Model Checking Template-Semantics Specifications

Yun Lu Joanne M. Atlee Nancy A. Day Jianwei Niu
School of Computer Science
University of Waterloo
Waterloo, Ontario, N2L 3G1
Email: {y4lu, jmatlee, nday, jniu}@uwaterloo.ca

Technical Report : CS-2004-20
April 05, 2004

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. We introduce a translator that takes as input a specification, and a set of template parameters, encoding the specification's semantics, and produces a model suitable for the NuSMV model checker. 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 NuSMV language, in which there is no matching language construct.

I. Introduction

Template semantics [1] is a way of describing the semantics of model-based notations, which are formal notations with step semantics, such as statecharts variants and process algebras. The semantics that are common among notations (e.g., the concept of an enabled transition) are pre-defined as a template of parameterized definitions, and users instantiate the template into a complete semantics by providing notation-specific parameter values (e.g., predicates that describe how states, events, and variables enable transitions). Composition operators are parameterized constraints on how components execute and share information. In previous work, we showed how template semantics can describe succinctly the meanings of many popular requirements notations [2].

Our main goal in creating template semantics was to ease the construction of notation-specific analysis tools, for new, customized, or evolving requirements notations. In this report, we investigate using template semantics to parameterize the translation from a requirements notation to the input language of an existing analysis tool, the NuSMV model checker [3]. We describe a fully automated translator that takes as input a specification written in template-semantics syntax and a set of template parameters detailing the specification's

semantics; the translator combines these inputs with template-semantics' parameterized definitions, to generate a NuSMV model of the specification. The translator supports all of the template-parameter values and the composition operators that were used in [1] to describe the semantics of basic transition systems (BTS) [4], CSP [5], CCS [6], basic LOTOS [7], and several statecharts [8] variants. Because the supported parameter values describe a variety of ways in which states, events, variables, and priorities affect a notation's semantics, the translator can be used for many more notations (defined by different combinations of parameter values and composition operators) than the notations listed above.

In translating template semantics to NuSMV, our goal was to handle a large collection of notations, without sacrificing analysis efficiency. First, we wanted the NuSMV code to 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. We also wanted the NuSMV code to match the composition structure of the specification to make it easier to check the correctness of our translation. Second, we wanted to avoid introducing intermediate execution steps or variables, so that the counterexamples output by NuSMV would be in terms of the original specification. Third, once the first two goals were met, we endeavoured to keep the specification's state space as small as possible. The report discusses how well we have succeeded in meeting these three goals.

Efforts to facilitate notation-specific analysis have involved writing custom analysis tools for particular notations (e.g., Concurrency Workbench [9], STATEMATE [10]); constructing direct translators from requirements notations to analysis tools [11], [12], [13], [14]; and using intermediate languages (e.g., SAL [15], IF [16]) to reduce the number of translations between requirements notations and analysis tools. Template semantics is like a parameterized intermediate language; because our translator supports multiple options for each of the template parameters, we can effectively generate new notation-specific translators by simply selecting different combinations of template-parameter values and composition operators. Furthermore, the modularity of our translation 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. Template semantics supports a richer set of composition operators than other intermediate languages, so our translations better preserve the structure, modularity, and composition operators of the original specification.

The main contribution of this work is a parameterized technique for model checking specifications written in a variety of notations. This work validates our claim that using template semantics considerably reduces the effort involved in constructing notation-specific analyzers. A second important contribution of this work is in describing how to model a rich set of composition operators – including rendezvous, environmental synchronization, sequence, choice, interrupt (a form of goto) – within the fairly simple language features of NuSMV. Because of the simplicity of the NuSMV language, this result should be generalizable to other model checkers.

We evaluated our work with two case studies, chosen to exercise a broad range of composition operators. Using NuSMV simulation and model checking, we checked that our translator produces a specification whose behaviour matches the expected behaviour. We also introspectively evaluated how well our translation algorithm meets the goals for translation that were outlined above.

In Section II, we present two case studies that we will use to both demonstrate and evaluate our translation. In Section III, we provide an overview of template semantics. In Section IV, we briefly review the NuSMV

language features that we use in our translation, which is described in Sections V. We evaluate our translation with respect to the two case studies in Section VI. We discuss related work in Section VI, and conclude in Section VIII. We give the SMV models produced by our translator for the two case studies in the Appendices.

II. Two Examples

We use specifications of a single lane bridge [17] and a heating system [18] as examples throughout the report. The bridge specification (Page 5) 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 7 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 are of type integer range from 0 to 2, and are initialized to 0. The system has 8 environment events. For example, event *entRedA* means a red car enters the bridge, and event *entRedA* means a red car exits the bridge. The system has 4 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.

The heating system (Page 6) 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 desired, 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 12 declares variables needed for the specification of the heating system. Variable *valvePos* is the valve position of the room, which has type integer range from 0 to 2, with 0 representing the valve is closed, 1 representing the valve is half open, and 2 representing the valve is open. Constants *furnaceTimer*, *warmUpTimer*, and *coolDownTimer* represent the time needed to start up the furnace, to warm up the room, and to cool down the room, respectively. Variables *furnaceStartupTime*, *warmUpTime*, and *coolDownTime* are counters to track the time needed to start up the furnace, to warm up the room, and to cool down the room, respectively. Variable *requestHeat* is a Boolean variable to indicate whether the room needs heat. The actual temperature (variable *actualTemp*) is sensed by the sensor from the environment, and the user can set the desired temperature using variable *setTemp*. Condition *tooCold* indicates the room is too cold when the difference between the desired temperature (*setTemp*) and the actual temperature (*actualTemp*) is greater than 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 constant 2.

III. Template Semantics

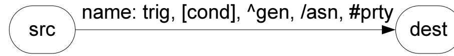
In this section, we briefly describe **template semantics**, a template-based approach to structuring the semantics of model-based notations. The basic computation model is a non-concurrent, hierarchical transition system (HTS). An HTS is an extended state machine, adapted from basic transition systems [4] and statecharts' hierarchy [8]. A specification is a hierarchical composition of HTSs; concurrency is introduced via composition.

A. 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 specify the sets of initial states and final states, respectively; and 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, each of which 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.

To model check a template-semantics specification, we first represent the specification as a collection of HTSs. In the single lane bridge specification, each car is modeled as an HTS that alternates between the car waiting to move onto the bridge and the car being on the bridge There are two cars of each colour (Figure 1 and Figure 2). 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 order (Figure 3 and Figure 4). The bridge is modeled by an HTS with five states that indicate the number and colours of cars on the bridge (Figure 5).

In the heating system, the *furnace* HTS (Figure 8) has hierarchical states: **super-states** (e.g., *furnaceNormal*) contain child states, and **basic states** (e.g., *furnaceRun*) contain no other states. A super-state has a default child state that is entered when the super-state is a transition's destination (e.g, *furnaceNormal* and *furnaceOff* are entered by transition *t6*). The room (Figure 11) is modelled as two HTSs: *noHeatReq* models room and valve behaviour when the furnace is off, and the *heatReq* HTS models behaviour when the furnace is on. The *controller* HTS activates and deactivates the furnace on the request from the room (Figure 9).

B. Semantics of HTS

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

$$\langle CS, IE, AV, O, CS_a, IE_a, AV_a, I_a \rangle$$

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. CS_a, AV_a, IE_a, I_a are auxiliary elements that

Fig. 1. Red Car HTSs

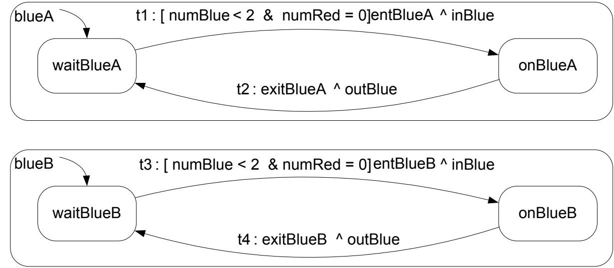


Fig. 2. Blue Car HTSs

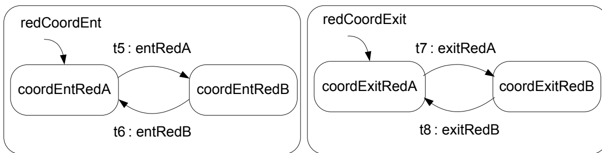


Fig. 3. Red Car Coordinator HTSs

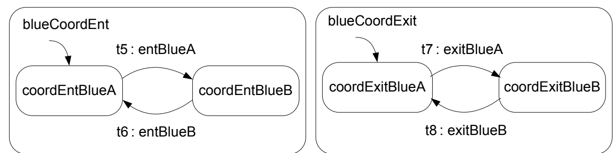


Fig. 4. Blue Car Coordinator HTSs

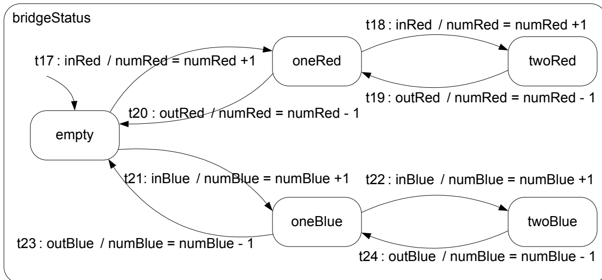


Fig. 5. Bridge HTS

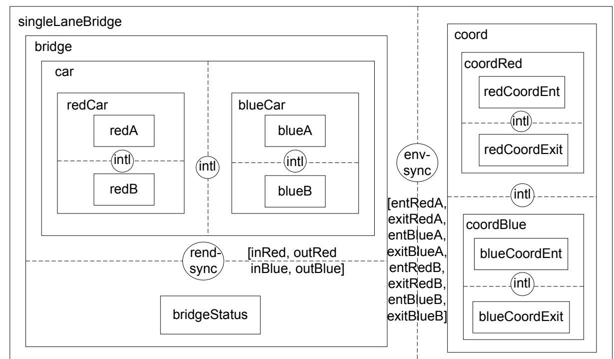


Fig. 6. Single Lane Bridge

```
/*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;
```

Fig. 7. Variable Declarations

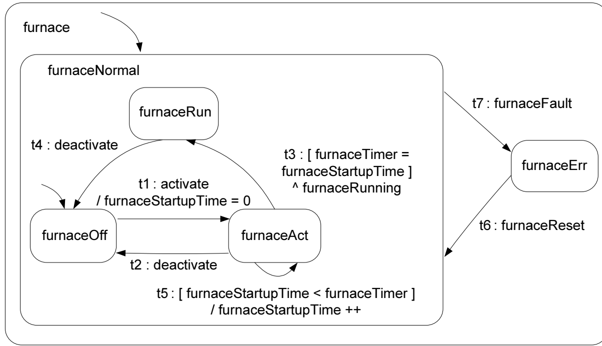


Fig. 8. Furnace HTS

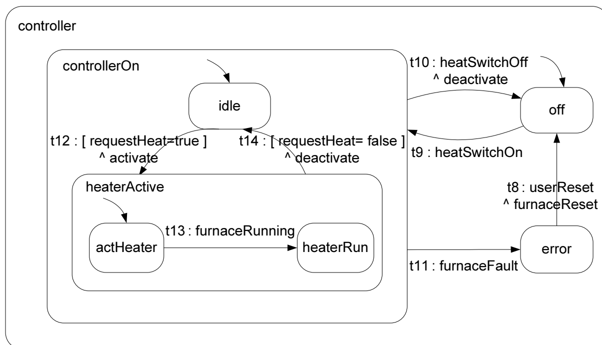


Fig. 9. Controller HTS

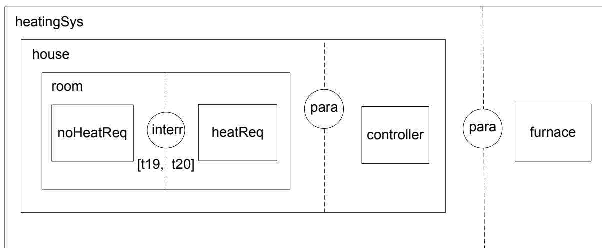


Fig. 10. Heating System

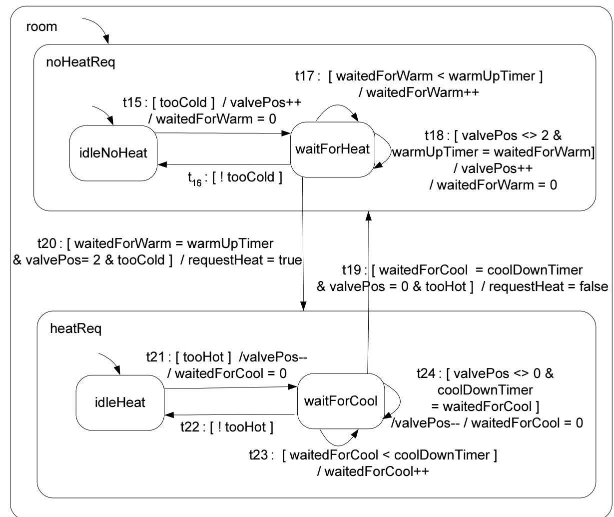


Fig. 11. Room Composed HTS

```
/*variables*/
var valvePos: range(0,2) = 0;
var furnaceStartupTime : range(0,5) = 0;
var warmUpTimer : range(0,5) = 0;
var coolDownTime : range(0,5) = 0;
var requestHeat : bool = false;

/*environmental variables*/
evar setTemp : range(12,24);
evar actualTemp : range(10,30);

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

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

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

Fig. 12. Variable Declarations

accumulate data about states, variables values, and internal and external events, respectively. Notations use these auxiliary elements for different purposes.

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

- ***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
- ***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

These definitions are instantiated by 22 template parameters that use, reset, and update the snapshot elements

Parameter		STATEMATE		CCS with variables	
		$resetXX(ss, I)$	$nextXX(ss, \tau)$	$resetXX(ss, I)$	$nextXX(ss, \tau)$
states	$CS =$	$ss.CS$	$entered(dest(\tau))$	$ss.CS$	$dest(\tau)$
	en_states	$src(\tau) \subseteq ss.CS$		$src(\tau) \subseteq ss.CS$	
events	$IE =$	$\emptyset$	$gen(\tau)$	n/a	
	$I_a =$	$I.ev$	$\emptyset$	$I.ev$	$ss.I_a \cup gen(\tau)$
	en_events	$trig(\tau) \subseteq ss.I_a \cup ss.IE$		$trig(\tau) \subseteq ss.I_a$	
	$O =$	$\emptyset$	$gen(\tau)$	$\emptyset$	$gen(\tau)$
variables	$AV =$	$assign(ss.AV, I.var)$	$assign(ss.AV, eval(ss.AV, last(asn(\tau))))$	$assign(ss.AV, I.var)$	$assign(ss.AV, eval(ss.AV, asn(\tau)))$
	en_cond	$ss.AV \models cond(\tau)$		$ss.AV \models cond(\tau)$	
macro-semantics		stable		simple diligent	
pri		lowest-ranked scope		none	
resolve_conflicts		$resolve(vv_1, vv_2, vv)$		n/a	

$entered(s)$: set of states when state s is entered, including s's ancestors and relevant descendants' default states
$assign(X, Y)$: 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
$resolve(vv_1, vv_2, vv)$: the conflicting variable assignments in vv_1 and in vv_2 are resolved to vv non-deterministically
$gen(\tau)$, $dest(\tau)$, $src(\tau)$, $trig(\tau)$, $cond(\tau)$, and $asn(\tau)$ are accessor functions on transition τ

TABLE I. Template Parameter Definitions for STATEMATE and CCS with variables

in different ways for different notations.

Table I shows template instantiations for two notations, STATEMATE statecharts [19] and CCS [6] with variables.[1] Column ***resetXX(ss,I)*** lists the parameters used in template definition ***reset***: each parameter resets a snapshot element in snapshot ss, removing old data and incorporating new system inputs I. Column ***nextXX(ss,τ)*** lists the parameters used in ***apply***: each parameter updates a snapshot element with respect to transition τ's actions. For example, five event-related parameters work together to define STATEMATE event semantics: when the current snapshot is ***reset***, snapshot element IE is emptied of any old internal events, and element I_a is set to the input's events, $I.ev$; whenever a transition executes, IE is set to the transition's set of generated events, and I_a is emptied; transitions are enabled by any event in IE or I_a (i.e., by an external event in I_a if the snapshot has just been reset, otherwise by an internal event generated by the most recently executed transition).

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**,1 in which enabled transitions have priority over an idle step, or **non-diligent**. 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.

Parameter ***pri*** determines the priority scheme among enabled transitions. In STATEMATE, the highest-priority transitions are those whose scope (the nearest common ancestor of a transition's source and destination states) has the lowest rank (i.e., is closest to the top of the state hierarchy).

Parameter ***resolve_conflicts*** sets the notation's policy for resolving conflicting assignments made to the same variable by multiple transitions in the same micro-step.

Our single lane bridge specification uses CCS semantics with variables, and our heating system uses

[1]Table I's *CCS with variables* is CCS with shared global variables; this is different from data-passing CCS [6], which allows internal events to carry data parameters.

STATEMATE semantics.

C. Composition Operators

We use composition operators 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 result is a named composite HTS that can be further composed. We have defined the template semantics for eight composition operators: two kinds of parallel, interleaving, environmental synchronization, rendezvous, sequence, choice, and interrupt. Interrupt composition transfers control between its operand HTSs, via new composition-level transitions; interrupt transitions' source and destination states can be either states of a component or a component itself.

Figure 6 shows the composition of the single lane bridge HTSs: 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. 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 *carBridge* on environmental events such as $entRed_A$ and $entRed_B$.

Figure 10 shows the composition of the heating system HTSs. HTSs *heatReq* and *noHeatReq* are composed using interrupt composition[2], which has associated with it the interrupt transitions *t19* and *t20* to transfer control between the two HTSs (Figure 11). The room, the controller, and the furnace execute concurrently via parallel composition.

In both specifications, events are broadcast, and data variables are global and shared among the HTSs.

IV. NuSMV

NuSMV is a symbolic model checker descended from SMV [20], which has a fairly simple input language. Variables are declared and assigned in the `VAR` and `ASSIGN` sections, respectively. NuSMV provides built-in finite data types, such as boolean, enumerated type, and integer range. Variables are assigned new values in every NuSMV 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)`). Expression operators `!`, `&`, `|`, `->`, and `<->`, represent "not," "and," "or," "implies," and "iff" respectively. An assignment can be a non-deterministic choice by putting the multiple expressions in curly brackets({}). Comments follow the symbol "`--`".

NuSMV supports macro definitions, declared in the `DEFINE` section. Macros are replaced by their definitions, so they do not increase the system's state space. In the following sections, we sometimes refer to boolean macros as "flags". NuSMV also supports the specification of invariants, as boolean expressions following keyword `INVAR`.

A `MODULE` bundles statements together, so that the statements can be reused by creating an instance of the module; module instances are declared as variables in a model's `VAR` section. A module can also be used to structure variables into a record that can be passed as a parameter to another module. If `a` identifies an instance

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

of a module, then the expression `a.b` identifies the internal variable or macro named `b` within module instance `a`. All statements in all modules run synchronously in every NuSMV step.

V. Translation to NuSMV

In this section, we show how to translate a composed hierarchical transition system into a collection of NuSMV modules, structured to match the organization of template semantics. The translator produces different output for different template-parameter values. A step in NuSMV corresponds to a micro-step in the original specification. We use the two case studies to illustrate our approach.

A. Main Module

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

Fig. 13. Main Module for a Composed HTS

Figure 13 shows the `main` module for the heating system, which uses STATEMATE semantics. Comments indicate which module instances contain variables and which are only collections of macros and invariants. The elements `iss`, `pss_en`, `iss_en` are instances of modules used as records and contain only macro definitions, which means they do not add to the system state space. The element `iss_exe` contains mostly 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 names beginning with "_" denote modules that contain only statements and no variables or macros. We briefly overview the parts of this module and describe the sub-modules in more detail in later subsections.

Snapshot `pss` is the result of the previous micro-step, and `iss` results from reseting `pss`. In each step, the system reacts to the contents of snapshot `iss`, to produce the *next* assignment to `pss`.

We say that `pss_en`, `iss_en` have type `en` to mean that they contain a boolean macro for every transition, HTS, and composed component in the original specification to indicate whether these entities are enabled in a certain snapshot (`pss` or `iss`). Module `enabled` sets these macros and realizes the template definition ***enabled_trans***. Macro `stable` uses `pss_en` to test whether the system is enabled after the execution of a micro-step. Other modules use `iss_en` to learn which entities are enabled in the reset snapshot `iss`.

Module `reset` realizes the template definition ***reset*** and sets `iss` to be the result of reseting snapshot `pss`. When the system is stable, module `reset` removes old data from the snapshot elements and non-deterministically chooses new values for input events and input variables; otherwise, `iss` is set to equal `pss`. If a notation's macro-semantics are simple rather than stable, the `main` module can be simplified to not use `stable` and `pss_en`.

Module `execute` decides which entities execute in the current step based on the macros identifying enabled entities in `iss_en`. We say the entity `iss_exe` is of type `exe` to mean that it contains a boolean macro for every transition, HTS, and composed component in the specification to indicate whether these entities are executed in the current step. The module `execute` sets these macros.

Finally, module `init` initializes the snapshot elements, and module `apply` realizes template definition ***apply***, which sets the next value of snapshot `pss` based on the macros identifying executing entities in `iss_exe`.

In the following, we provide more details for modules `snapshot, enabled, reset, execute,` and `apply`, including how `execute` is defined for each composition operator. The records of type `en` and `exe` may contain additional fields depending on the specification's composition operators (e.g., rendezvous composition adds macros for the occurrence of rendezvous events). The modules' details vary for different template-parameter values.

B. Snapshot

Recall that a snapshot is an 8-tuple

$$\langle CS, IE, AV, O, CS_a, IE_a, AV_a, I_a \rangle .$$

The module `snapshot` declares a sub-module for each snapshot element used in a notation's semantics. In STATEMATE, only the snapshot elements CS, IE, AV, I_a and O are used. Figure 14 shows the corresponding `snapshot` module, where data types `states`, `variables`, `intEvents`, `envInputs`, and `outputs` are sub-modules that group the specification's states, variables, internal events, inputs, and outputs, respectively. Snapshot elements of the same type have the same sub-module type. Thus in a notation using snapshot elements CS_a, CS and CS_a would both be instances of the module `states`.

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

Fig. 14. Snapshot Module (STATEMATE)

In the `main` module (Figure 13), snapshot `pss` is an instance of module `snapshot`. Snapshot `iss` contains the same fields as `pss`, but is defined using macros rather than variables, so it does not contribute to the state space.

In the module `states`, we declare for each HTS an enumerated variable with one value for each of the HTS's basic states, and a value `noState` indicating that the HTS is not active. We also define a boolean macro for each basic and super-state that indicates whether that state is in the current states of the snapshot. Figure 15

shows the `states` module declarations for the furnace HTS.

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

Fig. 15. Snapshot Sub-module for CS (Heating System)

In the module `variables`, a variable is declared for each of the specification's data variables. Currently, our translator supports variables of types boolean, enumerated, and integer range. In the modules `intEvents`, `envInputs`, and `outputs`, a boolean variable is declared for each of the specification's events to indicate the occurrence of the event in the snapshot.

C. Enabled

Module `enabled` defines the record type, `en`, used for `pss_en` and `iss_en`, whose fields indicate the enabled entities. Figure 16 shows module `enabled` for the heating system. The structure of an `en` record follows the specification hierarchy, in that there is a field (a macro) for each HTS transition (e.g., `furnace.t1`), for each HTS (e.g., `furnace.any`), and for each composed HTS (e.g., `house.any`). A sub-module for each HTS sets the HTS's record fields. The `enabled` module sets the fields for the composed HTSs. Interrupt composition adds fields (`roomTrans`) and a sub-module (`enabled_roomTrans`) for determining whether the interrupt transitions are enabled. Dependencies among the fields flow up through the specification hierarchy, with enabled transitions implying enabled HTSs implying enabled components.

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

Fig. 16. Enabled Module for the Heating System

Figure 17 shows how sub-module `enabled_furnace` determines if transition *t1* in the furnace HTS is enabled. The macros define fields in the record `en`, and their semantics reflect the template parameters that specify how states, events, and variables enable transitions. In STATEMATE semantics, a transition is enabled if its source state is a current state (i.e., in snapshot element CS), its triggering events are occurring (i.e., in IE or I_a, depending on whether they are internal or external events), and its guard condition is entailed by the current variable values (i.e., AV). The translator defines four macros for each transition in the HTS:

three macros test the three transition-enabling conditions, based on the template-parameter values for enabling transitions; and one macro that uses the test results to set the transition's field in `en`. The macro `any` indicates that the HTS is enabled when any of its transitions are enabled.

```
MODULE enabled_furnace(ss)
  DEFINE
    enStates_t1 := ss.CS.in_furnaceOff;
    enEvents_t1 := ss.IE.activate;
    enCond_t1 := 1;
    ...
    t1 := enStates_t1 & enEvents_t1 & enCond_t1;
    ...
    any := t1|t2|t3|t4|t5|t6|t7;
```

Fig. 17. Enabled Sub-module for the Furnace HTS

D. Reset

Figure 18 shows the `reset` module, which declares a sub-module for each snapshot element. A sub-module `resetXX` represents template parameter ***resetXX***, which specifies how to reset snapshot element ***XX***. For notations with stable macro-step semantics, each of these sub-modules also depends on the value of the macro `stable`.

```
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);
```

Fig. 18. Reset Module (STATEMATE)

Figure 19 shows how sub-module `resetIa` maintains information on one external event, `furnaceFault`, of the heating system. The sub-module defines a macro for each of the specification's external events, which is used to indicate if the event occurred and can trigger transitions. when the system is stable, the macro is set to the value of its respective event in the system input `I.ev`; otherwise, the macro's value is unchanged (i.e., is the same as stored in parameter snapshot `ss`). This method of using a macro representing "stable" to decide when to make variable assignments is adapted from the work of Chan et al. [14]. Other `resetXX` sub-modules are defined similarly.

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

Fig. 19. Reset Sub-module for I_a for the Heating System

E. Execute

Module `execute` defines the record type `exe` used for `iss_exe`, whose fields indicate which of the enabled entities execute in the current step. Figure 20 shows the module `execute` for the heating system. The structure of an `exe` record follows the specification hierarchy, in that there is a field (a macro) for each HTS transition (e.g., `furnace.t1`), for each HTS (e.g., `furnace.any`), and for each composition (e.g., `house.any`). A sub-module for each HTS constrains whether the HTS's transitions and the HTS execute. A sub-module for each composition takes as parameters the `en` and `exe` sub-records for its two operands and the `en` sub-record for itself; and constrains fields in the `exe` record. (The `en` record is for the reset snapshot `iss`.) Several composition operators, if used, add fields to the `execute` record; interrupt composition adds a sub-module, to constrain the interrupt transitions. The invariant ensures that each NuSMV step corresponds to a micro-step and not an idle step to represent the stable macro-step semantics of STATEMATE.

```
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
   roomTrans:  execute_roomTrans(en.roomTrans);
   room:   interrupt(en.noHeatReq, en.heatReq, en.roomTrans, noHeatReq, heatReq, roomTrans, en.room);
   house:  parallel(en.room, en.controller, room, controller, en.house);
   heatingSys:  parallel(en.house, en.furnace, house, furnace, en.heatingSys);
 INVAR
   -- require diligence
   en.heatingSys.any -> heatingSys.any
```

Fig. 20. Execute Module for the Heating System

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

Fig. 21. Execute Sub-module for the Furnace HTS

Figure 21 shows part of sub-module `execute_furnace`. Each HTS sub-module for `exe` declares a variable `tran` whose enumerated type has a value for each of the HTS's transitions, plus the value `noTranExe`, to represent when no transitions are taken in the HTS in a step. Macros are defined in the `exe` record for each

transition to indicate whether it executes in the step (e.g., `t1`). The assignment of a value to `tran` is constrained by the sub-module's invariant to be a highest-priority enabled transition. The priority scheme is specified using template parameter ***pri***. In the *furnace* HTS, which uses STATEMATE's priority scheme, transitions *t7* and *t6* have higher priority than *t1*, because their scopes are nearer the top of the state hierarchy. Therefore, *t1* can execute only if it is enabled and *t7* and *t6* are not enabled. The invariant constrains the HTS from executing if none of its transitions are enabled.

F. Apply

Figure 22 shows the module `apply`, which changes the next value of snapshot `pss` based on the effects of the executing transitions. The module declares a sub-module for each snapshot element; the sub-module updates that element according to the relevant template parameter ***nextXX***.

```
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);
```

Fig. 22. Apply Module (STATEMATE)

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

Fig. 23. nextCS Sub-module for the Heating System

Figure 23 shows part of the sub-module `nextCS` for the heating system. In STATEMATE semantics, the next state in each HTS is the default basic-state descendent of the destination state of the executing transition; if no transition executes, the state has the value in the reset snapshot `iss`. Sub-module `nextCS` contains a *next* statement for each HTS in the specification.

For the `nextAV` module, we have to consider possible overflow and underflow for integer range variables, and possible conflicting assignments. SMV reports a syntax error for each possible overflow or underflow error for integer range variables. For example, in Figure 24, the execution of transition *t1* might cause variable *a* to overflow, and the execution of transition *t2* might cause it to underflow. Figure 25 shows the sub-module `nextAV` for the composed HTS in Figure 24 with STATEMATE semantics. We introduce for each transition (e.g., *t1*) that updates each integer range variable (e.g., *a*) two macros. One macro (e.g., `at1`) captures the value of the variable assignment when this transition executes; if no underflow or overflow happens, the variable is assigned the value resulting from executing the transition, otherwise, it keeps the original value. A second macro

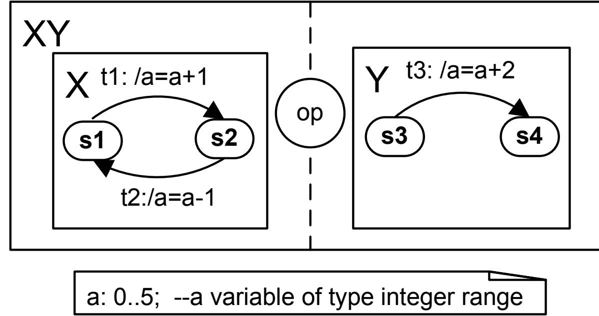


Fig. 24. Resolve conflicts

```
MODULE nextAV(pss,iss,exe)
  DEFINE
    at1 := case
      ((((iss.AV.a) + (1)))>=0)&((((iss.AV.a) + (1)))<=5) : ((iss.AV.a) + (1));
      1 : a;
    esac;
    at1error := exe.XHts.t1 & ((((iss.AV.a) + (1)))<0)|((((iss.AV.a) + (1)))>5) ;

  ASSIGN
    next(pss.AV.a):=case
      exe.XHts.t1 & exe.YHts.t3 : {at1, at3};
      exe.XHts.t2 & exe.YHts.t3 : {at2, at3};
      exe.XHts.t1 : at1;
      exe.XHts.t2 : at2;
      exe.YHts.t3 : at3;
      1 : iss.AV.a;
    esac;

  ASSIGN
    next(pss.AV.error_variables):= iss.AV.error_variables | at1error | at2error | at3error;
```

Fig. 25. nextAV Sub-module for Figure 24

(e.g., `at1error`) indicates whether underflow or overflow happens when a transition executes. In Figure 25, when *t1*'s execution updates the value of *a*, if no underflow or overflow happens, it is assigned according to *t1*'s actions; otherwise, it is not changed, however, at the same time, the macro `at1error` is set to 1, indicating an exception happens.

In addition, there is a variable called `error_variables` in the `nextAV` module to catch any overflow or underflow errors of the system. Initially, it is 0; thereafter, 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., `at1error`), 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 property AG !(error_variables).

STATEMATE permits conflicting variable assignments, thus sub-module `nextAV` uses the ***resolve*** template parameter to resolve the conflicts. In STATEMATE semantics, conflicts are resolved non-deterministically. For example, the execution of transition *t1* and *t2* in HTS *X*, and transition *t3* in HTS *Y* update variable *a*'s value. It is possible that transition *t1* and *t3*, or transition *t2* and *t3* execute in the same step. If this situation happens, an assignment is chosen non-deterministically. 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, for the system of Figure 24, if both HTSs *X* and *Y* have a transition execute that update the variable (e.g., *t1* and *t3* execute), the next value of variable *a* is assigned non-deterministically to either the result of executing the transition in HTS *X* (e.g., macro `at1`), or the result of executing the transition in HTS *Y* (e.g., macro `at3`); 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 value in the reset snapshot `iss`.

The remaining sub-modules of apply (e.g., `nextIE`) are defined in a similar manner considering the effects of the executing transitions on the snapshot elements.

G. Composition Operators

The semantics of each composition operator determine whether each of its components is allowed to execute. In this section, we describe how the NuSMV modules representing the composition operators of template semantics use the `exe` macros to constrain the behaviour of their components. Because we are only describing composition at the micro-step level, all operators are diligent, in other words, if a component is enabled it must execute according to the behaviour of the composition operator.

As illustrated in the `execute` module (Figure 20), all the composition operators of template semantics are represented using NuSMV synchronous composition of modules. The sub-modules of the record `exe` constrain the possible execution of components and transitions. The module for each composition operator takes as arguments the enabled and execution records of its left and right component, and the enabled status of itself. Each composition module uses an invariant to constrain the `exe` flags to represent the behaviour of composition. Some composition operators require the addition of information to the `exe` or `en` sets of macros, but these are only added as needed.

1) Interleaving

Figure 26 shows the module representing interleaving composition. The invariant disallows both components from executing in the same micro-step. In all composition modules, the field of `exe` for the composed component (`any`) is equal to the disjunction of the `exe` macros for its two components. All compositions also contain the invariant `any` $\rightarrow$ `enMe.any` to enforce the constraint that a component only executes if it is enabled.

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

Fig. 26. Interleaving Composition

NuSMV provides asynchronous composition of modules which is similar to interleaving but does enforce diligence.

2) Parallel

NuSMV's synchronous composition of modules matches the meaning of one 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. These flags are referenced by higher-level components to determine the behaviour of the overall system.

Figure 27 shows the modules representing the two types of parallel composition in template semantics. In the first one, called `parallel`, if both components are enabled, both must execute. The second, called `paraHarel`, captures the meaning of parallel composition in Harel's original definition of statecharts [8], which allows only one component to execute even if both are enabled. No additional invariant is needed to capture this behaviour because the `execute` module enforces diligence (i.e., if the system is enabled then it must execute), which in this case means that one of `exeLeft.any` or `exeRight.any` must be true.

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

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

Fig. 27. Parallel Composition

3) Environmental Synchronization

In environmental synchronization composition, two components can take a step if they both take transitions triggered by a *synchronization* event; otherwise their behaviour is interleaved (taking transitions not based on a synchronization event). The NuSMV module representing environmental synchronization is customized for the particular synchronization set. If environmental synchronization is used multiple places in the specification, the synchronization set at a higher level must explicitly be made part of the set at a lower level to represent the semantics of some notations such as LOTOS.

Figure 28 shows a NuSMV module for environmental synchronization on the event set $\{a, b\}$. We introduce extra elements in the `exe` record to indicate whether the component is executing transitions triggered by a synchronization event (e.g., `exeLeft.a_trig` to indicate the left component is 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 can only take transitions triggered by a single synchronization event. Finally, if transitions are not being taken on synchronization events, then only one of the components can execute.

Within the execute sub-module for an HTS, the macro that indicates whether an HTS is taking a transition triggered by a particular synchronization event (e.g., `a_trig`) is defined as the disjunction of the execution

```
MODULE env_sync_a_b(enLeft,enRight,exeLeft,exeRight,enMe)
  DEFINE
    any := exeLeft.any | exeRight.any;
    a_trig := exeLeft.a_trig & exeRight.a_trig;
    b_trig := exeLeft.b_trig & exeRight.b_trig;
  INVAR
     (any -> enMe.any)
     -- 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 & exeRight.b_trig)
   & !(exeLeft.b_trig & exeRight.a_trig)
     -- interleaved behaviour
   & (!(a_trig|b_trig) -> !(exeLeft.any & exeRight.any))
```

Fig. 28. Environmental Synchronization on Events a and b

macros of all the transitions that are triggered by the synchronization event. When there are composition operators at lower-levels in the hierarchy than environmental synchronization, fields are added to the `exe` macro to indicate whether components at each level in the hierarchy are taking transitions triggered by a synchronization event. This macro is equal to the disjunction of the macros at the lower levels. Figure 31 shows both how the HTS for *redA* has the extra macro for synchronization event `entRedA_trig`, and the additions to the `execute` module for the single lane bridge because environmental synchronization is used in the specification.

```
MODULE execute_redA(en)
  DEFINE
    ...
    entRedA_trig:=t1;
    ...

MODULE execute(en)
  DEFINE
    ...
    redCar.entRedA_trig := redA.entRedA_trig | redB.entRedA_trig;
    redCar.entRedA_trig := redA.entRedA_trig | redB.entRedA_trig;
    bridge.entRedA_trig := car.entRedA_trig | bridgeStatus.entRedA_trig;
```

Fig. 29. Execute Macros for Environmental Synchronization (Single Lane Bridge)

4) Rendezvous

Rendezvous composition means that 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 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 events, so the NuSMV module is customized for the rendezvous set.

Figure 30 shows the NuSMV module for rendezvous synchronization with rendezvous events $\{a, b\}$. The translator adds macros to the `exe` record that record whether a rendezvous is occurring in this micro-step (e.g., `a_rend`), and macros that indicate whether a component is taking a transition that is triggered by, or

```
MODULE rend_sync_a_b(enLeft,enRight,exeLeft,exeRight,enMe)
  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
     (any -> enMe.any)
     -- 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))
```

Fig. 30. Rendezvous Synchronization on Events a and b

generates a synchronization event (e.g., `a_trig`, `a_gen`). A rendezvous occurs if the left component executes a transition triggered by a and the right component generates a, or vice versa for each synchronization event. The invariant enforces the constraint that in a rendezvous the left component must be triggered on an event when the right component generates it, 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 additional 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. The extra macro `more_than_one` has to be defined for all components lower in the hierarchy as 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 (i.e., `exeLeft.any & exeRight.any`). In an HTS, it is always the case that only one transition can be taken.

```
MODULE execute_redA(en)
  DEFINE
    more_than_one:=0;
    inRed_trig:=0;
    inRed_gen:=t1;
    ...

MODULE execute(en)
  DEFINE
  ...
  redCar.more_than_one := redA.more_than_one | redB.more_than_one | (redA.any & redB.any);
  redCar.inRed_trig := redA.inRed_trig | redB.inRed_trig;
  redCar.inRed_gen := redA.inRed_gen | redB.inRed_gen;
  car.more_than_one := redCar.more_than_one | blueCar.more_than_one | (redCar.any & blueCar.any);
  car.inRed_trig := redCar.inRed_trig | blueCar.inRed_trig;
  car.inRed_gen := redCar.inRed_gen | blueCar.inRed_gen;
  ...
```

Fig. 31. Execute Macros for Rendezvous Composition (Single Lane Bridge)

Rendezvous composition also effects the way transitions are enabled. We have to represent how a transition

in one component generates a synchronization event that enables a transition in another component within the *same* step. Figure 32 shows part of the `enabled` module for the single lane bridge, which uses rendezvous synchronization. We introduce an macro (e.g., `sync_events.inRed`) to be equal to the disjunction of the enabled status of all transitions that generate each rendezvous 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 *t17* of the *bridge* HTS depends on the rendezvous macro for *inRed* rather than its status in the snapshot. In fact, we can eliminate rendezvous events from the snapshot.

```
MODULE enabled(ss)
  VAR
    redA : enabled_redA(ss, sync_events);
    redB : enabled_redB(ss, sync_events);
    blueA : enabled_blueA(ss, sync_events);
    blueB : enabled_blueB(ss, sync_events);
    bridgeStatus : enabled_bridgeStatus(ss, sync_events);
    redCoordEnt : enabled_redCoordEnt(ss);
    ...
  DEFINE
    sync_events.inRed := redA.t1| redB.t3;
    ...

MODULE enabled_bridgeStatus(ss, sync_events)
  DEFINE
    enStates_t17 := ss.CS.in_empty ;
    enEvents_t17 := sync_events.inRed;
    enCond_t17 := 1;
    t17 := enStates_t17 & enEvents_t17 & enCond_t17;
    ...
```

Fig. 32. Enabled Modules for the Single Lane Bridge

5) 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 33 shows the module representing choice composition. A boolean variable `choice` is declared to capture the choice of which component executes.

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

Fig. 33. Choice Composition

6) Sequence

In sequence composition, the first component executes in isolation until it reaches its final states, and then the second component executes in isolation. (No transitions leave the final states of an HTS, and the left component must have final states to be used in sequence composition.)

Figure 34 shows the module representing sequence composition. If this form of composition is used in the specification, additional macros called `final` are added to the `en` record. For an HTS, the macro is set to be the disjunction of the macros indicating whether the HTS is in a final state. To pass this information through the hierarchy, we add a `final` macro in the `en` record for each component that is a descendent of sequence equal to the conjunction of the macros indicating whether the sub-components are in final states. For sequence composition, the machine is in its final state if its right component is in its final states. Using these macros from the `en` record, the left component in sequence composition can only execute if it is not in its final states.

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

Fig. 34. Sequence Composition

7) Interrupt

Interrupt composition allows control to pass between two components via a provided set of interrupt transitions. Figure 35 shows the module `interrupt` to represent interrupt composition. Interrupt composition has additional parameters, `enIntrTrans` and `exeIntrTrans`, which are sub-records of `en` and `exe`, and hold the enabling and execute flags for the interrupt transitions.

In interrupt composition, either the left component, the right component, or an interrupt transition is chosen to execute, based on which has the highest priority. Therefore, the invariant in interrupt composition depends on the value of template parameter ***pri***, which specifies the priority scheme. When a specification uses interrupt

```
MODULE interrupt(enLeft,enRight,enIntrTrans,exeLeft,exeRight,exeIntrTrans,enMe)
  DEFINE
    any := exeLeft.any | exeRight.any | exeIntrTrans.any;
  INVAR
     (any -> enMe.any)
     -- 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)
```

Fig. 35. Interrupt Composition

composition, the translator adds, for each HTS and each subcomponent within the interrupt composed component, a priority field in the `en` record. This field records the priority of the highest-priority enabled transition in that entity. Figure 36 shows the priority fields that are added to sub-module `enabled_furnace` for the *furnace* HTS. For STATEMATE semantics, transition priorities are based on the scopes of the transitions; field `pri` is set to the highest priority of the enabled transitions. Lower `pri` values denote higher priorities, with constant `MAX_PRI` representing the lowest priority in the system.

```
MODULE enabled_furnace(ss)
  DEFINE
    ...
    pri_t1 := 2;  -- rank of scope
    pri_t6 := 1;
    pri_t7 := 1;
    pri := case
      t1 & !t6 & !t7: pri_t1;
      ...
      t6 : pri.t6;
      t7 : pri.t7;
      1 : MAX_PRI;
    esac;
    ...
```

Fig. 36. Priority Macros for the Heating System

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 37 shows how the `nextCS` module for the heating system is modified to accommodate interrupt transition *t19* and *t20*. For example, 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. Interrupt composition may also change other `nextXX` sub-modules of `apply`, if interrupt transition perform actions that updates these snapshot elements.

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

Fig. 37. Module nextCS for Interrupt Transitions

VI. Evaluation

We used our two case studies, the heating system and single lane bridge, to evaluate our translator. These examples 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 SMV models generated by our translator for the two case studies can be found in the appendices.

Previous sections showed how the translator preserves the structure of the specification, and the names of the control states, events, and variables.

Table II shows how elements of a composed HTS are translated into NuSMV variables to demonstrate that our translation does not increase the state space of the specification (except for one variable for choice composition). Table III shows 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 was calculated by NuSMV.

Specification Element	NuSMV Variables
m transitions in an HTS	1 variable with (m+1) values
p basic states in an HTS	1 variable with (p+1) values
n super-states in an HTS	0
q typed variables	q typed variables
r non-rendezvous events	r boolean variables
1 HTS	0
1 Composition Operator	0 (except 1 for choice)

TABLE II. Composed HTSs as Translated to NuSMV Variables

Variable Category	Number of NuSMV Variables	
	single lane bridge	heating system
transition-related	9	5
state-related	9	4
variable-related	2	7
event-related	8	8
state space	1.37106e+15	2.52256e+14
reachable state space	7.87212e+5	1.03840e+5

TABLE III. Case Study Statistics

To test the correctness of our translator we verified a set of properties in NuSMV for each of the case studies. The following are examples of properties we checked on the generated SMV model of the single lane bridge specification[3]:

1) A red car and a blue car cannot be on the bridge the bridge at the same time.

```
AG !(pss.AV.numRed>0 & pss.AV.numBlue>0)
```

2) A blue car cannot enter if the red car is on the bridge.

```
AG ((pss.CS.redAHts_state=onRedA | pss.CS.redBHts_state=onRedB)
    -> !(pss.CS.blueAHts_state=onBlueA | pss.CS.blueBHts_state=onBlueB))
```

[3]NuSMV checks properties in both CTL and LTL.

3) A car cannot pass another car on the bridge.

```
(((pss.CS.redAHts_state=onRedA & pss.CS.redBHts_state=waitRedB)
 & X (pss.CS.redBHts_state=onRedB))
     ->
(pss.CS.redBHts_state=onRedB
 U (pss.CS.redAHts_state=waitRedA & !(pss.CS.redBHts_state=onRedB))))
```

The following are examples of properties we checked on the generated SMV model of the heating system specification:

1) The furnace will eventually be on if a room requests heat.

```
G (pss.AV.requestHeat -> F (pss.CS.furnaceHts_state=furnaceRun))
```

2) The furnace will eventually be off if no room requests heat.

```
G (!pss.AV.requestHeat & pss.CS.in_heaterActive
     -> F (pss.CS.furnaceHts_state=furnaceOff))
```

3) If the furnace fails, it will not start before the user resets it.

```
((pss.CS.furnaceHts_state=furnaceErr)
  -> (pss.CS.furnaceHts_state=furnaceErr) U furnaceReset)
```

Because our translator uses the same names of variables, events and states, and because one step of the NuSMV model exactly corresponds to a micro-step, the counterexamples produced as we debugged our specification can be easily understood in terms of the original system description.

VII. Related Work

To check a specification written in a notation, one can either construct an analyzer for the notation, or translate the notation into the input languages of existing analyzers. Examples of translators for specific notations include SCR to the EMC model checker [11], RSML to SMV [14], SCR to SPIN and SMV [13], and the work of Avrunin *et al* to translate several notations into hybrid automata [12].

To reduce the number of translators and to facilitate analysis using multiple tools, intermediate languages have been introduced, such as SAL [15], IF [16], Action Language [21], Bandera Intermediate Representation (BIR) [22], and CDL [23]. In most of these cases, there are translators from several notations to the intermediate language, and to and from the intermediate language and verification tools. Compared to these approaches, template semantics are parameterized and contain a richer set of composition operators. By providing a translator from template semantics descriptions of a notation's semantics to NuSMV, it is possible to model check a wide range of model-based notations. We can even create new notations by combining existing parameter values without any change to our translator. Bogor [24] is the only other parameterized translator of which we are aware, but it is a back-end translator for BIR for creating optimized model checkers.

There are other approaches for generating a model or analysis tool from a description of a notation's semantics. Day and Joyce [25] embedded the semantics of a notation in higher-order logic and produced a next-state relation in logic for a specification. Pezzè and Young [26] 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 [27]. Compared to these approaches, template semantics allows one to only specify how a notation differs from the common features of model-based notations rather than providing a complete semantic description.

VIII. Conclusion

We have described a fully automated translator that takes the template semantics description of a notation's meaning, a specification in the notation, and produces a NuSMV model for the specification. Using our translator, we can model check specifications written in a large range of model-based notations. A secondary, but key, contribution of the report is showing how to represent a rich collection of composition operators within the language features provided by NuSMV. Our translator neither adds to nor abstracts the state space of the original specification.

Our translator implements a fixed set of commonly-used parameter values and composition operators. It does not yet support template-parameter values of event queues, as are found in message-passing languages such as SDL [28]. Also, our translator assumes that variables in the specification are of types supported in NuSMV (booleans, enumerated types, finite ranges of integers).

The modularity of our translation method (matching the modularity of template semantics) is such that the addition of a new parameter value or composition operator has a localized effect on the existing implementation. For example, adding a new composition operator would involve the creation of only one new SMV module.

Template semantics provides a means of classifying notations. We plan to explore how it can be used for determining which model checker is the "best fit" for analyzing a specification. We are also working on a method for handling arbitrary template-parameter values and composition operators.

References

[1] J. Niu, J. M. Atlee, and N. A. Day, "Template semantics for model-based notations," *IEEE Trans. on Soft. Eng.*, vol. 20, no. 10, pp. 866–882, Oct. 2003.

[2] ——, "Understanding and comparing model-based soecifi cation notations," in *IEEE Intl. Requirements Eng. Conf.* IEEE Comp. Soc. Press 2003, pp. 188–199.

[3] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri, "NuSMV: A new symbolic model checker," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 410–425, 2000.

[4] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems: Specifi cation*. Springer-Verlag, 1991.

[5] C. A. R. Hoare, *Communicating Sequential Processes*. UK: Prentice Hall, 1985.

[6] R. Milner, *Communication and Concurrency*. New York: Prentice Hall, 1989.

[7] ISO8807, "LOTOS - a formal description technique based on the temporal ordering of observational behaviour," ISO, Tech. Rep., 1988.

[8] D. Harel *et al.*, "On the formal semantics of statecharts," in *Symp. on Logic in Comp. Sci.*, 1987, pp. 54–64.

[9] R. Cleaveland, J. Parrow, and B. Steffen, "The Concurrency Workbench: A semantics-based tool for the verifi cation of concurrent systems," *ACM Trans. on Prog. Lang. and Syst.*, vol. 15, no. 1, pp. 36–72, Jan 1993.

[10] D. Harel *et al.*, "STATEMATE: A working environment for the development of complex reactive systems," *IEEE Trans. on Soft. Eng.*, vol. 16, no. 4, pp. 403–414, April 1990.

[11] J. M. Atlee and J. Gannon, "State-based model checking of event-driven system requirements," *IEEE Trans. on Soft. Eng.*, vol. 19, no. 1, pp. 24–40, 1993.

[12] G. Avrunin, J. Corbett, and L. Dillon, "Analyzing partially-implemented real-time systems," in *Int. Conf. on Soft. Eng.* ACM Press, 1997, pp. 228–238.

[13] R. Bharadwaj and C. L. Heitmeyer, "Model checking complete requirements specifi cations using abstraction," *Auto. Soft. Eng.*, vol. 6, no. 1, pp. 37–68, 1999.

[14] W. Chan, R. J. Anderson, P. Beame, S. Burns, F. Modugno, and D. Notkin, "Model checking large software specifi cations," *IEEE Trans. on Soft. Eng.*, vol. 24, no. 7, pp. 498–519, 1998.

[15] S. Bensalem *et al.*, "An overview of SAL," in *Langley Formal Methods Workshop*. Center for Aerospace Information, NASA, 2000, pp. 187–196.

[16] M. Bozga, S. Graf, L. Mounier, and J. Sifakis, "The intermediate representation IF: syntax and semantics," Verimag, Grenoble, Tech. Rep., 1999.

[17] J. Magee and J. Kramer, *Concurrency, State Models & Java Programs*. UK: John Wiley & Sons, 1999.

[18] N. Day, "A framework for multi-notation, model-oriented requirements analysis," Ph.D. dissertation, Univ. of British Columbia, October 1998.

[19] D. Harel and A. Naamad, "The Statemate semantics of statecharts," *ACM Trans. on Soft. Eng. Meth.*, vol. 5, no. 4, pp. 293–333, 1996.

[20] K. McMillan, *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic, 1993.

[21] T. Bultan, "Action language: A specification language for model checking reactive systems," in *Int. Conf. on Soft. Eng.*, 2000, pp. 335–344.

[22] J. C. Corbett, M. B. Dwyer, J. Hatcliff, and et al., "Bandera: Extracting finite-state models from java source code," in *Int. Conf. on Soft. Eng.*, 2000, pp. 439–448.

[23] S. Katz and O. Grumberg, "A framework for translating models and specifications," in *Integrated Formal Methods: Third Intl. Conf., volume 2335 of LNCS*. Springer-Verlag, 2002, pp. 145–164.

[24] Robby, M. B. Dwyer, and J. Hatcliff, "Bogor: An extensible and highly-modular software model checking framework," in *ESEC/FSE2003*. ACM Press, 2003, pp. 267–276.

[25] N. A. Day and J. J. Joyce, "Symbolic functional evaluation," in *Theorem Proving in Higher Order Logics*, ser. LNCS, vol. 1690. Springer-Verlag, 1999, pp. 341–358.

[26] M. Pezzè and M. Young, "Creating of multi-formalism state-space analysis tools," in *Int. Symp. on Soft. Testing and Analysis*. ACM Press, 1996, pp. 172–179.

[27] L. Dillon and R. Stirewalt, "Inference graphs: a computational structure supporting generation of customizable and correct analysis components," *IEEE Trans. on Soft. Eng.*, vol. 29, no. 2, pp. 133–150, Feb 2003.

[28] ITU-T, "Recommendation Z.100. Specification and Description Language (SDL)," International Telecommunication Union - Standardization Sector, Tech. Rep. Z-100, 1999.

Appendix

A. Heating System SMV Model (automatically generated by the translator)

```
MODULE states
  VAR
    noHeatReqHts_state : {idleNoHeat,waitForHeat,noState};
    heatReqHts_state : {idleHeat,waitForCool,noState};
    controllerHts_state : {off,error,idle,actHeater,heaterRun,noState};
    furnaceHts_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 := noHeatReqHts_state=idleNoHeat;
    in_waitForHeat := noHeatReqHts_state=waitForHeat;
    in_heatReq := in_idleHeat | in_waitForCool;
    in_idleHeat := heatReqHts_state=idleHeat;
    in_waitForCool := heatReqHts_state=waitForCool;
    in_controller := in_off | in_error | in_controllerOn;
    in_off := controllerHts_state=off;
    in_error := controllerHts_state=error;
    in_controllerOn := in_idle | in_heaterActive;
    in_idle := controllerHts_state=idle;
    in_heaterActive := in_actHeater | in_heaterRun;
    in_actHeater := controllerHts_state=actHeater;
    in_heaterRun := controllerHts_state=heaterRun;
    in_furnace := in_furnaceNormal | in_furnaceErr;
    in_furnaceNormal := in_furnaceOff | in_furnaceAct | in_furnaceRun;
    in_furnaceOff := furnaceHts_state=furnaceOff;
    in_furnaceAct := furnaceHts_state=furnaceAct;
    in_furnaceRun := furnaceHts_state=furnaceRun;
    in_furnaceErr := furnaceHts_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;
    furnaceStartupTime : 0..5;
    warmUpTime : 0..5;
    coolDownTime : 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_noHeatReqHts(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.warmUpTime) < (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.warmUpTime) = (5)) & (!((ss.AV.valvePos) = (2))));
    t18:=(enStates_t18)&(enEvents_t18)&(enCond_t18);


  DEFINE
    any := t15|t16|t17|t18;
  --noHeatReqHts 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_noHeatReqHts(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)
    &(any -> en.any)


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

MODULE enabled_heatReqHts(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.coolDownTime) < (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.coolDownTime) = (5)) & (!((ss.AV.valvePos) = (0))));
    t24:=(enStates_t24)&(enEvents_t24)&(enCond_t24);


  DEFINE
    any := t21|t22|t23|t24;
  --heatReqHts 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_heatReqHts(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)
    &(any -> en.any)


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

MODULE enabled_controllerHts(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
  DEFINE
    enStates_t9:=(ss.CS.in_off);
    enEvents_t9:=(ss.Ia.heatSwitchOn=1);
    enCond_t9 := 1;
    t9:=(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;
    t10:=(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;
    t11:=(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);
    t12:=(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;
    t13:=(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);
    t14:=(enStates_t14)&(enEvents_t14)&(enCond_t14);


  DEFINE
    any := t8|t9|t10|t11|t12|t13|t14;
MODULE execute_controllerHts(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);

  --define INVAR constrait to define which transition to be executed
  INVAR
     (t8 -> en.t8)
    &(t9 -> en.t9)
    &(t10 -> en.t10)
    &(t11 -> en.t11)
    &(t12 -> en.t12&!en.t8&!en.t9&!en.t10&!en.t11)
    &(t13 -> en.t13&!en.t8&!en.t9&!en.t10&!en.t11&!en.t12&!en.t14)
    &(t14 -> en.t14&!en.t8&!en.t9&!en.t10&!en.t11)
    &(any -> en.any)


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

MODULE enabled_furnaceHts(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;
    t1:=(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;
    t2:=(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.furnaceStartupTime) = (5));
    t3:=(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;
    t4:=(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.furnaceStartupTime) < (5));
    t5:=(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;
    t6:=(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;
    t7:=(enStates_t7)&(enEvents_t7)&(enCond_t7);


  DEFINE
    any := t1|t2|t3|t4|t5|t6|t7;
MODULE execute_furnaceHts(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);

  --define INVAR constrait to define which transition to be executed
  INVAR
     (t1 -> en.t1&!en.t6&!en.t7)
    &(t2 -> en.t2&!en.t6&!en.t7)
    &(t3 -> en.t3&!en.t6&!en.t7)
    &(t4 -> en.t4&!en.t6&!en.t7)
    &(t5 -> en.t5&!en.t6&!en.t7)
    &(t6 -> en.t6)
    &(t7 -> en.t7)
    &(any -> en.any)


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

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

```
  INVAR
     (any -> enMe.any)
    &(any -> ((enLeft.any -> exeLeft.any) & (enRight.any -> exeRight.any)))

MODULE enabled_roomHtsIntrTrans(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.coolDownTime) = (5)) & ((ss.AV.valvePos) = (0)))
                  & (((ss.AV.actualTemp) - (ss.AV.setTemp)) > (2)));
    t19:=(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.warmUpTime) = (5)) & ((ss.AV.valvePos) = (2)))
                 & (((ss.AV.setTemp) - (ss.AV.actualTemp)) > (2)));
    t20:=(enStates_t20)&(enEvents_t20)&(enCond_t20);


  DEFINE
    any := t19|t20;
  --roomHtsIntrTrans 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_roomHtsIntrTrans(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);

  --define INVAR constrait to define which transition to be executed
  INVAR
     (t19 -> en.t19)
    &(t20 -> en.t20)
    &(any -> en.any)


MODULE microInterr(enLeft,enRight,enIntrTran,exeLeft,exeRight,exeIntrTran,enMe)
  DEFINE
    any:=exeLeft.any | exeRight.any ;
  INVAR
     (any -> enMe.any)
    &(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
    noHeatReqHts_state := case
      stable : ss.CS.noHeatReqHts_state;
      1 : ss.CS.noHeatReqHts_state;
    esac;
  DEFINE
    heatReqHts_state := case
      stable : ss.CS.heatReqHts_state;
      1 : ss.CS.heatReqHts_state;
    esac;
  DEFINE
    controllerHts_state := case
      stable : ss.CS.controllerHts_state;
      1 : ss.CS.controllerHts_state;
    esac;
  DEFINE
    furnaceHts_state := case
      stable : ss.CS.furnaceHts_state;
      1 : ss.CS.furnaceHts_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 := noHeatReqHts_state=idleNoHeat;
    in_waitForHeat := noHeatReqHts_state=waitForHeat;
    in_heatReq := in_idleHeat | in_waitForCool;
    in_idleHeat := heatReqHts_state=idleHeat;
    in_waitForCool := heatReqHts_state=waitForCool;
    in_controller := in_off | in_error | in_controllerOn;
    in_off := controllerHts_state=off;
    in_error := controllerHts_state=error;
    in_controllerOn := in_idle | in_heaterActive;
    in_idle := controllerHts_state=idle;
    in_heaterActive := in_actHeater | in_heaterRun;
    in_actHeater := controllerHts_state=actHeater;
    in_heaterRun := controllerHts_state=heaterRun;
    in_furnace := in_furnaceNormal | in_furnaceErr;
    in_furnaceNormal := in_furnaceOff | in_furnaceAct | in_furnaceRun;
    in_furnaceOff := furnaceHts_state=furnaceOff;
    in_furnaceAct := furnaceHts_state=furnaceAct;
    in_furnaceRun := furnaceHts_state=furnaceRun;
    in_furnaceErr := furnaceHts_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
    furnaceStartupTime := case
      stable : ss.AV.furnaceStartupTime;
      1 : ss.AV.furnaceStartupTime;
    esac;

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

  DEFINE
    coolDownTime := case
      stable : ss.AV.coolDownTime;
      1 : ss.AV.coolDownTime;
    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
    noHeatReqHts : enabled_noHeatReqHts(ss);
    heatReqHts : enabled_heatReqHts(ss);
    controllerHts : enabled_controllerHts(ss);
    furnaceHts : enabled_furnaceHts(ss);

    roomHtsIntrTrans : enabled_roomHtsIntrTrans(ss);
  --define enabled macros for each composite HTSs
  DEFINE
    heatingSystemHts.any := houseHts.any | furnaceHts.any;
    houseHts.any := roomHts.any | controllerHts.any;
    roomHts.any := noHeatReqHts.any | heatReqHts.any | roomHtsIntrTrans.any;

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

MODULE execute(en)
  VAR
    noHeatReqHts : execute_noHeatReqHts(en.noHeatReqHts);
    heatReqHts : execute_heatReqHts(en.heatReqHts);
    controllerHts : execute_controllerHts(en.controllerHts);
    furnaceHts : execute_furnaceHts(en.furnaceHts);
    roomHtsIntrTrans : execute_roomHtsIntrTrans(en.roomHtsIntrTrans);
```

```
    heatingSystemHts : microParaDili(en.houseHts,en.furnaceHts,
                              houseHts,furnaceHts,en.heatingSystemHts);
    houseHts : microParaDili(en.roomHts,en.controllerHts,roomHts,controllerHts,en.houseHts);
    roomHts : microInterr(en.noHeatReqHts,en.heatReqHts,en.roomHtsIntrTrans,
                      noHeatReqHts,heatReqHts,roomHtsIntrTrans,en.roomHts);

  INVAR
    (en.heatingSystemHts.any -> heatingSystemHts.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.noHeatReqHts_state):=case
      exe.noHeatReqHts.t15 : waitForHeat;
      exe.noHeatReqHts.t16 : idleNoHeat;
      exe.noHeatReqHts.t17 : waitForHeat;
      exe.noHeatReqHts.t18 : waitForHeat;
      exe.roomHtsIntrTrans.t19 : idleNoHeat;
      exe.roomHtsIntrTrans.t20 : noState;
      1 : iss.CS.noHeatReqHts_state;
    esac;

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

  ASSIGN
    next(pss.CS.controllerHts_state):=case
      exe.controllerHts.t8 : off;
      exe.controllerHts.t9 : idle;
      exe.controllerHts.t10 : off;
      exe.controllerHts.t11 : error;
      exe.controllerHts.t12 : actHeater;
      exe.controllerHts.t13 : heaterRun;
      exe.controllerHts.t14 : idle;
      1 : iss.CS.controllerHts_state;
    esac;

  ASSIGN
    next(pss.CS.furnaceHts_state):=case
      exe.furnaceHts.t1 : furnaceAct;
      exe.furnaceHts.t2 : furnaceOff;
      exe.furnaceHts.t3 : furnaceRun;
      exe.furnaceHts.t4 : furnaceOff;
      exe.furnaceHts.t5 : furnaceAct;
      exe.furnaceHts.t6 : furnaceOff;
      exe.furnaceHts.t7 : furnaceErr;
      1 : iss.CS.furnaceHts_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 := case
  ((((iss.AV.valvePos) + (1)))<0)|((((iss.AV.valvePos) + (1)))>2) : 1;
  1 : 0;
 esac;
  DEFINE valvePost18 := case
  ((((iss.AV.valvePos) + (1)))>=0)&((((iss.AV.valvePos) + (1)))<=2) : ((iss.AV.valvePos) + (1));
  1 : iss.AV.valvePos;
 esac;
  DEFINE valvePost18error := case
  ((((iss.AV.valvePos) + (1)))<0)|((((iss.AV.valvePos) + (1)))>2) : 1;
  1 : 0;
 esac;
  DEFINE valvePost21 := case
  ((((iss.AV.valvePos) - (1)))>=0)&((((iss.AV.valvePos) - (1)))<=2) : ((iss.AV.valvePos) - (1));
  1 : iss.AV.valvePos;
 esac;
  DEFINE valvePost21error := case
  ((((iss.AV.valvePos) - (1)))<0)|((((iss.AV.valvePos) - (1)))>2) : 1;
  1 : 0;
 esac;
  DEFINE valvePost24 := case
  ((((iss.AV.valvePos) - (1)))>=0)&((((iss.AV.valvePos) - (1)))<=2) : ((iss.AV.valvePos) - (1));
  1 : iss.AV.valvePos;
 esac;
  DEFINE valvePost24error := case
  ((((iss.AV.valvePos) - (1)))<0)|((((iss.AV.valvePos) - (1)))>2) : 1;
  1 : 0;
 esac;
  ASSIGN
    next(pss.AV.valvePos):=case
      exe.noHeatReqHts.t15 : valvePost15;
      exe.noHeatReqHts.t18 : valvePost18;
      exe.heatReqHts.t21 : valvePost21;
      exe.heatReqHts.t24 : valvePost24;
      1 : iss.AV.valvePos;
    esac;

  --next state relation for  furnaceStartupTime
  DEFINE furnaceStartupTimet1 := case
  (((0))>=0)&(((0))<=5) : (0);
  1 : iss.AV.furnaceStartupTime;
 esac;
  DEFINE furnaceStartupTimet1error := case
  (((0))<0)|(((0))>5) : 1;
  1 : 0;
 esac;
  DEFINE furnaceStartupTimet5 := case
  ((((iss.AV.furnaceStartupTime) + (1)))>=0)&((((iss.AV.furnaceStartupTime) + (1)))<=5)
               : ((iss.AV.furnaceStartupTime) + (1));
  1 : iss.AV.furnaceStartupTime;
 esac;
  DEFINE furnaceStartupTimet5error := case
  ((((iss.AV.furnaceStartupTime) + (1)))<0)|((((iss.AV.furnaceStartupTime) + (1)))>5) : 1;
  1 : 0;
 esac;
  ASSIGN
    next(pss.AV.furnaceStartupTime):=case
      exe.furnaceHts.t1 : furnaceStartupTimet1;
      exe.furnaceHts.t5 : furnaceStartupTimet5;
      1 : iss.AV.furnaceStartupTime;
    esac;

  --next state relation for  warmUpTime
  DEFINE warmUpTimet15 := case
```

```
 (((0))>=0)&(((0))<=5) : (0);
 1 : iss.AV.warmUpTime;
esac;
 DEFINE warmUpTimet15error := case
 (((0))<0)|(((0))>5) : 1;
 1 : 0;
esac;
 DEFINE warmUpTimet17 := case
 ((((iss.AV.warmUpTime) + (1)))>=0)&((((iss.AV.warmUpTime) + (1)))<=5) : ((iss.AV.warmUpTime) + (1));
 1 : iss.AV.warmUpTime;
esac;
 DEFINE warmUpTimet17error := case
 ((((iss.AV.warmUpTime) + (1)))<0)|((((iss.AV.warmUpTime) + (1)))>5) : 1;
 1 : 0;
esac;
 DEFINE warmUpTimet18 := case
 (((0))>=0)&(((0))<=5) : (0);
 1 : iss.AV.warmUpTime;
esac;
 DEFINE warmUpTimet18error := case
 (((0))<0)|(((0))>5) : 1;
 1 : 0;
esac;
 ASSIGN
   next(pss.AV.warmUpTime):=case
     exe.noHeatReqHts.t15 : warmUpTimet15;
     exe.noHeatReqHts.t17 : warmUpTimet17;
     exe.noHeatReqHts.t18 : warmUpTimet18;
     1 : iss.AV.warmUpTime;
   esac;

 --next state relation for  coolDownTime
 DEFINE coolDownTimet21 := case
 (((0))>=0)&(((0))<=5) : (0);
 1 : iss.AV.coolDownTime;
esac;
 DEFINE coolDownTimet21error := case
 (((0))<0)|(((0))>5) : 1;
 1 : 0;
esac;
 DEFINE coolDownTimet23 := case
 ((((iss.AV.coolDownTime) + (1)))>=0)&((((iss.AV.coolDownTime) + (1)))<=5)
              : ((iss.AV.coolDownTime) + (1));
 1 : iss.AV.coolDownTime;
esac;
 DEFINE coolDownTimet23error := case
 ((((iss.AV.coolDownTime) + (1)))<0)|((((iss.AV.coolDownTime) + (1)))>5) : 1;
 1 : 0;
esac;
 DEFINE coolDownTimet24 := case
 (((0))>=0)&(((0))<=5) : (0);
 1 : iss.AV.coolDownTime;
esac;
 DEFINE coolDownTimet24error := case
 (((0))<0)|(((0))>5) : 1;
 1 : 0;
esac;
 ASSIGN
   next(pss.AV.coolDownTime):=case
     exe.heatReqHts.t21 : coolDownTimet21;
     exe.heatReqHts.t23 : coolDownTimet23;
     exe.heatReqHts.t24 : coolDownTimet24;
     1 : iss.AV.coolDownTime;
   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=1)
      | (exe.noHeatReqHts.t15) & (valvePost15error=1)
      | (exe.noHeatReqHts.t18) & (valvePost18error=1)
      | (exe.heatReqHts.t21) & (valvePost21error=1)
      | (exe.heatReqHts.t24) & (valvePost24error=1)
      | (exe.furnaceHts.t1) & (furnaceStartupTimet1error=1)
      | (exe.furnaceHts.t5) & (furnaceStartupTimet5error=1)
      | (exe.noHeatReqHts.t15) & (warmUpTimet15error=1)
      | (exe.noHeatReqHts.t17) & (warmUpTimet17error=1)
      | (exe.noHeatReqHts.t18) & (warmUpTimet18error=1)
      | (exe.heatReqHts.t21) & (coolDownTimet21error=1)
      | (exe.heatReqHts.t23) & (coolDownTimet23error=1)
      | (exe.heatReqHts.t24) & (coolDownTimet24error=1)

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

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

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

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

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

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

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

  --next state relation for output furnaceRunning
  ASSIGN
    next(pss.OO.furnaceRunning):=case
      exe.furnaceHts.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.noHeatReqHts_state):=idleNoHeat;
    init(pss.CS.heatReqHts_state):=idleHeat;
    init(pss.CS.controllerHts_state):=off;
    init(pss.CS.furnaceHts_state):=furnaceOff;
    init(pss.AV.valvePos):={0};
    init(pss.AV.setTemp):={20};
    init(pss.AV.actualTemp):={20};
    init(pss.AV.furnaceStartupTime):={0};
    init(pss.AV.warmUpTime):={0};
    init(pss.AV.coolDownTime):={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(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);

DEFINE
 stable := !(pss_en.heatingSystemHts.any);
```

B. Single Lane Bridge SMV Model (automatically generated by the translator)

```
MODULE states
  VAR
    redAHts_state : {waitRedA,onRedA,noState};
    redBHts_state : {waitRedB,onRedB,noState};
    blueAHts_state : {waitBlueA,onBlueA,noState};
    blueBHts_state : {waitBlueB,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_onRedA;
    in_waitRedA := redAHts_state=waitRedA;
    in_onRedA := redAHts_state=onRedA;
    in_redB := in_waitRedB | in_onRedB;
    in_waitRedB := redBHts_state=waitRedB;
    in_onRedB := redBHts_state=onRedB;
    in_blueCar := in_blueA | in_blueB;
    in_blueA := in_waitBlueA | in_onBlueA;
    in_waitBlueA := blueAHts_state=waitBlueA;
    in_onBlueA := blueAHts_state=onBlueA;
    in_blueB := in_waitBlueB | in_onBlueB;
    in_waitBlueB := blueBHts_state=waitBlueB;
    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
    gen : envEvents;
    asn : 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)));
    t1:=(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;
    t2:=(enStates_t2)&(enEvents_t2)&(enCond_t2);


  DEFINE
    any := t1|t2;
MODULE execute_redAHts(en)
  --transition declaration
  VAR
    tran : {t1_exe, t2_exe, noTran_exe};

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

  --define INVAR constrait to define which transition to be executed
  INVAR
```

```
     (t1 -> en.t1)
    &(t2 -> en.t2)
    &(any -> en.any)

  --define the env sync event triggering macros for single 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;

  --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 single Hts redAHts
  DEFINE
    inRed_trig:=0;
    outRed_trig:=0;
    inBlue_trig:=0;
    outBlue_trig:=0;

  --define the rend sync event generating macros for single Hts redAHts
  DEFINE
    inRed_gen:=t1;
    outRed_gen:=t2;
    inBlue_gen:=0;
    outBlue_gen:=0;


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

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)));
    t3:=(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;
    t4:=(enStates_t4)&(enEvents_t4)&(enCond_t4);


  DEFINE
    any := t3|t4;
MODULE execute_redBHts(en)
  --transition declaration
  VAR
    tran : {t3_exe, t4_exe, noTran_exe};

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

  --define INVAR constrait to define which transition to be executed
```

```
  INVAR
     (t3 -> en.t3)
    &(t4 -> en.t4)
    &(any -> en.any)

  --define the env sync event triggering macros for single 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;

  --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 single Hts redBHts
  DEFINE
    inRed_trig:=0;
    outRed_trig:=0;
    inBlue_trig:=0;
    outBlue_trig:=0;

  --define the rend sync event generating macros for single Hts redBHts
  DEFINE
    inRed_gen:=t3;
    outRed_gen:=t4;
    inBlue_gen:=0;
    outBlue_gen:=0;


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

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)));
    t9:=(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;
    t10:=(enStates_t10)&(enEvents_t10)&(enCond_t10);


  DEFINE
    any := t9|t10;
MODULE execute_blueAHts(en)
  --transition declaration
  VAR
    tran : {t9_exe, t10_exe, noTran_exe};

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

```
  --define INVAR constrait to define which transition to be executed
  INVAR
     (t9 -> en.t9)
    &(t10 -> en.t10)
    &(any -> en.any)

  --define the env sync event triggering macros for single 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;

  --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 single Hts blueAHts
  DEFINE
    inRed_trig:=0;
    outRed_trig:=0;
    inBlue_trig:=0;
    outBlue_trig:=0;

  --define the rend sync event generating macros for single Hts blueAHts
  DEFINE
    inRed_gen:=0;
    outRed_gen:=0;
    inBlue_gen:=t9;
    outBlue_gen:=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)));
    t11:=(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;
    t12:=(enStates_t12)&(enEvents_t12)&(enCond_t12);


  DEFINE
    any := t11|t12;
MODULE execute_blueBHts(en)
  --transition declaration
  VAR
    tran : {t11_exe, t12_exe, noTran_exe};

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

```
  --define INVAR constrait to define which transition to be executed
  INVAR
     (t11 -> en.t11)
    &(t12 -> en.t12)
    &(any -> en.any)

  --define the env sync event triggering macros for single 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;

  --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 single Hts blueBHts
  DEFINE
    inRed_trig:=0;
    outRed_trig:=0;
    inBlue_trig:=0;
    outBlue_trig:=0;

  --define the rend sync event generating macros for single Hts blueBHts
  DEFINE
    inRed_gen:=0;
    outRed_gen:=0;
    inBlue_gen:=t11;
    outBlue_gen:=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;
    t17:=(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;
    t18:=(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;
    t19:=(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;
```

```
    t20:=(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;
    t21:=(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;
    t22:=(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;
    t23:=(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;
    t24:=(enStates_t24)&(enEvents_t24)&(enCond_t24);


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

  --define INVAR constrait to define which transition to be executed
  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 single 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;

  --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 single 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 single Hts bridgeStatusHts
  DEFINE
    inRed_gen:=0;
    outRed_gen:=0;
    inBlue_gen:=0;
    outBlue_gen:=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;
    t5:=(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;
    t6:=(enStates_t6)&(enEvents_t6)&(enCond_t6);


  DEFINE
    any := t5|t6;
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);

  --define INVAR constrait to define which transition to be executed
  INVAR
     (t5 -> en.t5)
    &(t6 -> en.t6)
    &(any -> en.any)

  --define the env sync event triggering macros for single 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;


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

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;
    t7:=(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;
    t8:=(enStates_t8)&(enEvents_t8)&(enCond_t8);


  DEFINE
    any := t7|t8;
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);

  --define INVAR constrait to define which transition to be executed
  INVAR
     (t7 -> en.t7)
    &(t8 -> en.t8)
    &(any -> en.any)

  --define the env sync event triggering macros for single 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;


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

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;
    t13:=(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;
    t14:=(enStates_t14)&(enEvents_t14)&(enCond_t14);


  DEFINE
    any := t13|t14;
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);

  --define INVAR constrait to define which transition to be executed
  INVAR
     (t13 -> en.t13)
    &(t14 -> en.t14)
    &(any -> en.any)

  --define the env sync event triggering macros for single 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;


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

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;
    t15:=(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;
    t16:=(enStates_t16)&(enEvents_t16)&(enCond_t16);


  DEFINE
    any := t15|t16;
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);

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

  --define the env sync event triggering macros for single 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;


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

MODULE microEnvSync_entRedA_exitRedA_entRedB_exitRedB_entBlueA_exitBlueA_entBlueB_exitBlueB
              (enLeft,enRight,exeLeft,exeRight,enMe)
  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;
  INVAR
     (any -> enMe.any)
    &(! (entRedA_trig|exitRedA_trig|entRedB_trig|exitRedB_trig
         |entBlueA_trig|exitBlueA_trig|entBlueB_trig|exitBlueB_trig)
             -> !(exeLeft.any & exeRight.any))
    &(exeLeft.entRedA_trig <-> exeRight.entRedA_trig)
     &!(exeLeft.entRedA_trig
         & (exeRight.exitRedA_trig|exeRight.entRedB_trig|exeRight.exitRedB_trig|exeRight.entBlueA_trig
            |exeRight.exitBlueA_trig|exeRight.entBlueB_trig|exeRight.exitBlueB_trig))
    &(exeLeft.exitRedA_trig <-> exeRight.exitRedA_trig)
     &!(exeLeft.exitRedA_trig
         & (exeRight.entRedA_trig|exeRight.entRedB_trig|exeRight.exitRedB_trig|exeRight.entBlueA_trig
            |exeRight.exitBlueA_trig|exeRight.entBlueB_trig|exeRight.exitBlueB_trig))
    &(exeLeft.entRedB_trig <-> exeRight.entRedB_trig)
     &!(exeLeft.entRedB_trig
         & (exeRight.entRedA_trig|exeRight.exitRedA_trig|exeRight.exitRedB_trig|exeRight.entBlueA_trig
            |exeRight.exitBlueA_trig|exeRight.entBlueB_trig|exeRight.exitBlueB_trig))
    &(exeLeft.exitRedB_trig <-> exeRight.exitRedB_trig)
     &!(exeLeft.exitRedB_trig
         & (exeRight.entRedA_trig|exeRight.exitRedA_trig|exeRight.entRedB_trig|exeRight.entBlueA_trig
            |exeRight.exitBlueA_trig|exeRight.entBlueB_trig|exeRight.exitBlueB_trig))
    &(exeLeft.entBlueA_trig <-> exeRight.entBlueA_trig)
     &!(exeLeft.entBlueA_trig
         & (exeRight.entRedA_trig|exeRight.exitRedA_trig|exeRight.entRedB_trig|exeRight.exitRedB_trig
            |exeRight.exitBlueA_trig|exeRight.entBlueB_trig|exeRight.exitBlueB_trig))
    &(exeLeft.exitBlueA_trig <-> exeRight.exitBlueA_trig)
     &!(exeLeft.exitBlueA_trig
         & (exeRight.entRedA_trig|exeRight.exitRedA_trig|exeRight.entRedB_trig|exeRight.exitRedB_trig
            |exeRight.entBlueA_trig|exeRight.entBlueB_trig|exeRight.exitBlueB_trig))
    &(exeLeft.entBlueB_trig <-> exeRight.entBlueB_trig)
     &!(exeLeft.entBlueB_trig
         & (exeRight.entRedA_trig|exeRight.exitRedA_trig|exeRight.entRedB_trig|exeRight.exitRedB_trig
            |exeRight.entBlueA_trig|exeRight.exitBlueA_trig|exeRight.exitBlueB_trig))
    &(exeLeft.exitBlueB_trig <-> exeRight.exitBlueB_trig)
     &!(exeLeft.exitBlueB_trig
```

```
        & (exeRight.entRedA_trig|exeRight.exitRedA_trig|exeRight.entRedB_trig|exeRight.exitRedB_trig
           |exeRight.entBlueA_trig|exeRight.exitBlueA_trig|exeRight.entBlueB_trig))

MODULE microRendSync_inRed_outRed_inBlue_outBlue(enLeft,enRight,exeLeft,exeRight,enMe)
  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
     (any -> enMe.any)
    &(! (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 microIntl(enLeft,enRight,exeLeft,exeRight,enMe)
  DEFINE
    any:=exeLeft.any | exeRight.any ;
  INVAR
     (any -> enMe.any)
    &!(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_onRedA;
    in_waitRedA := redAHts_state=waitRedA;
    in_onRedA := redAHts_state=onRedA;
    in_redB := in_waitRedB | in_onRedB;
    in_waitRedB := redBHts_state=waitRedB;
    in_onRedB := redBHts_state=onRedB;
    in_blueCar := in_blueA | in_blueB;
    in_blueA := in_waitBlueA | in_onBlueA;
    in_waitBlueA := blueAHts_state=waitBlueA;
    in_onBlueA := blueAHts_state=onBlueA;
    in_blueB := in_waitBlueB | in_onBlueB;
```

```
    in_waitBlueB := blueBHts_state=waitBlueB;
    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.gen.entRedA;
DEFINE exitRedA := I.gen.exitRedA;
DEFINE entRedB := I.gen.entRedB;
DEFINE exitRedB := I.gen.exitRedB;
DEFINE entBlueA := I.gen.entBlueA;
DEFINE exitBlueA := I.gen.exitBlueA;
DEFINE entBlueB := I.gen.entBlueB;
DEFINE exitBlueB := I.gen.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);

  --extra macro for rendzevous event inRed
  DEFINE
    rend_events.inRed := redAHts.t1 | redBHts.t3 ;

  --extra macro for rendzevous event outRed
  DEFINE
```

```
    rend_events.outRed := redAHts.t2 | redBHts.t4 ;

  --extra macro for rendzevous event inBlue
  DEFINE
    rend_events.inBlue := blueAHts.t9 | blueBHts.t11 ;

  --extra macro for rendzevous event outBlue
  DEFINE
    rend_events.outBlue := blueAHts.t10 | blueBHts.t12 ;


  --define enabled macros for each composite HTSs
  DEFINE
    singleLaneBridgeHts.any := bridgeHts.any | coordHts.any;
    bridgeHts.any := carHts.any | bridgeStatusHts.any;
    carHts.any := redCarHts.any | blueCarHts.any;
    redCarHts.any := redAHts.any | redBHts.any;
    blueCarHts.any := blueAHts.any | blueBHts.any;
    coordHts.any := coordRedHts.any | coordBlueHts.any;
    coordRedHts.any := redCoordEntHts.any | redCoordExitHts.any;
    coordBlueHts.any := blueCoordEntHts.any | blueCoordExitHts.any;

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

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 :
      microEnvSync_entRedA_exitRedA_entRedB_exitRedB_entBlueA_exitBlueA_entBlueB_exitBlueB
                 (en.bridgeHts,en.coordHts,bridgeHts,coordHts,en.singleLaneBridgeHts);
  VAR
    bridgeHts :
      microRendSync_inRed_outRed_inBlue_outBlue
                 (en.carHts,en.bridgeStatusHts,carHts,bridgeStatusHts,en.bridgeHts);
  --defining macro flag for environmental sync events
  DEFINE
    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 : microIntl(en.redCarHts,en.blueCarHts,redCarHts,blueCarHts,en.carHts);
  --defining macro flag for environmental sync events
  DEFINE
    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.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 : microIntl(en.redAHts,en.redBHts,redAHts,redBHts,en.redCarHts);
--defining macro flag for environmental sync events
DEFINE
  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.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 : microIntl(en.blueAHts,en.blueBHts,blueAHts,blueBHts,en.blueCarHts);
--defining macro flag for environmental sync events
DEFINE
  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.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 : microIntl(en.coordRedHts,en.coordBlueHts,coordRedHts,coordBlueHts,en.coordHts);
--defining macro flag for environmental sync events
DEFINE
  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 :
      microIntl(en.redCoordEntHts,en.redCoordExitHts,redCoordEntHts,redCoordExitHts,en.coordRedHts);
  --defining macro flag for environmental sync events
  DEFINE
    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 :
      microIntl(en.blueCoordEntHts,en.blueCoordExitHts,
                blueCoordEntHts,blueCoordExitHts,en.coordBlueHts);
  --defining macro flag for environmental sync events
  DEFINE
    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 : onRedA;
      exe.redAHts.t2 : waitRedA;
      1 : iss.CS.redAHts_state;
    esac;

  ASSIGN
    next(pss.CS.redBHts_state):=case
      exe.redBHts.t3 : onRedB;
      exe.redBHts.t4 : waitRedB;
      1 : iss.CS.redBHts_state;
    esac;

  ASSIGN
    next(pss.CS.blueAHts_state):=case
      exe.blueAHts.t9 : onBlueA;
      exe.blueAHts.t10 : waitBlueA;
      1 : iss.CS.blueAHts_state;
    esac;

  ASSIGN
```

```
    next(pss.CS.blueBHts_state):=case
      exe.blueBHts.t11 : onBlueB;
      exe.blueBHts.t12 : waitBlueB;
      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 numRedt17 := case
  ((((iss.AV.numRed) + (1)))>=0)&((((iss.AV.numRed) + (1)))<=2) : ((iss.AV.numRed) + (1));
  1 : iss.AV.numRed;
 esac;
  DEFINE numRedt17error := case
  ((((iss.AV.numRed) + (1)))<0)|((((iss.AV.numRed) + (1)))>2) : 1;
  1 : 0;
 esac;
  DEFINE numRedt18 := case
  ((((iss.AV.numRed) + (1)))>=0)&((((iss.AV.numRed) + (1)))<=2) : ((iss.AV.numRed) + (1));
  1 : iss.AV.numRed;
 esac;
  DEFINE numRedt18error := case
  ((((iss.AV.numRed) + (1)))<0)|((((iss.AV.numRed) + (1)))>2) : 1;
  1 : 0;
 esac;
  DEFINE numRedt19 := case
  ((((iss.AV.numRed) - (1)))>=0)&((((iss.AV.numRed) - (1)))<=2) : ((iss.AV.numRed) - (1));
  1 : iss.AV.numRed;
```

```
esac;
 DEFINE numRedt19error := case
 ((((iss.AV.numRed) - (1)))<0)|((((iss.AV.numRed) - (1)))>2) : 1;
 1 : 0;
esac;
 DEFINE numRedt20 := case
 ((((iss.AV.numRed) - (1)))>=0)&((((iss.AV.numRed) - (1)))<=2) : ((iss.AV.numRed) - (1));
 1 : iss.AV.numRed;
esac;
 DEFINE numRedt20error := case
 ((((iss.AV.numRed) - (1)))<0)|((((iss.AV.numRed) - (1)))>2) : 1;
 1 : 0;
esac;
 ASSIGN
   next(pss.AV.numRed):=case
     exe.bridgeStatusHts.t17 : numRedt17;
     exe.bridgeStatusHts.t18 : numRedt18;
     exe.bridgeStatusHts.t19 : numRedt19;
     exe.bridgeStatusHts.t20 : numRedt20;
     1 : iss.AV.numRed;
   esac;

 --next state relation for  numBlue
 DEFINE numBluet21 := case
 ((((iss.AV.numBlue) + (1)))>=0)&((((iss.AV.numBlue) + (1)))<=2) : ((iss.AV.numBlue) + (1));
 1 : iss.AV.numBlue;
esac;
 DEFINE numBluet21error := case
 ((((iss.AV.numBlue) + (1)))<0)|((((iss.AV.numBlue) + (1)))>2) : 1;
 1 : 0;
esac;
 DEFINE numBluet22 := case
 ((((iss.AV.numBlue) + (1)))>=0)&((((iss.AV.numBlue) + (1)))<=2) : ((iss.AV.numBlue) + (1));
 1 : iss.AV.numBlue;
esac;
 DEFINE numBluet22error := case
 ((((iss.AV.numBlue) + (1)))<0)|((((iss.AV.numBlue) + (1)))>2) : 1;
 1 : 0;
esac;
 DEFINE numBluet23 := case
 ((((iss.AV.numBlue) - (1)))>=0)&((((iss.AV.numBlue) - (1)))<=2) : ((iss.AV.numBlue) - (1));
 1 : iss.AV.numBlue;
esac;
 DEFINE numBluet23error := case
 ((((iss.AV.numBlue) - (1)))<0)|((((iss.AV.numBlue) - (1)))>2) : 1;
 1 : 0;
esac;
 DEFINE numBluet24 := case
 ((((iss.AV.numBlue) - (1)))>=0)&((((iss.AV.numBlue) - (1)))<=2) : ((iss.AV.numBlue) - (1));
 1 : iss.AV.numBlue;
esac;
 DEFINE numBluet24error := case
 ((((iss.AV.numBlue) - (1)))<0)|((((iss.AV.numBlue) - (1)))>2) : 1;
 1 : 0;
esac;
 ASSIGN
   next(pss.AV.numBlue):=case
     exe.bridgeStatusHts.t21 : numBluet21;
     exe.bridgeStatusHts.t22 : numBluet22;
     exe.bridgeStatusHts.t23 : numBluet23;
     exe.bridgeStatusHts.t24 : numBluet24;
     1 : iss.AV.numBlue;
   esac;


 --next state relation for variable overflow and underflow
 ASSIGN
   next(pss.AV.error_variables):=
      (iss.AV.error_variables=1)
    | (exe.bridgeStatusHts.t17) & (numRedt17error=1)
```

```
      | (exe.bridgeStatusHts.t18) & (numRedt18error=1)
      | (exe.bridgeStatusHts.t19) & (numRedt19error=1)
      | (exe.bridgeStatusHts.t20) & (numRedt20error=1)
      | (exe.bridgeStatusHts.t21) & (numBluet21error=1)
      | (exe.bridgeStatusHts.t22) & (numBluet22error=1)
      | (exe.bridgeStatusHts.t23) & (numBluet23error=1)
      | (exe.bridgeStatusHts.t24) & (numBluet24error=1);

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);
```