlue_gen;

--define en.blueCarHts.any that use composition interleaving
  blueCarHts.any := blueAHts.any | blueBHts.any;
--defining macro flag for environmental sync events in blueCarHts
DEFINE
  blueCarHts.env_other_trig := blueAHts.env_other_trig | blueBHts.env_other_trig;
  blueCarHts.entRedA_trig := blueAHts.entRedA_trig | blueBHts.entRedA_trig;
  blueCarHts.exitRedA_trig := blueAHts.exitRedA_trig | blueBHts.exitRedA_trig;
  blueCarHts.entRedB_trig := blueAHts.entRedB_trig | blueBHts.entRedB_trig;
  blueCarHts.exitRedB_trig := blueAHts.exitRedB_trig | blueBHts.exitRedB_trig;
  blueCarHts.entBlueA_trig := blueAHts.entBlueA_trig | blueBHts.entBlueA_trig;
  blueCarHts.exitBlueA_trig := blueAHts.exitBlueA_trig | blueBHts.exitBlueA_trig;
  blueCarHts.entBlueB_trig := blueAHts.entBlueB_trig | blueBHts.entBlueB_trig;
  blueCarHts.exitBlueB_trig := blueAHts.exitBlueB_trig | blueBHts.exitBlueB_trig;
--defining macro flag for rendezvous events in blueCarHts
DEFINE
  blueCarHts.rend_other_trig := blueAHts.rend_other_trig | blueBHts.rend_other_trig;
  blueCarHts.inRed_trig := blueAHts.inRed_trig | blueBHts.inRed_trig;
  blueCarHts.inRed_gen := blueAHts.inRed_gen | blueBHts.inRed_gen;
  blueCarHts.outRed_trig := blueAHts.outRed_trig | blueBHts.outRed_trig;
  blueCarHts.outRed_gen := blueAHts.outRed_gen | blueBHts.outRed_gen;
  blueCarHts.inBlue_trig := blueAHts.inBlue_trig | blueBHts.inBlue_trig;
  blueCarHts.inBlue_gen := blueAHts.inBlue_gen | blueBHts.inBlue_gen;
  blueCarHts.outBlue_trig := blueAHts.outBlue_trig | blueBHts.outBlue_trig;
  blueCarHts.outBlue_gen := blueAHts.outBlue_gen | blueBHts.outBlue_gen;

--define en.coordHts.any that use composition interleaving
  coordHts.any := coordRedHts.any | coordBlueHts.any;
--defining macro flag for environmental sync events in coordHts
DEFINE
  coordHts.env_other_trig := coordRedHts.env_other_trig
                           | coordBlueHts.env_other_trig;
  coordHts.entRedA_trig := coordRedHts.entRedA_trig
                           | coordBlueHts.entRedA_trig;
  coordHts.exitRedA_trig := coordRedHts.exitRedA_trig
                           | coordBlueHts.exitRedA_trig;
  coordHts.entRedB_trig := coordRedHts.entRedB_trig
                           | coordBlueHts.entRedB_trig;
  coordHts.exitRedB_trig := coordRedHts.exitRedB_trig
                           | coordBlueHts.exitRedB_trig;
  coordHts.entBlueA_trig := coordRedHts.entBlueA_trig
                           | coordBlueHts.entBlueA_trig;
  coordHts.exitBlueA_trig := coordRedHts.exitBlueA_trig
                           | coordBlueHts.exitBlueA_trig;
  coordHts.entBlueB_trig := coordRedHts.entBlueB_trig
                           | coordBlueHts.entBlueB_trig;
  coordHts.exitBlueB_trig := coordRedHts.exitBlueB_trig
```

```
                          | coordBlueHts.exitBlueB_trig;

  --define en.coordRedHts.any that use composition interleaving
    coordRedHts.any := redCoordEntHts.any | redCoordExitHts.any;
  --defining macro flag for environmental sync events in coordRedHts
  DEFINE
    coordRedHts.env_other_trig := redCoordEntHts.env_other_trig
                                | redCoordExitHts.env_other_trig;
    coordRedHts.entRedA_trig := redCoordEntHts.entRedA_trig
                                | redCoordExitHts.entRedA_trig;
    coordRedHts.exitRedA_trig := redCoordEntHts.exitRedA_trig
                                | redCoordExitHts.exitRedA_trig;
    coordRedHts.entRedB_trig := redCoordEntHts.entRedB_trig
                                | redCoordExitHts.entRedB_trig;
    coordRedHts.exitRedB_trig := redCoordEntHts.exitRedB_trig
                                | redCoordExitHts.exitRedB_trig;
    coordRedHts.entBlueA_trig := redCoordEntHts.entBlueA_trig
                                | redCoordExitHts.entBlueA_trig;
    coordRedHts.exitBlueA_trig := redCoordEntHts.exitBlueA_trig
                                | redCoordExitHts.exitBlueA_trig;
    coordRedHts.entBlueB_trig := redCoordEntHts.entBlueB_trig
                                | redCoordExitHts.entBlueB_trig;
    coordRedHts.exitBlueB_trig := redCoordEntHts.exitBlueB_trig
                                | redCoordExitHts.exitBlueB_trig;

  --define en.coordBlueHts.any that use composition interleaving
    coordBlueHts.any := blueCoordEntHts.any | blueCoordExitHts.any;
  --defining macro flag for environmental sync events in coordBlueHts
  DEFINE
    coordBlueHts.env_other_trig := blueCoordEntHts.env_other_trig
                                 | blueCoordExitHts.env_other_trig;
    coordBlueHts.entRedA_trig := blueCoordEntHts.entRedA_trig
                                 | blueCoordExitHts.entRedA_trig;
    coordBlueHts.exitRedA_trig := blueCoordEntHts.exitRedA_trig
                                 | blueCoordExitHts.exitRedA_trig;
    coordBlueHts.entRedB_trig := blueCoordEntHts.entRedB_trig
                                 | blueCoordExitHts.entRedB_trig;
    coordBlueHts.exitRedB_trig := blueCoordEntHts.exitRedB_trig
                                 | blueCoordExitHts.exitRedB_trig;
    coordBlueHts.entBlueA_trig := blueCoordEntHts.entBlueA_trig
                                 | blueCoordExitHts.entBlueA_trig;
    coordBlueHts.exitBlueA_trig := blueCoordEntHts.exitBlueA_trig
                                 | blueCoordExitHts.exitBlueA_trig;
    coordBlueHts.entBlueB_trig := blueCoordEntHts.entBlueB_trig
                                 | blueCoordExitHts.entBlueB_trig;
    coordBlueHts.exitBlueB_trig := blueCoordEntHts.exitBlueB_trig
                                 | blueCoordExitHts.exitBlueB_trig;
```

```
--------------------------------------------------------------------------

MODULE execute(en)
  VAR
    redAHts : execute_redAHts(en.redAHts);
    redBHts : execute_redBHts(en.redBHts);
    blueAHts : execute_blueAHts(en.blueAHts);
    blueBHts : execute_blueBHts(en.blueBHts);
    bridgeStatusHts : execute_bridgeStatusHts(en.bridgeStatusHts);
    redCoordEntHts : execute_redCoordEntHts(en.redCoordEntHts);
    redCoordExitHts : execute_redCoordExitHts(en.redCoordExitHts);
    blueCoordEntHts : execute_blueCoordEntHts(en.blueCoordEntHts);
    blueCoordExitHts : execute_blueCoordExitHts(en.blueCoordExitHts);
  VAR
    singleLaneBridgeHts :
      envsync_entRedA_exitRedA_entRedB_exitRedB_entBlueA_exitBlueA
        _entBlueB_exitBlueB(en.bridgeHts,en.coordHts,bridgeHts,coordHts);
  VAR
    bridgeHts : rendezvous_inRed_outRed_inBlue_outBlue
                (en.carHts,en.bridgeStatusHts,carHts,bridgeStatusHts);
  --defining macro flag for environmental sync events
  DEFINE
    bridgeHts.env_other_trig := carHts.env_other_trig | bridgeStatusHts.env_other_trig;
    bridgeHts.entRedA_trig := carHts.entRedA_trig | bridgeStatusHts.entRedA_trig;
    bridgeHts.exitRedA_trig := carHts.exitRedA_trig | bridgeStatusHts.exitRedA_trig;
    bridgeHts.entRedB_trig := carHts.entRedB_trig | bridgeStatusHts.entRedB_trig;
    bridgeHts.exitRedB_trig := carHts.exitRedB_trig | bridgeStatusHts.exitRedB_trig;
    bridgeHts.entBlueA_trig := carHts.entBlueA_trig | bridgeStatusHts.entBlueA_trig;
    bridgeHts.exitBlueA_trig := carHts.exitBlueA_trig | bridgeStatusHts.exitBlueA_trig;
    bridgeHts.entBlueB_trig := carHts.entBlueB_trig | bridgeStatusHts.entBlueB_trig;
    bridgeHts.exitBlueB_trig := carHts.exitBlueB_trig | bridgeStatusHts.exitBlueB_trig;
  VAR
    carHts : interleaving(en.redCarHts,en.blueCarHts,redCarHts,blueCarHts);
  --defining macro flag for environmental sync events
  DEFINE
    carHts.env_other_trig := redCarHts.env_other_trig | blueCarHts.env_other_trig;
    carHts.entRedA_trig := redCarHts.entRedA_trig | blueCarHts.entRedA_trig;
    carHts.exitRedA_trig := redCarHts.exitRedA_trig | blueCarHts.exitRedA_trig;
    carHts.entRedB_trig := redCarHts.entRedB_trig | blueCarHts.entRedB_trig;
    carHts.exitRedB_trig := redCarHts.exitRedB_trig | blueCarHts.exitRedB_trig;
    carHts.entBlueA_trig := redCarHts.entBlueA_trig | blueCarHts.entBlueA_trig;
    carHts.exitBlueA_trig := redCarHts.exitBlueA_trig | blueCarHts.exitBlueA_trig;
    carHts.entBlueB_trig := redCarHts.entBlueB_trig | blueCarHts.entBlueB_trig;
    carHts.exitBlueB_trig := redCarHts.exitBlueB_trig | blueCarHts.exitBlueB_trig;
  --define macro of more_than_one
  DEFINE
  carHts.more_than_one := redCarHts.more_than_one | blueCarHts.more_than_one
```

```
                      | (redCarHts.any & blueCarHts.any);
--defining macro flag for rendezvous events
DEFINE
  carHts.rend_other_trig := redCarHts.rend_other_trig | blueCarHts.rend_other_trig;
  carHts.inRed_trig := redCarHts.inRed_trig | blueCarHts.inRed_trig;
  carHts.inRed_gen := redCarHts.inRed_gen | blueCarHts.inRed_gen;
  carHts.outRed_trig := redCarHts.outRed_trig | blueCarHts.outRed_trig;
  carHts.outRed_gen := redCarHts.outRed_gen | blueCarHts.outRed_gen;
  carHts.inBlue_trig := redCarHts.inBlue_trig | blueCarHts.inBlue_trig;
  carHts.inBlue_gen := redCarHts.inBlue_gen | blueCarHts.inBlue_gen;
  carHts.outBlue_trig := redCarHts.outBlue_trig | blueCarHts.outBlue_trig;
  carHts.outBlue_gen := redCarHts.outBlue_gen | blueCarHts.outBlue_gen;
VAR
  redCarHts : interleaving(en.redAHts,en.redBHts,redAHts,redBHts);
--defining macro flag for environmental sync events
DEFINE
  redCarHts.env_other_trig := redAHts.env_other_trig | redBHts.env_other_trig;
  redCarHts.entRedA_trig := redAHts.entRedA_trig | redBHts.entRedA_trig;
  redCarHts.exitRedA_trig := redAHts.exitRedA_trig | redBHts.exitRedA_trig;
  redCarHts.entRedB_trig := redAHts.entRedB_trig | redBHts.entRedB_trig;
  redCarHts.exitRedB_trig := redAHts.exitRedB_trig | redBHts.exitRedB_trig;
  redCarHts.entBlueA_trig := redAHts.entBlueA_trig | redBHts.entBlueA_trig;
  redCarHts.exitBlueA_trig := redAHts.exitBlueA_trig | redBHts.exitBlueA_trig;
  redCarHts.entBlueB_trig := redAHts.entBlueB_trig | redBHts.entBlueB_trig;
  redCarHts.exitBlueB_trig := redAHts.exitBlueB_trig | redBHts.exitBlueB_trig;
--define macro of more_than_one
DEFINE
redCarHts.more_than_one := redAHts.more_than_one | redBHts.more_than_one
                         | (redAHts.any & redBHts.any);
--defining macro flag for rendezvous events
DEFINE
  redCarHts.rend_other_trig := redAHts.rend_other_trig | redBHts.rend_other_trig;
  redCarHts.inRed_trig := redAHts.inRed_trig | redBHts.inRed_trig;
  redCarHts.inRed_gen := redAHts.inRed_gen | redBHts.inRed_gen;
  redCarHts.outRed_trig := redAHts.outRed_trig | redBHts.outRed_trig;
  redCarHts.outRed_gen := redAHts.outRed_gen | redBHts.outRed_gen;
  redCarHts.inBlue_trig := redAHts.inBlue_trig | redBHts.inBlue_trig;
  redCarHts.inBlue_gen := redAHts.inBlue_gen | redBHts.inBlue_gen;
  redCarHts.outBlue_trig := redAHts.outBlue_trig | redBHts.outBlue_trig;
  redCarHts.outBlue_gen := redAHts.outBlue_gen | redBHts.outBlue_gen;
VAR
  blueCarHts : interleaving(en.blueAHts,en.blueBHts,blueAHts,blueBHts);
--defining macro flag for environmental sync events
DEFINE
  blueCarHts.env_other_trig := blueAHts.env_other_trig | blueBHts.env_other_trig;
  blueCarHts.entRedA_trig := blueAHts.entRedA_trig | blueBHts.entRedA_trig;
  blueCarHts.exitRedA_trig := blueAHts.exitRedA_trig | blueBHts.exitRedA_trig;
  blueCarHts.entRedB_trig := blueAHts.entRedB_trig | blueBHts.entRedB_trig;
```

```
  blueCarHts.exitRedB_trig := blueAHts.exitRedB_trig | blueBHts.exitRedB_trig;
  blueCarHts.entBlueA_trig := blueAHts.entBlueA_trig | blueBHts.entBlueA_trig;
  blueCarHts.exitBlueA_trig := blueAHts.exitBlueA_trig | blueBHts.exitBlueA_trig;
  blueCarHts.entBlueB_trig := blueAHts.entBlueB_trig | blueBHts.entBlueB_trig;
  blueCarHts.exitBlueB_trig := blueAHts.exitBlueB_trig | blueBHts.exitBlueB_trig;
--define macro of more_than_one
DEFINE
blueCarHts.more_than_one := blueAHts.more_than_one | blueBHts.more_than_one
                          | (blueAHts.any & blueBHts.any);
--defining macro flag for rendezvous events
DEFINE
  blueCarHts.rend_other_trig := blueAHts.rend_other_trig | blueBHts.rend_other_trig;
  blueCarHts.inRed_trig := blueAHts.inRed_trig | blueBHts.inRed_trig;
  blueCarHts.inRed_gen := blueAHts.inRed_gen | blueBHts.inRed_gen;
  blueCarHts.outRed_trig := blueAHts.outRed_trig | blueBHts.outRed_trig;
  blueCarHts.outRed_gen := blueAHts.outRed_gen | blueBHts.outRed_gen;
  blueCarHts.inBlue_trig := blueAHts.inBlue_trig | blueBHts.inBlue_trig;
  blueCarHts.inBlue_gen := blueAHts.inBlue_gen | blueBHts.inBlue_gen;
  blueCarHts.outBlue_trig := blueAHts.outBlue_trig | blueBHts.outBlue_trig;
  blueCarHts.outBlue_gen := blueAHts.outBlue_gen | blueBHts.outBlue_gen;
VAR
  coordHts : interleaving(en.coordRedHts,en.coordBlueHts,coordRedHts,coordBlueHts);
--defining macro flag for environmental sync events
DEFINE
  coordHts.env_other_trig := coordRedHts.env_other_trig | coordBlueHts.env_other_trig;
  coordHts.entRedA_trig := coordRedHts.entRedA_trig | coordBlueHts.entRedA_trig;
  coordHts.exitRedA_trig := coordRedHts.exitRedA_trig | coordBlueHts.exitRedA_trig;
  coordHts.entRedB_trig := coordRedHts.entRedB_trig | coordBlueHts.entRedB_trig;
  coordHts.exitRedB_trig := coordRedHts.exitRedB_trig | coordBlueHts.exitRedB_trig;
  coordHts.entBlueA_trig := coordRedHts.entBlueA_trig | coordBlueHts.entBlueA_trig;
  coordHts.exitBlueA_trig := coordRedHts.exitBlueA_trig | coordBlueHts.exitBlueA_trig;
  coordHts.entBlueB_trig := coordRedHts.entBlueB_trig | coordBlueHts.entBlueB_trig;
  coordHts.exitBlueB_trig := coordRedHts.exitBlueB_trig | coordBlueHts.exitBlueB_trig;
VAR
  coordRedHts : interleaving(en.redCoordEntHts,en.redCoordExitHts,
                             redCoordEntHts,redCoordExitHts);
--defining macro flag for environmental sync events
DEFINE
  coordRedHts.env_other_trig :=  redCoordEntHts.env_other_trig
                               | redCoordExitHts.env_other_trig;
  coordRedHts.entRedA_trig :=    redCoordEntHts.entRedA_trig
                               | redCoordExitHts.entRedA_trig;
  coordRedHts.exitRedA_trig :=   redCoordEntHts.exitRedA_trig
                               | redCoordExitHts.exitRedA_trig;
  coordRedHts.entRedB_trig :=    redCoordEntHts.entRedB_trig
                               | redCoordExitHts.entRedB_trig;
  coordRedHts.exitRedB_trig :=   redCoordEntHts.exitRedB_trig
                               | redCoordExitHts.exitRedB_trig;
```

```
    coordRedHts.entBlueA_trig :=   redCoordEntHts.entBlueA_trig
                                 | redCoordExitHts.entBlueA_trig;
    coordRedHts.exitBlueA_trig :=  redCoordEntHts.exitBlueA_trig
                                 | redCoordExitHts.exitBlueA_trig;
    coordRedHts.entBlueB_trig :=   redCoordEntHts.entBlueB_trig
                                 | redCoordExitHts.entBlueB_trig;
    coordRedHts.exitBlueB_trig :=  redCoordEntHts.exitBlueB_trig
                                 | redCoordExitHts.exitBlueB_trig;
  VAR
    coordBlueHts : interleaving(en.blueCoordEntHts,en.blueCoordExitHts,
                                blueCoordEntHts,blueCoordExitHts);
  --defining macro flag for environmental sync events
  DEFINE
    coordBlueHts.env_other_trig :=  blueCoordEntHts.env_other_trig
                                  | blueCoordExitHts.env_other_trig;
    coordBlueHts.entRedA_trig :=    blueCoordEntHts.entRedA_trig
                                  | blueCoordExitHts.entRedA_trig;
    coordBlueHts.exitRedA_trig :=   blueCoordEntHts.exitRedA_trig
                                  | blueCoordExitHts.exitRedA_trig;
    coordBlueHts.entRedB_trig :=    blueCoordEntHts.entRedB_trig
                                  | blueCoordExitHts.entRedB_trig;
    coordBlueHts.exitRedB_trig :=   blueCoordEntHts.exitRedB_trig
                                  | blueCoordExitHts.exitRedB_trig;
    coordBlueHts.entBlueA_trig :=   blueCoordEntHts.entBlueA_trig
                                  | blueCoordExitHts.entBlueA_trig;
    coordBlueHts.exitBlueA_trig :=  blueCoordEntHts.exitBlueA_trig
                                  | blueCoordExitHts.exitBlueA_trig;
    coordBlueHts.entBlueB_trig :=   blueCoordEntHts.entBlueB_trig
                                  | blueCoordExitHts.entBlueB_trig;
    coordBlueHts.exitBlueB_trig :=  blueCoordEntHts.exitBlueB_trig
                                  | blueCoordExitHts.exitBlueB_trig;

  INVAR
    (en.singleLaneBridgeHts.any -> singleLaneBridgeHts.any)



--------------------------------------------------------------------------

MODULE apply(pss,iss,exe)
  VAR
    nextCS : nextCS(pss,iss,exe);
    nextAV : nextAV(pss,iss,exe);
    nextO : nextO(pss,iss,exe);
    nextIa : nextIa(pss,iss,exe);

MODULE nextCS(pss,iss,exe)
--nextCS: destT
```

```
ASSIGN
  next(pss.CS.redAHts_state):=case
    exe.redAHts.t1 : moveOnRedA;
    exe.redAHts.t2 : moveOutRedA;
    exe.redAHts.t25 : onRedA;
    exe.redAHts.t26 : waitRedA;
    1 : iss.CS.redAHts_state;
  esac;

ASSIGN
  next(pss.CS.redBHts_state):=case
    exe.redBHts.t3 : moveOnRedB;
    exe.redBHts.t4 : moveOutRedB;
    exe.redBHts.t27 : onRedB;
    exe.redBHts.t28 : waitRedB;
    1 : iss.CS.redBHts_state;
  esac;

ASSIGN
  next(pss.CS.blueAHts_state):=case
    exe.blueAHts.t9 : moveOnBlueA;
    exe.blueAHts.t10 : moveOutBlueA;
    exe.blueAHts.t29 : onBlueA;
    exe.blueAHts.t30 : waitBlueA;
    1 : iss.CS.blueAHts_state;
  esac;

ASSIGN
  next(pss.CS.blueBHts_state):=case
    exe.blueBHts.t11 : moveOnBlueB;
    exe.blueBHts.t12 : moveOutBlueB;
    exe.blueBHts.t31 : onBlueB;
    exe.blueBHts.t32 : onBlueB;
    1 : iss.CS.blueBHts_state;
  esac;

ASSIGN
  next(pss.CS.bridgeStatusHts_state):=case
    exe.bridgeStatusHts.t17 : oneRed;
    exe.bridgeStatusHts.t18 : twoRed;
    exe.bridgeStatusHts.t19 : oneRed;
    exe.bridgeStatusHts.t20 : empty;
    exe.bridgeStatusHts.t21 : oneBlue;
    exe.bridgeStatusHts.t22 : twoBlue;
    exe.bridgeStatusHts.t23 : oneBlue;
    exe.bridgeStatusHts.t24 : empty;
    1 : iss.CS.bridgeStatusHts_state;
  esac;
```

```
  ASSIGN
    next(pss.CS.redCoordEntHts_state):=case
      exe.redCoordEntHts.t5 : coordEntRedB;
      exe.redCoordEntHts.t6 : coordEntRedA;
      1 : iss.CS.redCoordEntHts_state;
    esac;

  ASSIGN
    next(pss.CS.redCoordExitHts_state):=case
      exe.redCoordExitHts.t7 : coordExitRedB;
      exe.redCoordExitHts.t8 : coordExitRedA;
      1 : iss.CS.redCoordExitHts_state;
    esac;

  ASSIGN
    next(pss.CS.blueCoordEntHts_state):=case
      exe.blueCoordEntHts.t13 : coordEntBlueB;
      exe.blueCoordEntHts.t14 : coordEntBlueA;
      1 : iss.CS.blueCoordEntHts_state;
    esac;

  ASSIGN
    next(pss.CS.blueCoordExitHts_state):=case
      exe.blueCoordExitHts.t15 : coordExitBlueB;
      exe.blueCoordExitHts.t16 : coordExitBlueA;
      1 : iss.CS.blueCoordExitHts_state;
    esac;


MODULE nextAV(pss,iss,exe)
--nextAV : evalAsn
  --next state relation for  numRed
  DEFINE numRedt1 := case
  ((((iss.AV.numRed) + (1)))>=0)&((((iss.AV.numRed) + (1)))<=2)
                : ((iss.AV.numRed) + (1));
  1 : iss.AV.numRed;
 esac;
  DEFINE numRedt1error := exe.redAHts.t1 & ((((iss.AV.numRed) + (1)))<0)
                        |((((iss.AV.numRed) + (1)))>2);
  DEFINE numRedt2 := case
  ((((iss.AV.numRed) - (1)))>=0)&((((iss.AV.numRed) - (1)))<=2)
                : ((iss.AV.numRed) - (1));
  1 : iss.AV.numRed;
 esac;
  DEFINE numRedt2error := exe.redAHts.t2 & ((((iss.AV.numRed) - (1)))<0)
                        |((((iss.AV.numRed) - (1)))>2);
  DEFINE numRedt3 := case
```

```
 ((((iss.AV.numRed) + (1)))>=0)&((((iss.AV.numRed) + (1)))<=2)
               : ((iss.AV.numRed) + (1));
 1 : iss.AV.numRed;
esac;
 DEFINE numRedt3error := exe.redBHts.t3 & ((((iss.AV.numRed) + (1)))<0)
                        |((((iss.AV.numRed) + (1)))>2);
 DEFINE numRedt4 := case
 ((((iss.AV.numRed) - (1)))>=0)&((((iss.AV.numRed) - (1)))<=2)
               : ((iss.AV.numRed) - (1));
 1 : iss.AV.numRed;
esac;
 DEFINE numRedt4error := exe.redBHts.t4 & ((((iss.AV.numRed) - (1)))<0)
                        |((((iss.AV.numRed) - (1)))>2);
 ASSIGN
   next(pss.AV.numRed):=case
     exe.redAHts.t1 : numRedt1;
     exe.redAHts.t2 : numRedt2;
     exe.redBHts.t3 : numRedt3;
     exe.redBHts.t4 : numRedt4;
     1 : iss.AV.numRed;
   esac;

 --next state relation for  numBlue
 DEFINE numBluet9 := case
 ((((iss.AV.numBlue) + (1)))>=0)&((((iss.AV.numBlue) + (1)))<=2)
               : ((iss.AV.numBlue) + (1));
 1 : iss.AV.numBlue;
esac;
 DEFINE numBluet9error := exe.blueAHts.t9 & ((((iss.AV.numBlue) + (1)))<0)
                         |((((iss.AV.numBlue) + (1)))>2);
 DEFINE numBluet10 := case
 ((((iss.AV.numBlue) - (1)))>=0)&((((iss.AV.numBlue) - (1)))<=2)
               : ((iss.AV.numBlue) - (1));
 1 : iss.AV.numBlue;
esac;
 DEFINE numBluet10error := exe.blueAHts.t10 & ((((iss.AV.numBlue) - (1)))<0)
                          |((((iss.AV.numBlue) - (1)))>2);
 DEFINE numBluet11 := case
 ((((iss.AV.numBlue) + (1)))>=0)&((((iss.AV.numBlue) + (1)))<=2)
               : ((iss.AV.numBlue) + (1));
 1 : iss.AV.numBlue;
esac;
 DEFINE numBluet11error := exe.blueBHts.t11 & ((((iss.AV.numBlue) + (1)))<0)
                          |((((iss.AV.numBlue) + (1)))>2);
 DEFINE numBluet12 := case
 ((((iss.AV.numBlue) - (1)))>=0)&((((iss.AV.numBlue) - (1)))<=2)
               : ((iss.AV.numBlue) - (1));
 1 : iss.AV.numBlue;
```

```
 esac;
  DEFINE numBluet12error := exe.blueBHts.t12 & ((((iss.AV.numBlue) - (1)))<0)
                          |((((iss.AV.numBlue) - (1)))>2);
  ASSIGN
    next(pss.AV.numBlue):=case
      exe.blueAHts.t9 : numBluet9;
      exe.blueAHts.t10 : numBluet10;
      exe.blueBHts.t11 : numBluet11;
      exe.blueBHts.t12 : numBluet12;
      1 : iss.AV.numBlue;
    esac;


  --next state relation for variable overflow and underflow
  ASSIGN
    next(pss.AV.error_variables):=
       iss.AV.error_variables
      |numRedt1error
      |numRedt2error
      |numRedt3error
      |numRedt4error
      |numBluet9error
      |numBluet10error
      |numBluet11error
      |numBluet12error
       ;

MODULE nextO(pss,iss,exe)
--nextO : OGen
  --next state relation for output inRed
  ASSIGN
    next(pss.OO.inRed):=0;

  --next state relation for output outRed
  ASSIGN
    next(pss.OO.outRed):=0;

  --next state relation for output inBlue
  ASSIGN
    next(pss.OO.inBlue):=0;

  --next state relation for output outBlue
  ASSIGN
    next(pss.OO.outBlue):=0;
```

```
MODULE nextIa(pss,iss,exe)
--nextIa : ssIaUGen
  --next state relation for entRedA
  ASSIGN
    next(pss.Ia.entRedA) := case
      (iss.Ia.entRedA=1) : 1;
      1 : 0;
    esac;


  --next state relation for exitRedA
  ASSIGN
    next(pss.Ia.exitRedA) := case
      (iss.Ia.exitRedA=1) : 1;
      1 : 0;
    esac;


  --next state relation for entRedB
  ASSIGN
    next(pss.Ia.entRedB) := case
      (iss.Ia.entRedB=1) : 1;
      1 : 0;
    esac;


  --next state relation for exitRedB
  ASSIGN
    next(pss.Ia.exitRedB) := case
      (iss.Ia.exitRedB=1) : 1;
      1 : 0;
    esac;


  --next state relation for entBlueA
  ASSIGN
    next(pss.Ia.entBlueA) := case
      (iss.Ia.entBlueA=1) : 1;
      1 : 0;
    esac;


  --next state relation for exitBlueA
  ASSIGN
    next(pss.Ia.exitBlueA) := case
      (iss.Ia.exitBlueA=1) : 1;
      1 : 0;
    esac;
```

```
  --next state relation for entBlueB
  ASSIGN
    next(pss.Ia.entBlueB) := case
      (iss.Ia.entBlueB=1) : 1;
      1 : 0;
    esac;


  --next state relation for exitBlueB
  ASSIGN
    next(pss.Ia.exitBlueB) := case
      (iss.Ia.exitBlueB=1) : 1;
      1 : 0;
    esac;




--------------------------------------------------------------------------

MODULE initss(pss)
  ASSIGN
    init(pss.CS.redAHts_state):=waitRedA;
    init(pss.CS.redBHts_state):=waitRedB;
    init(pss.CS.blueAHts_state):=waitBlueA;
    init(pss.CS.blueBHts_state):=waitBlueB;
    init(pss.CS.bridgeStatusHts_state):=empty;
    init(pss.CS.redCoordEntHts_state):=coordEntRedA;
    init(pss.CS.redCoordExitHts_state):=coordExitRedA;
    init(pss.CS.blueCoordEntHts_state):=coordEntBlueA;
    init(pss.CS.blueCoordExitHts_state):=coordExitBlueA;
    init(pss.AV.numRed):={0};
    init(pss.AV.numBlue):={0};
    init(pss.AV.error_variables):=0;
    init(pss.Ia.entRedA):=0;
    init(pss.Ia.exitRedA):=0;
    init(pss.Ia.entRedB):=0;
    init(pss.Ia.exitRedB):=0;
    init(pss.Ia.entBlueA):=0;
    init(pss.Ia.exitBlueA):=0;
    init(pss.Ia.entBlueB):=0;
    init(pss.Ia.exitBlueB):=0;
MODULE main
  VAR
    --pss is a set of variables storing snapshot elements
```

```
pss : snapshot;
--I is a set of inputs
I : inputs;
--iss is a set of macros of type snapshot
iss : reset(pss,I);
--iss_en is a set of macros identifying enabled entities in iss
iss_en: enabled(iss);
--iss_exe is a set of macros identifying executing entities
iss_exe: execute(iss_en);
--initss is a module containing initlialization statements
_initss: initss(pss);
--apply is a module containing next statements
_apply : apply(pss,iss,iss_exe);
```

Bibliography

[1] Cadence SMV, `http://www-cad.eecs.berkeley.edu/~kenmcmil/smv/`, 2004.

[2] J. M. Atlee and J. Gannon. State-based model checking of event-driven system requirements. *IEEE Transactions on Software Engineering*, 19(1):24–40, 1993.

[3] S. Bensalem et al. An overview of SAL. In *NASA Langley Formal Methods Workshop*, pages 187–196. Center for Aerospace Information, NASA, 2000.

[4] R. Bharadwaj and C. L. Heitmeyer. Model checking complete requirements specifications using abstraction. *Automated Software Engineering*, 6(1):37–68, 1999.

[5] G. Booch. *Object Oriented Design*. Benjamin Cummings, 1991.

[6] M. Bozga, S. Graf, L. Mounier, and J. Sifakis. The intermediate representation IF: syntax and semantics. Technical report, Verimag, Grenoble, 1999.

[7] T. Bultan. Action language: A specification language for model checking reactive systems. In *International Conference on Software Engineering*, pages 335–344, 2000.

[8] W. Chan, R. J. Anderson, P. Beame, S. Burns, F. Modugno, and D. Notkin. Model checking large software specifications. *IEEE Transactions on Software Engineering*, 24(7):498–519, 1998.

[9] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.

[10] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[11] J. C. Corbett, M. B. Dwyer, J. Hatcliff, and et al. Bandera: Extracting finite-state models from java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.

[12] N. A. Day. Fusion User's Guide. Unpublished.

[13] N. A. Day. *A Framework for Multi-Notation, Model-Oriented Requirements Analysis*. PhD thesis, October 1998.

[14] N. A. Day and J. J. Joyce. Symbolic functional evaluation. In *Theorem Proving in Higher Order Logics*, volume 1690 of *Lecture Notes In Computer Science*, pages 341–358. Springer, 1999.

[15] L. Dillon and R. Stirewalt. Inference graphs: a computational structure supporting generation of customizable and correct analysis components. *IEEE Transactions on Software Engineering*, 29(2):133–150, Feb 2003.

[16] M. Gordon and T. Melham. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[17] D. Harel et al. On the formal semantics of statecharts. In *Logic in Computer Science*, pages 54–64, 1987.

[18] D. Harel and A. Naamad. The statemate semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.

[19] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, UK, 1985.

[20] G. Holzmann. The model checker Spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[21] ISO8807. LOTOS - a formal description technique based on the temporal ordering of observational behaviour. Technical report, ISO, 1988.

[22] ITU-T. Recommendation Z.100. Specification and Description Language (SDL). Technical Report Z-100, International Telecommunication Union - Standardization Sector, 1999.

[23] S. Katz and O. Grumberg. A framework for translating models and specifications. In *Integrated Formal Methods: Third Intl. Conf., volume 2335 of LNCS*, pages 145–164. Springer-Verlag, 2002.

[24] N. G. Leveson, M. P. E. Heimdahl, H. Hildreth, and J. D. Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, 20(9):684–707, 1994.

[25] Y. Lu, J. M. Atlee, N. A. Day, and J. Niu. Mapping template semantics to SMV. In *IEEE International Conference on Automated Software Engineering*, September 2004.

[26] J. Magee and J. Kramer. *Concurrency, State Models & Java Programs*. John Wiley & Sons, UK, 1999.

[27] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1991.

[28] K. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic, 1993.

[29] R. Milner. *Communication and Concurrency*. Prentice Hall, New York, 1989.

[30] J. Niu. *Metro: a semantics-based approach for mapping specification notations to analysis tools*. PhD thesis, October 2004. In preparation.

[31] J. Niu, J. M. Atlee, and N. A. Day. Template semantics for model-based notations. *IEEE Transactions on Software Engineering*, 20(10):866–882, Oct. 2003.

[32] J. Niu, J. M. Atlee, and N. A. Day. Understanding and comparing model-based specification notations. In *IEEE International Requirements Engineering Conference*, pages 188–199. IEEE Computer Society Press, Washington D.C., 2003.

[33] M. Pezzè and M. Young. Creating of multi-formalism state-space analysis tools. In *International Symposium on Software Testing and Analysis*, pages 172–179. ACM Press, 1996.

[34] Robby, M. B. Dwyer, and J. Hatcliff. Bogor: An extensible and highly-modular software model checking framework. In *European Software Engineering Conference and Foundations of Software Engineering*, pages 267–276. ACM Press, 2003.