Mapping Big-Step Modeling Languages to SMV

Fathiyeh Faghih and Nancy A. Day

Waterloo Formal Methods Laboratory (http://watform.uwaterloo.ca)

Cheriton School of Computer Science, University of Waterloo

Waterloo, ON, Canada, N2L 3G1

e-mail: {ffaghihe, nday}@cs.uwaterloo.ca

Technical Report CS-2011-29

December 23, 2011

Abstract

We propose an algorithm for creating a semantics-based, parameterized translator from the family of big-step modeling languages (BSMLs) to the input language of the model checker SMV. Our translator takes as input a specification in the CHTS notation and a set of user-provided parameters that encode the specification's semantics; it produces an SMV model suitable for model checking. We use a modular approach for translation, which means that the structure of the resulting SMV model matches the source CHTS structure.

1 Introduction

With the popularity of model-driven methodologies, many different behavioral modeling languages have been introduced. One of the major challenges arises in verifying properties of models designed using different languages. The issue can be addressed by "model transformations for

verification". By model transformation, we refer to a translation from a design model notation to the input language of a model checker such as SMV (Symbolic Model Verifier) [1] or SPIN (Simple Promela Interpreter) [2]. We propose an algorithm for creating a semantics-based, parameterized translator from a set of modeling languages, called big-step modeling languages (BSMLs), to the input language of the model checker SMV. Our translator takes as input a model specification in a standard notation and a set of parameters that encode the specification's semantics. It produces an SMV model with the same behaviour, which can be used for model checking of the specification. The semantics of the models are based on the semantic aspects and options introduced in Esmaeilsabzali *et al.* [3]. This work can be compared to writing a direct translator from one notation to another, or to an intermediate language (e.g., SAL [4] and Action Language [5]). The main difference between our research and this work is that using our translator, different combinations of options for semantic aspects lead to generating new translators for specific languages, so different translators for different languages do not need to be created.

The most similar work to our research is Lu *et al.*'s work on mapping template semantics to SMV [6]. There are two main differences between our translator and the translator in Lu *et al.* [6]. First of all, the parameters of the deconstruction in Esmaeilsabzali *et al.* [3] correspond to higher level semantic concerns compared to the parameters in Niu *et al.* [7], while having fewer dependencies among them. Second, the modular structure in the resulting SMV model matches the source CHTS structure in our work, while it matches the modularity of template semantics in Lu *et al.* [6]. Our contribution in changing the modularity of the resulting SMV model can be useful when the model user wants to break apart the original model, and take it to another specification, or check it for additional properties; she does not need to have the translator and run it again. This modularity can also improve the readability of the resulting SMV model, in case the model user needs to modify it and use it for other purposes.

We seek to reduce the complexity of the resulting SMV model by doing computation in the translator as much as possible as opposed to in the SMV model during its execution. For example, we find the inconsistent transitions based on the Small-Step Consistency and Preemption aspects in the translator and the corresponding invariants are added to the SMV model, so that the

resulting SMV model is much simpler.

The other ideas of our translation (which are similar to Lu *et al.*'s work in [6]) are, first, to use macros extensively, which can be used to structure the SMV model, as well as not adding to the state space (compared to the definition of new SMV variables). Second, we use invariants to remove paths of model execution not allowed by the semantic options, which helps to preserve all non-determinism. By preserving non-determinism, we mean whenever a transition (or a set of transitions) can be executed in the model, it can be executed in the resulting SMV model as well.

The rest of this report is structured as follows: In Section 2, we have a brief discussion on background including big-step modeling languages in terms of their syntax and semantics, and the SMV language. Section 3 is the overview of our translation algorithm. Section 4 describes the general structure of the SMV model. In Section 5, we present our algorithm for translating the structural semantic aspects. Section 6 is a discussion of the translation of the dynamic semantic aspects. Section 7 is the algorithm for extracting the SMV submodel corresponding to a submodel of the original CHTS model. Section 8 discusses related work, and finally, Section 9 concludes the paper.

2 Background

In this section, we present a brief discussion of BSMLs and the SMV language.

2.1 Big-Step Modeling Languages (BSMLs)

Big-Step Modeling Languages (BSMLs) are a popular class of behavioural modeling languages. In this group of languages, the modeler can specify the reaction of the system to an environmental input as something called a big-step. A big-step consists of a sequence of small-steps, each of which may contain one transition or a set of concurrent ones. There are many BSMLs with different syntaxes and semantics, such as statecharts variants [8], Argos [9], Reactive Modules [10], and SCR [11].

2.1.1 Normal Form Syntax of BSMLs

Some BSMLs use graphical syntaxes, such as statecharts, and others use textual syntaxes such as SCR. To study this class of languages, we have to use a syntactic "normal form" that is sufficiently expressive to represent the syntax of other notations. We use the normal-form syntax introduced in Esmaeilsabzali *et al.* [12], which is called the composed hierarchical transition system (CHTS) syntax. The CHTS syntax consists of two components; (i) control states, and (ii) transitions between the control states.

A control state is a named artifact that a modeler uses to represent a noteworthy moment in the execution of a model. There are three types of control states: Basic state, Orstate, or Andstate. A model in CHTS syntax is represented in a hierarchical structure. In the hierarchical tree, the leaf nodes are Basic control states, and the next levels contain Andstates and Orstates.

For an Or or an And control state, S, $children^*(S)$ is the set of all control states that are children of S, either directly or by transitivity. The *parent*, *ancestor*, and *descendant* relations are used with their usual meanings in CHTS. An Orstate S has a *default control state*, default(S), which is one of its children. A transition t has a *source* control state, and a *destination* control state. The *least common ancestor* of two control states s and s' is the lowest control state (closest to the leaves of the hierarchy tree) in the hierarchy tree such that: $s, s' \in children^*(lca(s, s'))$.

2.1.2 Prescriptive Semantics for Big-Step Modeling Languages

Different BSMLs use different semantics. In Esmaeilsabzali *et al.* [12], the operational semantics of a large class of BSMLs is deconstructed into eight high-level, mostly orthogonal, semantic aspects. Moreover, it presents an enumeration of the common semantic options found in existing BSMLs for each of these aspects. These options cover a wide range of existing BSMLs.

The semantic aspects are categorized into two groups : (i) structural, and (ii) dynamic. A structural semantic aspect deals with the structure of the hierarchy tree of a model, as opposed to a dynamic parameter. Structural semantic aspects include Concurrency, Small-Step Consistency, Preemption, and Priority, and the dynamic aspects are Big-Step Maximality, Event Lifeline, Enabledness Memory Protocol, Assignment Memory Protocol, Order of Small-Steps, and Combo-Step Maximality. We will explain the semantic aspects and their options later in

this report.

2.1.3 Semantic Definition Schema

The semantic definition schema, introduced in Esmaeilsabzali *et al.* [12] is a set of parametric definitions in standard logic and set theory. This schema defines the way in which a model is executed based on structural and dynamic semantic options. Based on this schema, in each small-step, the enabled transitions that can be executed are specified. It starts from an empty *execution set* at the bottom of the hierarchy of control states, and as it goes up the hierarchy, it modifies the execution set by considering higher-level transitions. The resulting set at each level will be a set of sets of transitions such that the transitions in each set can be executed in a small-step together. The *merge* operator ($\otimes$) formally specifies how the execution set should be modified at each level considering the structural semantic options:

$$\mathbb{T} \otimes T' = \{T_1 - T_1' \cup T'' | T_1 \in \mathbb{T} \wedge T_1' \subseteq T_1 \wedge T'' \subseteq T' \wedge$$

$$(\forall t' : (T_1 \cup T').t' \in (T' - T") \Leftrightarrow \exists t \in (T_1 - T_1' \cup T'').\neg C(t', t) \wedge \neg P(t', t)) \wedge \quad (1)$$

$$(\forall t : (T_1 \cup T').t \in T_1' \Leftrightarrow \exists t' \in T''.\neg C(t, t') \wedge \neg P(t, t')) \wedge \Pi(\mathbb{T}, T', T_1, T_1', T'')\} \quad (2)$$

C, P, and Π are parameters for Small-Step Consistency, Preemption, and Priority respectively. At each level of the hierarchy tree of a model, the merge operator decides how to choose from the set of sets of enabled transitions at the lower level of the tree (parameter $\mathbb{T}$), and the transitions at the current level of the tree (parameter T'). Parameter $\mathbb{T}$ is in a special font because its type is set of sets of transitions, as opposed to a set of transitions such as T'. Based on this definition, a transition (t') from the current level is not chosen, if there exists a transition (t) in the set that is not consistent with t' (line (1) in the equation). Moreover, a transition from the lower levels (t) gets removed from the set, if there exists a transition in the current level (t') that is not consistent with t, and also the priority of t is not higher than the priority of t' (line (2) in the equation). The result of a merge operation is a new set of sets of transitions. The final result is a set of sets of transitions, from which a set is selected non-deterministically for execution.

2.2 SMV Language

SMV is a symbolic model checker that allows us to formally verify temporal logic properties of finite state systems, such as software and hardware designs. Therefore, instead of writing a simulation test bench, we can verify the designed models for all possible input sequences. It allows us to verify the correctness of our specifications very early in the design process by building abstract system level models.

In the SMV language, models are described using variable declarations, assignments to variables (initial and next values to variables in every SMV step), and properties that we want to check. Variables can be Boolean, enumerated types, integer subranges, or an array of any of these. Models may also contain the specification of invariants, as Boolean expressions following the keyword `INVAR`. The other constructs that we use extensively in our translation are macros. Macros are declared after the keyword `DEFINE`. They are replaced by their definitions, so they have the advantage of not increasing the system's state space. Expression operators !, &, |, $\rightarrow$, and $\leftrightarrow$, represent "not", "and", "or", "implies", and "iff" respectively. Comments follow the symbol "- -".

A model in SMV may consist of several modules. Using modules, the statements can be reused by creating a module instance, which is declared as a variable. The declared variable can be passed as a parameter to another module. If `a` identifies an instance of a module, then the expression `a.b` identifies the internal variable or macro named `b` within the module instance `a`. All statements and assignments in all modules run synchronously in an SMV step.

3 Overview of Translation Algorithm

In this section, we will go through our algorithm that translates a CHTS model to an SMV model with the same behavior. The details of the algorithm will be explained in the following sections of this report.

The general steps of our translation algorithm are:

1. Check to see if the model is well-formed, based on the well-formedness rules of CHTS models. (Section 5.4)

```
If (the model is not well-formed)
  Return;
Create the general structure of the SMV model (Code in Fig. 3);
If (Concurrency == Single){
  Code in Fig. 4;
}
else{
    If (Preemption == Preemptive){
      Code in Fig. 14 and 15;
    }
    else{
      Code in Fig. 20 and 21;
    }
    If the Priority option is not No Priority
      Code in Fig. 24;
}
```

Figure 1: General steps of the translation algorithm

2. Create the general structure of the SMV model. (Section 4)

3. According to the Small-Step Consistency and Preemption options, generate the appropriate invariants and put them in the corresponding modules. If the Priority option is not No Priority, generate the required macros (if needed) and invariants. (Section 5)

4. According to the dynamic options, insert the appropriate next statements. (Section 6)

The pseudo code of the algorithm is illustrated in Fig. 1. The details of the algorithm will be described later in this report.

4 Structure of the SMV Model

In this section, we will go through the general structure of the SMV model resulting from our translation method. In the following, we explain the generated modules with the required macros and variables and give an example of the SMV model for the CHTS model in Fig. 2.

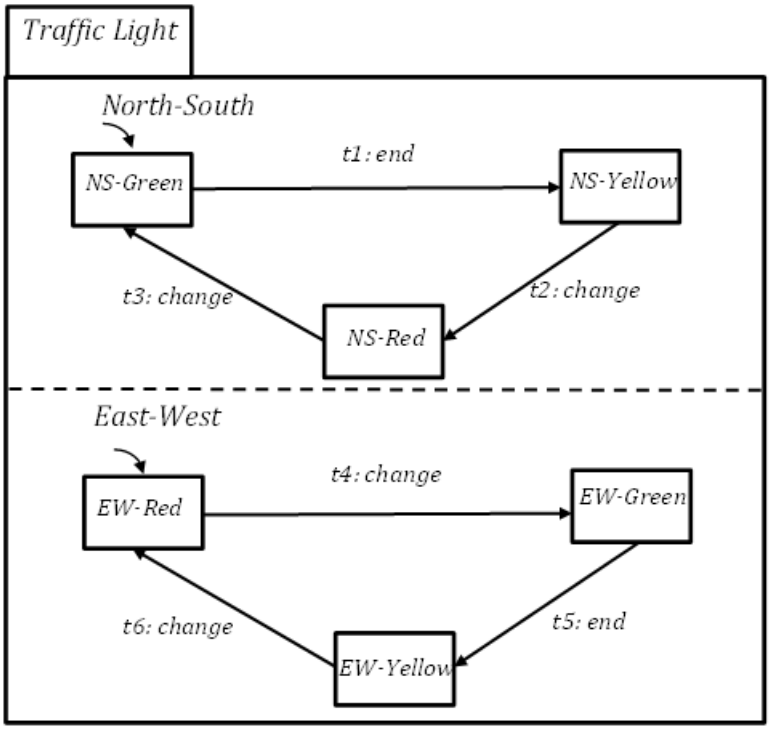


Figure 2: Traffic light system (adopted from [13])

- **Input Module:** The input module contains a set of variables corresponding to the different possible inputs from the environment.
- **Snapshot Module:** The snapshot module represents a specification's snapshot and contains the information about the CHTS that is needed by all modules. In this module, a Boolean variable is defined for each basic state representing whether it is active in a snapshot or not. A macro is also defined for every non-basic state that indicates whether that state is a current state in the snapshot. An Andstate or Orstate is current if one of its children states is current, therefore, the macro of a non-basic state is a disjunction of variables

or macros corresponding to its direct children. Thus, the specification's current state is represented using variables only for the basic states. A Boolean variable and a variable with the appropriate type are defined for each event and each variable in the original model respectively. The events and variables names are the same as their names in the original CHTS model. The snapshot module also includes an instance of the input module. Whenever the model is going to reset, the next values of events and variables are read from the input module.

There is also a Boolean variable defined for each transition in the model, which is called the "execution variable". The execution variable of the transition t1 has the name t1_execute. The next value of control states, variables, and events are assigned using the execution variables of the corresponding transitions (Section 6). As an example, for the CHTS model in Fig. 2, the snapshot module is:

```
MODULE snapshot(){
VAR
  --an instance of the input module
  input : input;
  --basic states
  NS_Green,NS_Yellow,NS_Red,EW_Green,EW_Yellow,EW_Red : boolean;
  --events
  end,change : boolean;
  -execution variables of transitions
  t1_execute,t2_execute,t3_execute,t4_execute,t5_execute,t6_execute : boolean;
DEFINE
  --non-basic states
  North-South := NS_Green | NS_Yellow | NS_Red;
  East-West := EW_Green | EW_Yellow | EW_Red;
 --next statements of control states, events, and variables
 ...
}
```

- **Main Module:** The main module contains an instance of the snapshot module, and an

instance of the root module with the snapshot module instance as an argument. The main module for the CHTS model in Fig. 2 is:

```
MODULE main(){
--instance of the module snapshot
ss : snapshot;
--instance of the root module
model_root : Traffic_Light(ss);
--properties to be checked
...
}
```

- **State Modules:** One SMV module is generated for each non-basic state. We call them "state modules" in this report. All state modules receive the instance of the snapshot module as an argument, with the name ss.

  Each state module has a flag for each transition whose scope is that state; a macro whose value determines whether the transition is enabled with respect to its conditions (events and variables) and source control state or not. The macro for the transition t1 has the name t1_enabled.

  For each direct child of the state, an instance of the corresponding state module is defined in the parent state module. It has the same name as the state name in the CHTS model. Each state module also contains one flag (macro) for enabledness and execution. The enabledness flag shows whether there is any enabled transition in that state or in any of its descendants. The execution flag shows whether any of the transitions in the state or its descendants are going to be taken in the next small-step or not. The enabledness/execution macro is a disjunction of the enabledness/execution macros of the transitions (in its scope) and the enabledness/execution flags of its non-basic direct children (Note that the execution variables of transitions are defined in the snapshot module). The execution variable and enabledness macro of the state S1 have the names S1_execute and S1_enabled respectively.

  For example, the state modules for the CHTS in Fig. 2 is:

```
MODULE North_South(ss){
  t1_enabled := ss.NS_green & ss.end;
  t2_enabled := ss.NS_yellow & ss.change;
  t3_enabled := ss.NS_red & ss.change;
  enabled := t1_enabled | t2_enabled | t3_enabled;
  execute := ss.t1_execute | ss.t2_execute | ss.t3_execute;
  --invariants
  ...
}
MODULE East_West(ss){
  t4_enabled := ss.EW_red & ss.change;
  t5_enabled := ss.EW_green & ss.end;
  t6_enabled := ss.EW_yellow & ss.change;
  enabled := t4_enabled | t5_enabled | t6_enabled;
  execute := ss.t4_execute | ss.t5_execute | ss.t6_execute;
  --invariants
  ...
}
MODULE Traffic_Light(ss){
  North_South : North_South(ss);
  East_West : East_West(ss);
  enabled := North_South.enabled | East_West.enabled;
  execute := North_South.execute | East_West.execute;
  --invariants
  ...
}
```

For priority, we may need priority flags as well (Section 5.5). There are a set of invariants in each module that will be explained later. The pseudo code for generating the general structure of the SMV model is illustrated in Fig. 3.

```
1- Create the Input module with one variable for each input variable in the CHTS;
2- Create the Snapshot module with one Boolean variable for each basic state
 and one macro for each non-basic state
 (which is disjunction of the variables and macros corresponding to its children)
 and a Boolean variable and a variable with the appropriate type
 for each event and each variable in the CHTS
 and a Boolean variable for each transition;
3- Create the Main module with an instance of the Snapshot module and
 an instance of the root module with the Snapshot module instance as an argument;
4- Create an SMV module for each non-basic state
 which receives the instance of the snapshot module as an argument
 and has a macro for each transition whose scope is that state
 and an instance of each state module corresponding to each of its children
 and two macros for enabledness and execution of the state;
```

Figure 3: Pseudo code for generating the general structure of the SMV model

5 Structural Parameters

In this section, we discuss different options for structural parameters and how they affect the resulting SMV model. BSMLs vary in how the enabled transitions of a model execute together in a small step. Structural semantic parameters deal with this property of BSMLs. There are four structural semantic aspects; Concurrency, Small-step consistency, Preemption, and Priority. In this section, we discuss the different options for these aspects, and specify the inconsistent pairs of transitions for their different options. Table 1 lists the options for Concurrency, Small-Step Consistency, and Preemption, and their definitions.

5.1 Concurrency

There are two options for Concurrency; Single and Many. In this section, we discuss the inconsistent pairs of transitions for each option.

5.1.1 Single

When the Concurrency option is Single, Small-Step Consistency and Preemption are not relevant, since only one transition can be executed in each small-step. Therefore, we first discuss the resulting model when the Concurrency option is Single.

With the Single option, only one of the enabled transitions can be taken in a small-step. Our translator generates invariants in all modules that disallow the concurrent execution of transitions. In each module, we add an invariant for every pair of transition-transition, transition-child, and child-child, so that they cannot be true at the same time. For the transitions, execution variables and for the children, the execution flags of the corresponding modules are considered. The pseudo-code for inserting the invariants, when the Concurrency option is Single, is illustrated in Fig. 4.

For example, consider a module with two children (A and B) and one transition (t1). The module includes instances of the modules A and B. It also has an instance of the Snapshot module (ss) as an argument. Modules A and B include a Boolean variable corresponding to the execution of at least one transition in their scopes, and the module Snapshot contains one execution variable for each transition. We add the following invariants to this module:

Options	Definition
Concurrency	
SINGLE	A small-step consists of the execution of exactly one transition.
MANY	A small-step may consist of the execution of more than one transition.
Small-Step Consistency	
ARENA ORTHOGONAL	The arenas of two distinct transitions of a small-step are orthogonal.
SOURCE/ DESTINATION ORTHOGONAL	The source control states and destination control states of two distinct transitions of a small-step are pairwise orthogonal.
Preemption	
PREEMPTIVE	Two transitions that one is an "interrupt for" another cannot be taken in a small-step.
NON-PREEMPTIVE	Two transitions that one is an "interrupt for" another can be taken in a small-step.

Table 1: Structural semantic aspects

```
~(ss.t1_execute & A.execute);
~(ss.t1_execute & B.execute);
~(A.execute & B.execute);
```

```
For each state module{
  For each pair of transitions t1 and t2
    Add the invariant ~(ss.t1_execute & ss.t2_execute);
  For each pair of transition t1 and direct child A
    Add the invariant ~(ss.t1_execute & A.execute);
  For each pair of direct children A and B
    Add the invariant ~(A.execute & B.execute);
}
```

Figure 4: Pseudo-code of the invariant generation, when the Concurrency option is Single

5.1.2 MANY

When the concurrency option is Many, more than one transition can be executed in a small-step. But the question is can all transitions be taken with each other? The execution of transitions is controlled by the other two structural aspects; Small-Step Consistency and Preemption.

Based on Esmaeilsabzali's semantic schema [12], two transitions can be taken together, if they are consistent based on Small-Step Consistency, or they are consistent based on Preemption. In other words, two transitions cannot be taken together, if they are not consistent based on Small-Step Consistency, and they are not consistent based on Preemption either. Our goal is to find transitions that cannot be taken together in the translator and insert the appropriate invariants into the resulting SMV model, so they cannot be executed together. Using invariants, we preserve non-determinism for all allowed combinations of executing transitions.

In this report, we first go through the inconsistent pairs of transitions based on different Small-Step Consistency options, and then, for each option, find the inconsistent pairs (among the result set) based on the Preemption option. Practically, in the translator, we can have four cases (for each possible pair of options for Small-Step Consistency and Preemption), and in each case, find the inconsistent pairs of transitions accordingly.

5.2 Small-Step Consistency

There are two options for the Small-Step Consistency semantic aspect; Arena Orthogonal, and Source-Destination Orthogonal. In this section, we will go through these options separately, and recognize the inconsistent pairs of transitions based on them. We do not provide invariants in this section, since we should consider both Small-Step Consistency and Preemption options to find the inconsistent pairs of transitions. In Section 6.3, the inconsistent pairs recognized in this section are used to find the inconsistent transitions (for each pair of Small-Step Consistency and Preemption options) and recognize the required invariants.

Before going into details, we distinguish three categories of transitions:

1. Simple transition: A transition whose scope is an Orstate and its source and destination are direct children of its scope.

2. Orstate cross transition: A transition whose scope is an Orstate and its source and/or destination are not direct children of its scope.

3. Andstate cross transition: A transition whose scope is an Andstate, therefore it crosses a dashed line.

For Orstate/Andstate cross transition, the "source part" is its source state or the ancestor of its source state that is *a direct child of its scope*. Similarly, "destination part" is defined as its destination state or the ancestor of its destination state that is *a direct child of its scope*.

5.2.1 Arena Orthogonal

In the Arena Orthogonal option, two transitions are consistent if their arenas are orthogonal. In the following, we discuss inconsistent transitions for Arena Orthogonal in Orstates and Andstates separately.

Orstate

In the Orstate in Fig. 5, t1 and t2 are representatives of simple transitions and t3 and t4 are representatives of Orstate cross transitions.

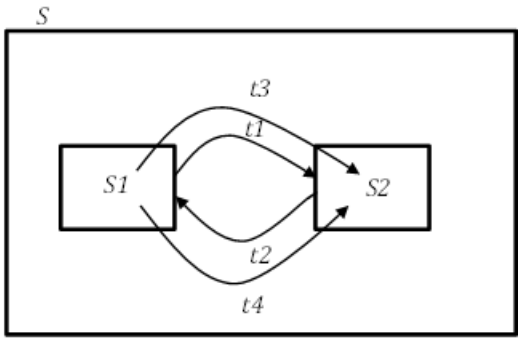


Figure 5: An Orstate with children and representatives of possible transitions

Transitions in S1 and S2 cannot be enabled with each other, since they are children of an Orstate and only one of them can be active at each snapshot. Therefore, no invariant is needed for them. This is also the case for t1/t3/t4 and transitions in S2, and also t1/t3/t4 and t2. The only pairs that can be enabled at the same time are S1 and t1/t3/t4, S2 and t2, t1 and t3/t4, and t3 and t4. Based on Arena Orthogonal, S1 and t1/t3/t4 are inconsistent, since their arenas cannot be orthogonal (S2 and t2 has a similar condition to S1 and t1). It is also the case for t1 and t3/t4 with a similar reason. t3's and t4's arenas are both S, and therefore, their arenas cannot be orthogonal either.

Andstate

When the Small-Step Consistency option is Arena Orthogonal, transitions in the direct children of an Andstate can be taken together as long as their arenas are descendants of the Andstate. The only case where this is not the case is when the Andstate has an Andstate as a child and it has an Andstate cross transition. For example, consider the CHTS model in Fig. 6.

In this example, S2 is an Andstate, and the transitions shown in the model cannot be taken with transitions in S1. That is because the arena of this transition is an ancestor or the same as the arena of transitions in S1. There is no Orstate in S2 that is the scope or the scope's ancestor of these transitions. So, the Andstate cross transitions' arenas cannot be orthogonal with the arenas of transitions in S1, and therefore, they are not consistent based on Arena Orthogonal.

We should also consider the Andstate cross transition, and investigate its consistency with

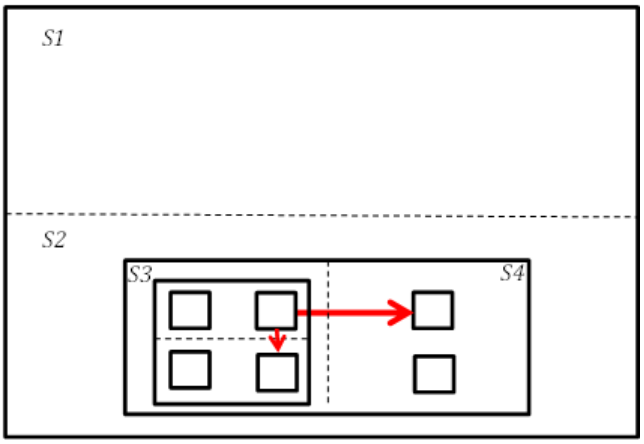


Figure 6: An example of transitions in a child of an Andstate that cannot be taken with transitions in the other child

the Andstate children and other cross transitions. Cross transitions of an Andstate (such as t1 and t2 in Fig. 7) are inconsistent with each other and with all transitions in the Andstate's children based on Arena Orthogonal. For example, the arena of t1 is an ancestor of S, while the arena of a transition in S1 is a descendant of S1, S1, or an ancestor of S1. t1's arena cannot be orthogonal with the arena of the transition in S1. The arenas of t1 and t2 are the same, therefore, they cannot be orthogonal with each other either. So, t1 and t2 are also inconsistent based on Arena-Orthogonal option.

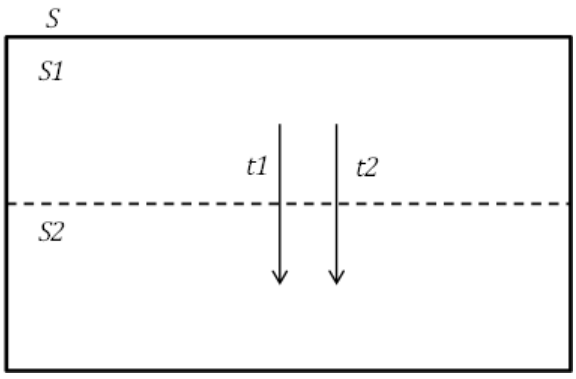


Figure 7: An Andstate with two Andstate cross transitions

As a summary, the inconsistent pairs of transitions in a CHTS model with the Arena Orthogonal

option are as follows:

- Simple transitions originating from the same state
- Simple transition and transitions in the simple transition's source state
- Orstate cross transitions with the same scope
- Orstate cross transition and transitions in the cross transition's source part (such as t3 with transitions in S1 in Fig. 5)
- Andstate cross transitions with the same scope
- Andstate cross transitions in an Andstate (A) that is a child of another Andstate (B) and transitions in the other child of the Andstate B.

5.2.2 Source-Destination Orthogonal

In the Source-Destination Orthogonal option, two transitions are consistent if their source states and destination states are pairwise orthogonal. In the following, we discuss inconsistent transitions for Source-Destination Orthogonal in Orstates and Andstates separately.

Orstate

In the Orstate in Fig. 8, transitions in S1 and S2 cannot be enabled with each other, since they are children of an Orstate and only one of them can be active at each snapshot. Therefore, no invariant is needed for them. This is also the case for t1/t3/t4 and transition in S2, and also t1/t3/t4 and t2. Transitions in S1 and S2 cannot be enabled with each other, since they are children of an Orstate and only one of them can be active at each snapshot. Therefore, no invariant is needed for them and we do not consider them in our algorithm. The only pairs that can be enabled at the same time are S1 and t1/t3/t4, t1 and t3/t4, and t3 and t4 (t2 and transitions in S2 can also be enabled at the same time, but we just consider S1 as a typical child of an Orstate).

Based on Source-Destination Orthogonal, S1 and t1/t3/t4 are inconsistent, since their destinations cannot be orthogonal. t1 is also inconsistent with t3/t4, since their destinations cannot be

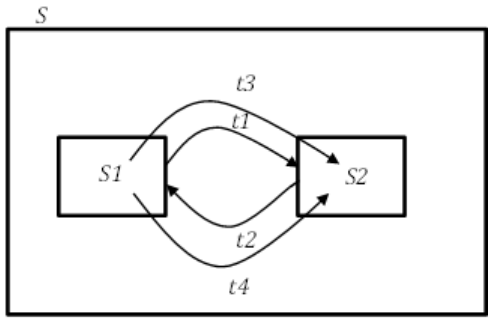


Figure 8: An Orstate with children and different possible transitions

orthogonal. t3 and t4 may be consistent with each other based on Source-destination orthogonal, and that happens when their sources (descendants of S1) are orthogonal, and their destinations (descendants of S2) are also orthogonal.

Andstate

When the Small-Step Consistency is Source-Destination Orthogonal, the transitions in children of an Andstate can be taken with each other. That's because their sources are orthogonal, and their destinations are orthogonal as well. But we should consider the cross transitions of Andstates as well.

The cross transition of an Andstate (like t1 or t2 in Fig. 7) can be taken with the transitions in the source part (S1), if their sources are orthogonal (since their destinations are orthogonal). They can be taken with the transitions in the destination part (S2), if their destinations are orthogonal (since their sources are orthogonal). t1 and t2 may also be consistent with each other, if their sources and destinations are pairwise orthogonal.

As a summary, the inconsistent pairs of transitions in a CHTS model based on Source-Destination Orthogonal option are as follows:

- Simple transitions originating from the same state
- Simple transition and transitions with the scope of the simple transition's source state

- Orstate cross transitions with the same scope, if their source and destination states are not pairwise orthogonal
- Orstate cross transition and transitions with the scope of the cross transition's source part
- Andstate cross transition and those transitions in the cross transition's source part that their source states are not orthogonal
- Andstate cross transition and those transitions in the cross transition's destination part that their destination states are not orthogonal
- Andstate cross transitions if their source and destination states are not pairwise orthogonal.

5.3 Preemption

There are two options for the Preemption semantic aspect; Non-Preemptive, and Preemptive. In this section, we go through the options for Preemption, and for each option consider the inconsistent transitions for each Small-Step Consistency option. Among these pairs, we find the inconsistent ones based on the Preemption option. Inconsistent transitions based on both Small-Step Consistency and Preemption options are not allowed to be taken together. Therefore, we need to find them to generate appropriate invariants to disallow their concurrent execution.

We aim to match the modular structure of the SMV model with the modularity of the original CHTS model. One of our goals in modularity preserving is to use the execution variable of each transition (defined in the snapshot module) only in its home module.

Definition 1. *The home module of a transition is where its enabledness macro is defined and is (usually) its definition scope in the CHTS model.*

In this section, we recognize the model structures where adding the required invariants will violate this modularity property, and resolve this issue (except for three non-conformances) by changing the home modules of transitions. The home module of a transition cannot be higher than its scope in the state hierarchy; otherwise, the enabledness macro of the transition might be lost when extracting a part of the SMV model corresponding to a CHTS submodel.

5.3.1 Preemptive

Based on the Preemptive option, two transitions where one is an "interrupt for" another cannot be taken in a small-step. We consider this option together with the two possible options of Small-Step Consistency to find inconsistent pairs of transitions.

5.3.1.1 Arena Orthogonal In this section, we discuss the inconsistent transitions based on Preemptive and Arena Orthogonal options.

Orstate

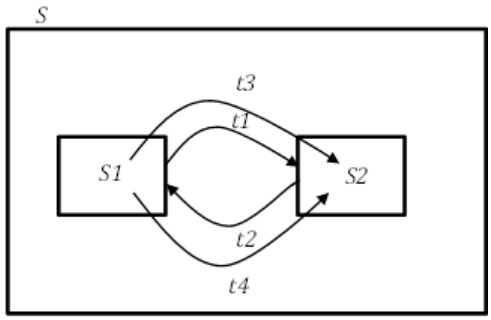


Figure 9: An Orstate with children and representatives of possible transitions

Inconsistent transitions based on the Arena Orthogonal option in an Orstate are:

- Simple transitions originating from the same state
- Simple transition and transitions in the simple transition's source state
- Orstate cross transitions with the same scope
- Orstate cross transition and transitions in the Orstate cross transition's source part
- Simple transitions and Orstate cross transitions with the same scope (that can be enabled with each other)

None of these transitions are consistent based on Preemptive option. Thus, they are all pairs of inconsistent transitions. Therefore, for the Arena Orthogonal and Preemptive options, we add the appropriate invariants to the Orstate module. For example, for the Orstate in Fig. 9, the following invariants are required:

```
~(ss.t1_execute & S1.execute);
~(ss.t2_execute & S2.execute);
~(ss.t3_execute & S1.execute);
~(ss.t4_execute & S1.execute);
~(ss.t1_execute & ss.t3_execute);
~(ss.t1_execute & ss.t4_execute);
~(ss.t3_execute & ss.t4_execute);
```

Andstate

Based on Arena Orthogonal, Andstate cross transitions are inconsistent with each other and with transitions in the children of the Andstate. They are not consistent based on Preemptive option either. Therefore, appropriate invariants are needed to prevent their concurrent execution.

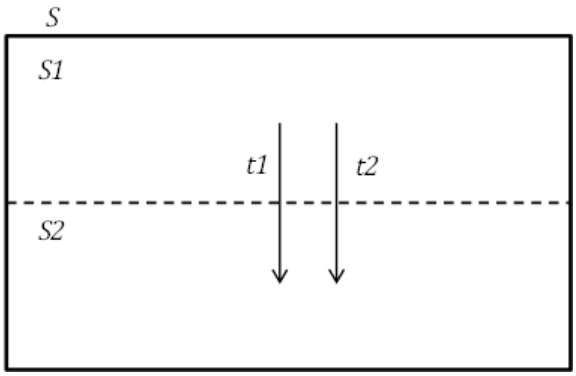


Figure 10: An Andstate with two cross transitions

For example, for the Andstate "S" in Fig. 10, we add the following invariants to the module "S":

```
~(ss.t1_execute & S1.execute);
~(ss.t1_execute & S2.execute);
```

```
~(ss.t2_execute & S1.execute);
~(ss.t2_execute & S2.execute);
~(ss.t1_execute & ss.t2_execute);
```

The transitions inside different Andstate children are consistent with each other, except for the case that the Andstate child is an Andstate and it has an Andstate cross transition. We call this kind of transitions, "unusual transitions". In the Andstate in Fig. 11, S2 is an Andstate with two children, S3 and S4, and t1 is an unusual transition.

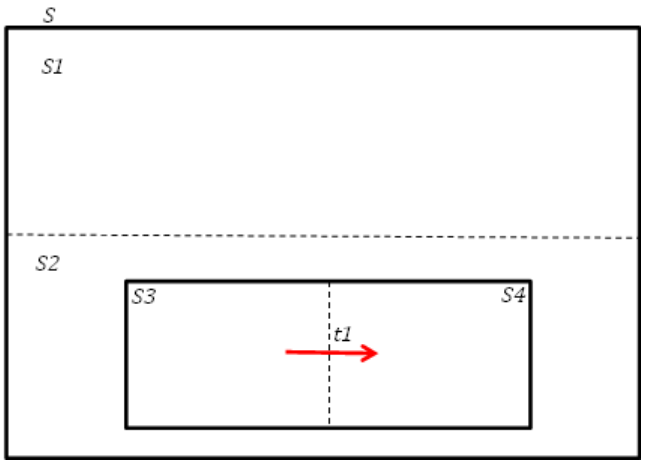


Figure 11: An Andstate with an unusual transition

t1 is inconsistent with all transitions in S. They are also inconsistent based on the Preemptive option, and therefore, we must take care of it by adding appropriate invariants. But the problem here is that t1 is not local to S (its home module is S2), and so we cannot add the invariant without violating the modularity.

Considering such a structure in a model, we can figure out that this is a bad modeling configuration. The reason behind our claim is that transitions inside children of an Andstate must be able to be taken with each other, and the unusual transition violates the nature of Andstates and thus, the model might not behave as the modeler expects. Therefore, our translator disallows unusual transitions when the consistency option is Arena Orthogonal (Regardless of the option for Preemption, unusual transitions lead to this weird behavior).

5.3.1.2 Source-Destination Orthogonal In this section, we discuss the inconsistent transitions based on Preemptive and Source-Destination Orthogonal options.

Orstate

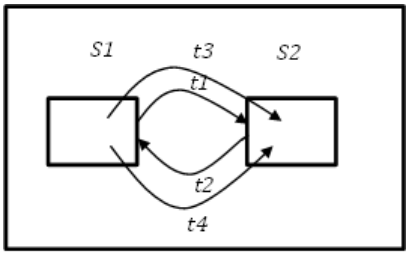


Figure 12: An Orstate with children and representatives of possible transitions

Inconsistent transitions based on Source-Destination Orthogonal option in an Orstate are:

- Simple transitions originating from the same state
- Simple transition and transitions in the simple transition's source state
- Orstate cross transitions with the same scope, if their source and destination states are not pairwise orthogonal
- Orstate cross transition and transitions in the cross transition's source part.
- Simple transitions and Orstate cross transitions with the same scope (that can be enabled with each other)

None of these transitions are consistent based on Preemptive option, and therefore, they are all pairs of inconsistent transitions. Therefore, we add the appropriate invariants to the SMV model for the Source-Destination Orthogonal and Preemptive options. For example, for the Orstate in Fig. 12, the following invariants are needed:

```
~(ss.t1_execute & S1.execute);
~(ss.t2_execute & S2.execute);
~(ss.t3_execute & S1.execute);
```

```
~(ss.t4_execute & S1.execute);
~(ss.t1_execute & ss.t3_execute);
~(ss.t1_execute & ss.t4_execute);
```

And:

```
~(t3_execute & t4_execute);
```

For t3 and t4, the translator checks to see if they are consistent based on Source-Destination Orthogonal option or not; if they are consistent based on either option, no invariant is needed for them.

Andstate

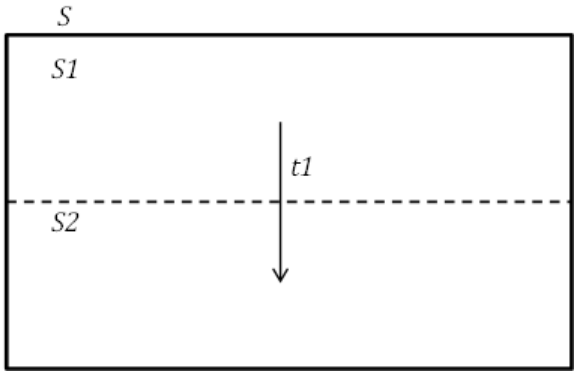


Figure 13: An Andstate with a cross transition

With Source-Destination Orthogonal option, an Andstate cross transition cannot be taken with the transitions in its source part (like t1 and transitions in S1 in Fig. 13), if their sources are not orthogonal (their destinations are orthogonal). It cannot be taken with the transitions in its destination part (like t1 and transitions in S2 in Fig. 13), if their destinations are not orthogonal (their sources are orthogonal). None of these transitions are consistent based on Preemptive option. Therefore, we need appropriate invariants for disallowing their concurrent execution.

The problem is that adding the invariants in this case might violate the modularity of the SMV model. That is because the execution variable of the Andstate cross transition might be needed to be used in modules other than its home state module.

We distinguish four cases of this kind of transitions and discuss them separately:

1. The lowest Andstate that is a parent of its source state is the same as its scope and the lowest Andstate that is a parent of its destination state is the same as its scope. The home module of the transition is its scope module. It cannot be taken with any transitions in its source part and any transition in its destination part and the required invariants can be defined locally (in its home module).

   The reason is that in the source part (like S1), the source of transitions cannot be orthogonal with the source of t1, and in the destination part (S2), the destination of transitions cannot be orthogonal with the destination of t1. This is the only case that we can add the appropriate invariants without violating the modularity. The required invariants, for example, for Fig. 13 will be:

   ```
   ~(ss.t1_execute & S1.execute);
   ~(ss.t1_execute & S2.execute);
   ```

   In other cases, we cannot have all the required invariants without using the execution variable of the transition in modules other than its home module.

2. The lowest Andstate that is a parent of its source state is the same as its scope and the lowest Andstate that is a parent of its destination state is not the same as its scope. In this case, t1 cannot be taken with transitions in S1 and cannot be taken with some of the transitions in S2 (those that their destinations are not orthogonal with t1's destination). Therefore, we cannot preserve the modularity, since we need the execution variable of t1 in both its scope and its destination part. It has to be used in t1 scope, so that we can say that this transition cannot be taken with any transitions in its source part (S1). We also have to use t1 in some of the substates in S2 to define the appropriate invariants not to let t1 be taken with its inconsistent transitions in S2.

3. The lowest Andstate that is a parent of its source state is not the same as its scope and the lowest Andstate that is a parent of its destination state is the same as its scope. The execution variable of the transition is used in the scope of the transition, so that we can

say that this transition cannot be taken with any transitions in the destination. It also has to be used in the source part (S1) for the definition of appropriate invariants.

4. The lowest Andstate that is a parent of its source state is not the same as its scope and the lowest Andstate that is a parent of its destination state is not the same as its scope. In this case, the transition has to be used in both its source and destination parts.

For cases 2-4, one option is to break the modularity and use the transition execution variable in its non-home modules. The other option is to discover these structures in the translator and not allow them as a well-formed model with Source-Destination and Preemptive options.

In summary, Figs. 14 and 15 illustrate the pseudo-code for adding the appropriate invariants, when the Preemption options is Preemptive.

5.3.2 Non-Preemptive

Based on the Non-Preemptive option, two transitions that one is an "interrupt for" another can be taken in a small-step. We consider this option together with the two possible options of Small-Step Consistency to find the inconsistent transitions for each pair of options.

5.3.2.1 Arena Orthogonal In this section, we discuss the inconsistent transitions based on Non-Preemptive and Arena Orthogonal options.

Orstate

Inconsistent transitions based on Arena Orthogonal in an Orstate are:

1. Simple transitions originating from the same state
2. Simple transitions and Orstate cross transitions with the same scope (that can be enabled with each other)
3. Orstate cross transitions with the same scope
4. Simple transition and transitions in the simple transition's source state
5. Orstate cross transition and transitions in the cross transition's source part.

```
If (Consistency == Arena Orthogonal){
 For each Orstate module{
   For each pair of simple transitions t1 and t2 with the same source state
      Add the invariant ~(ss.t1_execute & ss.t2_execute);
   For each simple transition t1 and its source state A
      Add the invariant ~(ss.t1_execute & A.execute);
   For each pair of Orstate cross transitions t1 and t2 with the same source part
      Add the invariant ~(ss.t1_execute & ss.t2_execute);
   For each Orstate cross transition t1 and its source part A
      Add the invariant ~(ss.t1_execute & A.execute);
   For each pair of simple and Orstate cross transitions t1 and t2
   that can be enabled with each other
      Add the invariant ~(ss.t1_execute & ss.t2_execute);
 }
 For each Andstate module{
    For each Andstate cross transition t1 and its source part A and
    destination part B
      Add the invariants
        ~(ss.t1_execute & A.execute);
        ~(ss.t1_execute & B.execute);
    For each pair of Andstate cross transitions t1 and t2
       Add the invariant ~(ss.t1_execute & ss.t2_execute);
 }
}
```

Figure 14: Pseudo code for generating the invariants for Preemptive option (Part 1)

```
If (Consistency == Source-Destination Orthogonal){
 For each Orstate module{
   For each pair of simple transitions t1 and t2 with the same source state
      Add the invariant ~(ss.t1_execute & ss.t2_execute);
   For each simple transition t1 and its source state A
      Add the invariant ~(ss.t1_execute & A.execute);
   For each pair of Orstate cross transitions t1 and t2 with the same source part
      If the source and destination states are not pairwise orthogonal
         Add the invariant ~(ss.t1_execute & ss.t2_execute);
   For each Orstate transition t1 and its source part A
      Add the invariant ~(ss.t1_execute & A.execute);
   For each pair of simple and Orstate cross transitions t1 and t2
   that can be enabled with each other
      Add the invariant ~(ss.t1_execute & ss.t2_execute);
 }
 For each Andstate module{
   For each Andstate cross transition t1 and source part A and destination part B{
        Add the invariants
          ~(ss.t1_execute & A.execute);
          ~(ss.t1_execute & B.execute);
     }
  }
}
```

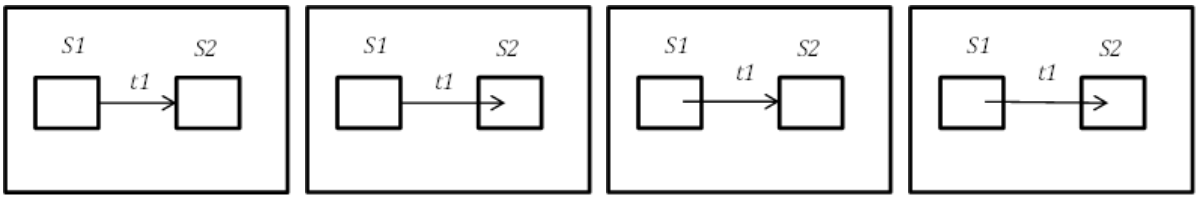
Figure 15: Pseudo code for generating the invariants for Preemptive option (Part 2)

Transitions in (1) and (2) are inconsistent based on the Non-Preemptive option too. Therefore, we add the appropriate invariants, so they will not be taken with each other. However, some of the transitions in (3) may be consistent with each other based on the Non-Preemptive option, and that happens, when one of them is an interrupt for the other one. Therefore, the translator checks the Orstate cross transitions based on interruption, and if they are not consistent, adds the appropriate invariant.

Fig. 16 shows different cases of transitions in (4) and (5). Based on Non-Preemptive, simple transitions and Orstate cross transitions originating from a direct child of the Orstate are inconsistent with transitions in their sources (like t1 and S1 in cases (1) and (2)). Therefore, it is sufficient to add an invariant to the Orstate module not to let them be taken together:

```
~(ss.t1_execute & S1.execute)
```

However, we cannot say the same thing about cases (3) and (4).



(a) case 1 (b) case 2 (c) case 3 (d) case 4

Figure 16: Different cases of transition-state inconsistency in an Orstate

In cases (3) and (4), t1 might be consistent with some of the transitions in S1. That happens, when the source and destination of a transition in S1 is orthogonal with the source of t1 ,and therefore, t1 is an interrupt for the transition. Otherwise, we must have an invariant to disallow their concurrent execution. Adding such invariants will violate the SMV modularity, since the scope of t1 is the Orstate and inner transitions in S1 are defined in lower level modules. For example, in Fig. 17, you can see inconsistent transitions with t1 in dashed line.

To overcome this problem, we change the home module of t1 in order to define invariants without violating the modularity. The new home module has to be lower than the transition scope; otherwise, the enabledness variable of t1 might be lost when cutting a part of the SMV model, corresponding to a submodel of the original CHTS model.

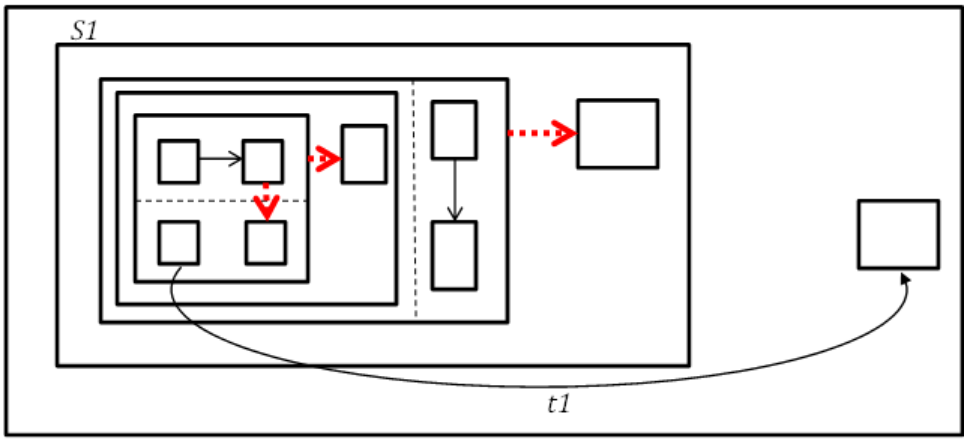


Figure 17: An example of an Orstate cross transition and its inconsistency with transitions in its source part

We define t1 enabledness variable (in cases (3) and (4)) in the lowest Andstate in its source part (S1) that is its source or source ancestor. If there is not such an Andstate, then, t1's home module will be its scope. The intuitive reason behind this approach is that if there is no Andstate in S1 that is its source or source ancestor, then, no transition in S1 can be taken with t1. Therefore, we can define t1 in its scope, and add an invariant not to let t1 and S1 be taken together. However, if there is such an Andstate in S1, then, t1 cannot be taken with transitions in the child of the Andstate that its source resides in, but can be taken with transitions in the other child of the Andstate (it is an interrupt for those transitions). So, we set t1's home module in this Andstate, and add an invariant not to let t1 be taken with transitions in the child that its source resides in. The consistency of other transitions in S1, outside of this Andstate, with t1 is *often* (except for a non-conformance) the same as their consistency status with transitions in this Andstate. Therefore, t1 can be considered as a transition in this Andstate, and be part of its execution flag. In the following, we prove our claim and show the non-conformance we will have.

In the Orstate in Fig. 18, we want to show that the consistency of every transition in S1, outside of S2, with t1 is the same as its consistency with transitions in S2.

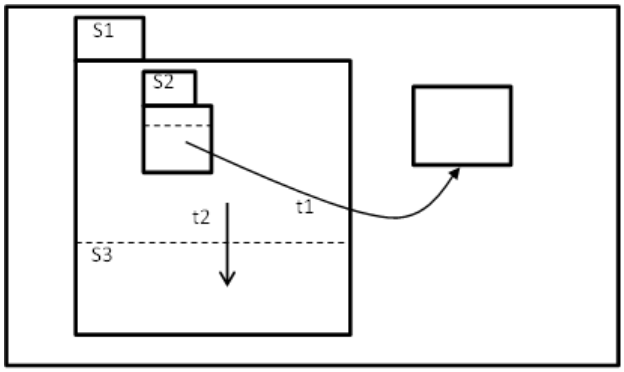


Figure 18: An Orstate with a cross transition (t1)

For all states that are S2's siblings within an Orstate, they cannot be enabled with transitions in S2, neither with t1. So, there is no need for invariants for any of them. For the transitions originating from one of the ancestors of S2, they cannot be taken with transitions in S2, neither with t1. But if a state is S2's sibling within an Andstate, like S3, then the transitions in S3 can be enabled with transitions in S2, and also with t1. The transitions in S3 can be taken with t1 based on Preemption, since their sources are orthogonal. They can also be taken with transitions in S2 based on Arena Orthogonal option. The only non-conformance is the unusual transitions as described before. These transitions are not allowed by our translator. The Andstate cross transitions like t2 cannot be taken with transitions in S2, neither with t1.

The other category of transitions in S1 are Orstate cross transitions. With this kind of transitions, the problem arises when the cross transition is originating from an orthogonal state with the origin of t1, and they are defined in different Andstates, but they cannot be taken together based on the Preemption option. For example, in Fig. 19, t2 can be taken with transitions in S2 based on Preemption, but not with t1. One option to deal with this non-conformance is to violate the modularity to define the appropriate invariants. The other option is to disallow this model structure in the translator.

Andstate

Based on Arena Orthogonal, the transitions inside the children of an Andstate are consistent, except for unusual transitions, which are not allowed by our translator. Andstate cross transitions

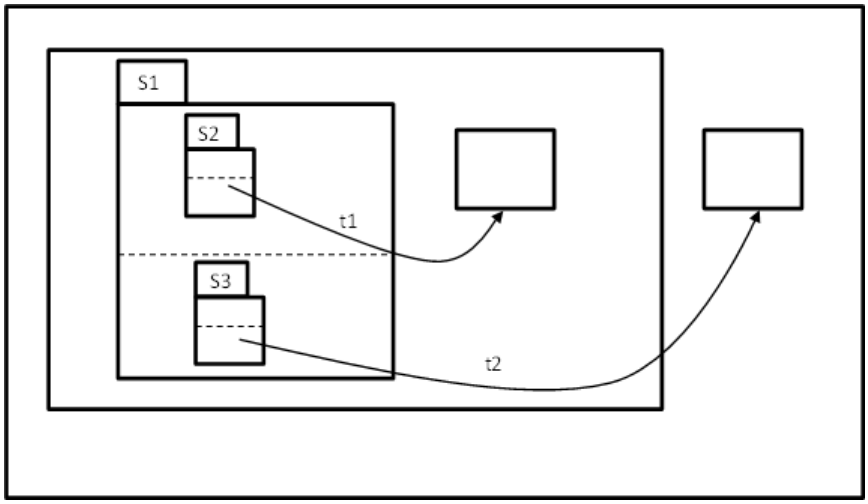


Figure 19: Two inconsistent Orstate cross transitions

are inconsistent with transitions in the children of the Andstate and with each other. These transitions are inconsistent based on the Non-Preemptive option too. So, we take care of them by adding appropriate invariants. For example, for the Andstate in Fig. 10, we add the following:

```
~(ss.t1_execute & S1.execute);
~(ss.t2_execute & S2.execute);
~(ss.t2_execute & S1.execute);
~(ss.t2_execute & S2.execute);
~(ss.t1_execute & ss.t2_execute);
```

5.3.2.2 Source-Destination orthogonal In this section, we discuss the inconsistent transitions based on Non-Preemptive and Source-Destination Orthogonal options.

Orstate

This case is similar to the Orstate in Non-Preemptive and Arena Orthogonal, with the difference that there is no unusual transition.

Andstate

This case is similar to the Andstate with Preemptive and Source-Destination Orthogonal options.

In summary, Figs. 20 and 21 illustrate the pseudo-code for adding the appropriate invariants, when the preemption options is Non-Preemptive.

5.4 Well-Formedness Rules

Our purpose is a modular translator in which the structure of the resulting SMV model matches the source CHTS structure. As we showed, there are a number of CHTS structures that do not allow this modular translation (modularity non-conformances). The details of why these structure do not allow modular translation were discussed in Section 5.3. In order to create a modular translation, we add new well-formedness rules to the CHTS structures that disallow CHTS structures that do not conform to the modularity rules.

1. Fig. 22 is an example of a CHTS model with the first kind of modularity non-conformances. The first kind of modularity non-conformances exist in CHTS models with an Andstate (such as S in Fig. 22) containing an Andstate as a child (S2) that has an Andstate cross transition (t1), when the Small-Step Consistency option is Arena Orthogonal. As discussed in Section 5.3, this is a bad modeling decision, since the Andstate cross transition (t1) cannot be taken with transitions in the other child of the Andstate (S1), and this is in contradiction with the nature of Andstates and may lead to model behaviors that are not expected by the modeler. Therefore, since it will also lead to modularity non-conformance, we do not allow this structure when the Small-Step Consistency option is Arena Orthogonal.

2. The second modularity non-conformances exist in CHTS models with at least two Orstate cross transitions (such as t1 and t2 in Fig. 23) with the following conditions:

   - originating from orthogonal states,
   - the Andstate that is the common ancestor of the source states (S1) is a descendent of the transition scopes,
   - the lowest Andstate containing the source state of at least one of the transitions (S2 for t1, and S3 for t2) is not the same as their common ancestor (S1), and
   - the Preemption option is Non-Preemptive.

```
If (Consistency == Arena Orthogonal){
 For each Orstate module{
   For each pair of simple transitions t1 and t2 with the same source state
     Add the invariant ~(ss.t1_execute & ss.t2_execute);
   For each simple transition t1 and its source state A
     Add the invariant ~(ss.t1_execute & A.execute);
   For each pair of Orstate cross transitions t1 and t2 with the same source part
     If none of them is an interrupt for the other one
       Add the invariant ~(ss.t1_execute & ss.t2_execute);
   For each pair of simple and Orstate cross transitions t1 and t2
   that can be enabled with each other
     Add the invariant ~(ss.t1_execute & ss.t2_execute);
   For each Orstate cross transition t1 and its source part A
     If t1 is defined in its own scope{
       Add the invariant ~(ss.t1_execute & A.execute);
     }else{
       Add the appropriate invariant to its home module state
     }}
  For each Andstate module{
    For each Andstate cross transition t1 and its source part A and
    destination part B
      Add the invariants
       ~(ss.t1_execute & A.execute);
       ~(ss.t1_execute & B.execute);
    For each pair of Andstate cross transitions t1 and t2
      Add the invariant ~(ss.t1_execute & ss.t2_execute);
    }}}
```

Figure 20: Pseudo code for generating the invariants for Non-Preemptive option (Part 1)

```
If (Consistency == Source-Destination Orthogonal){
  For each Orstate module{
   For each pair of simple transitions t1 and t2 with the same source state
     Add the invariant ~(ss.t1_execute & ss.t2_execute);
   For each simple transition t1 and its source state A
     Add the invariant ~(ss.t1_execute & A.execute);
   For each pair of Orstate cross transitions t1 and t2 with the same source part{
     If none of them is an interrupt for the other one
       Add the invariant ~(ss.t1_execute & ss.t2_execute);
   }
   For each pair of simple and Orstate cross transitions t1 and t2
   that can be enabled with each other
     Add the invariant ~(ss.t1_execute & ss.t2_execute);
   For each Orstate cross transition t1 and its source part A{
     If t1 is defined in its own scope{
       Add the invariant ~(ss.t1_execute & A.execute);
     }else{
       Add the appropriate invariant to its home module state
     }
   }
  }
  For each Andstate module{
   For each Andstate cross transition t1 and source part A and destination part B{
       Add the invariants
         ~(ss.t1_execute & A.execute);
         ~(ss.t1_execute & B.execute);
    }
  }
}
```

Figure 21: Pseudo code for generating the invariants for Non-Preemptive option (Part 2)

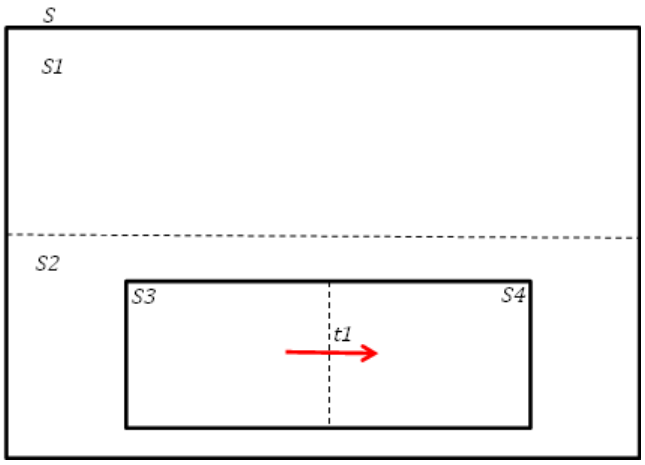


Figure 22: An example of the first modularity non-conformances model structure

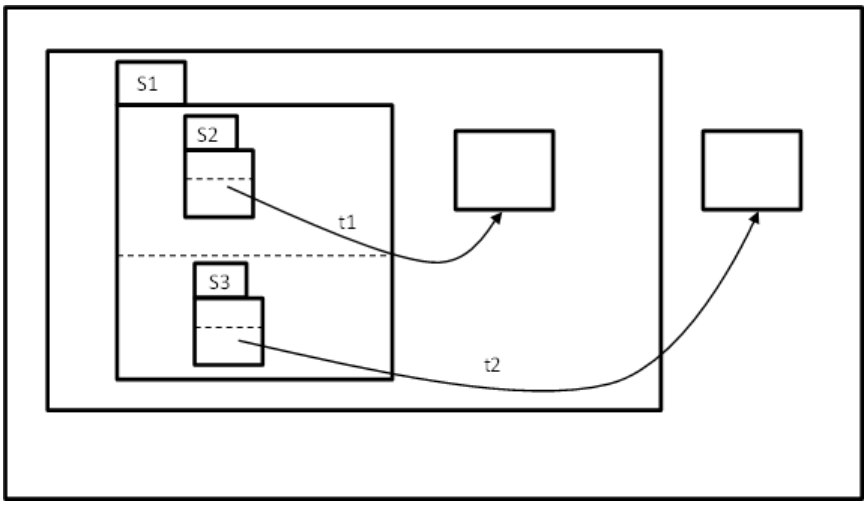


Figure 23: An example of the second modularity non-conformance model structure

3. With the Source-Destination Orthogonal option, there is a group of transitions that violate modularity; any Andstate cross transition that the lowest Andstate that is a parent of its source state is not the same as its scope, or the lowest Andstate that is a parent of its destination state is not the same as its scope.

The first modularity non-conformance is not allowed by our translator. The next two modularity non-conformances can be unacceptable, or the modeler might choose to break the modularity of the resulting SMV model by allowing these structures.

5.5 Priority

At each snapshot of a model, there could exist more than one set of transitions for execution as a small-step. The modeling language can have a way of assigning priority to transitions to avoid non-determinism. In Table 2, we summarize the ways for priority assignment based on Esmaeilsabzali's semantic aspects and options.

Options	Definition
HIERARCHICAL	The source and destination control states of transitions determine priority.
EXPLICIT PRIORITY	Each transition is given an explicit, relative priority.
NEGATION OF TRIGGERS	A transition is given higher priority than another by strengthening the event trigger of the second transition such that it is not enabled when the first transition is enabled.
NO PRIORITY	All transitions have equal hierarchical priority.

Table 2: Priority semantic options [13]

The Priority semantic aspect is applicable between inconsistent transitions. Based on Esmaeilsabzali's semantic scheme, a transition with lower priority cannot prevent a higher priority transition from getting executed. With the No Priority option, the SMV model does not need any changes, since all transitions have equal priority.

With the Hierarchical option, we need appropriate invariants between inconsistent transitions and state modules included in each module. The invariant indicates that when the higher priority one is enabled, the other one cannot get executed. If the Hierarchical option gives higher priority to lower level transitions, we add the following for each inconsistent pair of transition-module included in a state module:

```
(module1.enabled -> ~t1_execute)
```

and if the Hierarchical option gives higher priority to higher level transitions, we add the following for each inconsistent pair of transition-module included in a state module:

```
(t1_enabled -> ~module1.execute)
```

The reason is that the included transition belongs to a higher level of the CHTS model than transitions in the included state module. We can use this method only when the home module of each transition is in its scope; otherwise we use our method for the Explicit priority.

When the Priority option is Explicit, we define a priority macro for each state module whose value is the priority of the enabled transition or the priority macro of the state module in it with the highest value. Each transition may also have an assigned priority, which is given to the translator with the CHTS model. Then, in each state module, we add the appropriate invariants for inconsistent transitions, and pairs of transition-state module. For each inconsistent pair of transitions, we add the following invariant (assuming t1 has higher priority than t2):

```
t1_enabled -> ~t2_execute;
```

and for each pair of inconsistent module-transition, we add the following macro in the corresponding state module:

```
(module1.priority > t1_priority) -> (module1.enabled -> ~t1_execute)
(module1.priority < t1_priority) -> (t1_enabled -> ~module1.execute)
```

The invariant states that if the state module has a higher priority enabled transition in it, it should not be prevented from execution by the transition. Similarly, if the transition's priority is higher than the priority of every transition in the state module, it has precedence in the execution.

If the Priority option is Negation of Triggers, it is applied using the transition conditions in the model execution, therefore, no change in the SMV model is required.

The pseudo code for generating the required invariants when the Priority option is not No Priority is illustrated in Fig. 24.

5.6 Invariants

In this section, we summarize the invariants that the translator adds to the SMV model. We distinguish two categories of invariants in the resulting SMV model:

```
If (Priority == Hierarchical){
  If (the Hierarchical option gives higher priority to lower level transitions){
      For each inconsistent pair of transition-module in a state module{
        Add the invariant
        (module1.enabled -> ~t1_execute);
      }
  }
  else{
    For each inconsistent pair of transition-module in a state module{
        Add the invariant
        (t1_enabled -> ~module1.execute);
      }
  }
}
else if (Priority == Explicit){
    Define a priority macro for each state module whose
    value is the priority of the enabled transition or
    the priority macro of the state module in it with the highest value;
    For each inconsistent pair of transitions{
        Add the invariant (assuming t1 has higher priority than t2):
        t1_enabled -> ~t2_execute;
    }
    For each pair of inconsistent module-transition{
        Add the invariants:
        (module1.priority > t1_priority) -> (module1.enabled -> ~t1_execute);
        (module1.priority < t1_priority) -> (t1_enabled -> ~module1.execute);
    }
}
```

Figure 24: Pseudo code for generating the required invariants for Priority options

1. Semantics-Independent Invariants

   These are invariants needed for the SMV model, regardless of the semantic options. There is one group of these invariants:

   - The invariant needed to guarantee that if there is at least one enabled transition in a snapshot, at least one transition gets executed in the next small-step. For this purpose, we add the following invariant to the root module:

     ```
     enabled -> execute;
     ```

     The enabled macro in the root module is true, when at least one transition is enabled, and execute is true, when at least one transition gets executed in the next small-step.

2. Semantics-Dependant Invariants

   These are invariants required in the SMV model, depending on the values of the semantic options. There are three groups of these invariants:

   - The invariants needed to disallow concurrent execution of inconsistent transitions. When the concurrency option is Single, every two transitions are inconsistent. Inconsistent transitions for the Many option are explained in Section 5.1.2.
   - The invariants that apply the Priority options. Theses invariants are needed for inconsistent transitions with different priorities, and are explained in Section 5.5.
   - The invariants that enforce the execution of each transition, when it has no restriction for execution. These invariants are needed, when the Concurrency option is Many. For example, in the state module S with children S1 and S2, and the transition t1, in which S2 is inconsistent with S1 and t1, we add the following invariant for execution enforcement of S2:

     ```
     execute -> ((S2.enabled & ~t1_execute & ~S1.execute) -> S2.execute)
     ```

     Similar invariant are added for S1 and t1. It indicates that if the inconsistent transition (t1) and state module (S2) are not going to be taken, and there is at least one enabled transition in the S2, then at least one transition in the state module S2 must be taken.

We have "execute" in the left hand side, since the module S might be used in other invariants in higher levels of the hierarchy. So, it might be the case that S is not supposed to be taken because of other transitions or modules. Therefore, to keep the invariants consistent, we indicate that if the execute macro of the state module is true, then, every transition inside must be taken, if it doesn't have any restrictions.

6 Dynamic Parameters

Based on Esmaeilsabzali's semantic deconstruction, there are six dynamic parameters; Big-Step Maximality, Event Lifeline, Enabledness Memory Protocol, Assignment Memory Protocol, Order of Small-Steps, and Combo-Step Maximality. In this section, we discuss different options for dynamic parameters and how they can affect the resulting SMV model. We do not consider BSMLs with combo-steps in this work, therefore, will not discuss the Combo-Step Maximlaity aspect in this report.

6.1 Big-Step Maximality

The Big-Step Maximality semantics determines when the sequence of small-steps of a big-step concludes. Table 3 lists the three possible semantic options of this aspect. The Big-Step

Options	Definition
SYNTACTIC	No two transitions with overlapping arenas that enter designated "stable" control states can be taken in the same big-step.
TAKE ONE	No two transitions with overlapping arenas can be taken in the same big-step.
TAKE MANY	Small-steps continue until there are no more enabled transitions.

Table 3: Big-Step Maximality semantic options [13]

Maximality semantic option is applied in the next statements of variables, events, and control states in the snapshot module. A condition is defined on these next statements, which reflects

the Big-Step Maximality semantic option. Whenever this option gets true, the CHTS model resets and the input is read from the environment.

When the Big-Step Maximality option is Take Many, small-steps continue until there are no more enabled transition. We need to recognize snapshots with no enabled transitions, which we call stable snapshots. A variable named "stable" is defined in the snapshot module, and its value is assigned to the negation of "enabled" in the root state module.

```
ss.stable := ~enabled;
```

Whenever, the enabled macro of the root state module is false, there is no enabled transitions in the model, and therefore, the model is in a stable snapshot. We use the stable variable to recognize the stable snapshots, and reset the SMV model. For example, for a CHTS model with events event1 and event2, and variables var1 and var2, the next statements of events and variables in the snapshot module will be:

```
default
{next(event1) := [Based on the Event Lifeline option];
 next(event2):= [Based on the Event Lifeline option];
 next(var1) := [To be explained in Section 6.3.1];
 next(var2):= [To be explained in Section 6.3.1];}
in
 {
  if(stable)
  {
    next(event1) := input.event1;
    next(event2) := event2;
    next(var1) := input.var1;
    next(var2) := var2;
  }
 }
```

When the model resets, some of the events (which are called environmental input events in [13]), and some of the variables (which are called environmental input variables in [13]) get

value from the environment, and some of them keep their values. In the above example, event1 is an environmental input event, and var1 is an environmental input variable.

With "Take One" and "Syntactic" options, the translator distinguishes the transitions that cannot be taken together in the same big-step. For each such transition, a Boolean variable is defined in the snapshot module. For the transition t1, the Boolean variable has the name t1_ever_execute. The next value of these variables are stated as below:

```
default
{next(t1_ever_execute) := t1_ever_execute | t1_execute;}
in
 {
  if(stable)
  {
    next(t1_ever_execute) := 0;
  }
 }
```

t1_ever_execute variable has the value 1, if t1 has been executed in the current big-step. For every transition that cannot be taken with t1, the negation of t1_ever_execute is conjuncted with its event condition to apply the Big-Step Maximality option.

6.2 Event Lifeline

Events generated by transition execution are sensed by event triggers of all transitions in the model. The snapshots of a big-step in which a generated event can be sensed as present are determined by the Event Lifeline semantic aspect. Table 4 lists the three possible semantic options of this aspect.

We are not considering the modeling languages with the Present In Whole option in our study. The option for the Event Lifeline semantic aspect affects the next statements of the events in the snapshot module. With the Present in Remainder option, the event is present in the next snapshot, if it is present in the current snapshot, or one of the transitions that generate this event gets executed in the next small-step. For example, if the CHTS model has two events,

Options	Definition
PRESENT IN WHOLE	A generated event in a big-step is assumed to be present throughout the same big-step.
PRESENT IN REMAINDER	A generated event in a big-step is sensed as present in the same big-step after it is generated.
PRESENT IN NEXT SMALL-STEP	A generated event can be sensed as present only in the next small-step after it is generated.
PRESENT IN SAME	A generated event can be sensed as present only in the same small-step it is generated in.

Table 4: Event Lifeline semantic options [13]

event1 and event2, and event1 is generated by the transitions, t1 and t2, and event2 is generated by the transitions t3 and t4, then, the next statements of the events will be as followed:

```
default
{next(event1) := event1 | t1_execute | t2_execute;
 next(event2):= event2 | t3_execute | t4_execute;}
in
 {
  if(stable)
  {
    [Based on the Big-Step Maximality option]
  }
 }
```

When the Event Lifeline option is Present in Next Small-Step, then, the next value of the events will be true, if it is generated by at least a transition in the next small-step. Therefore, the next statements of the events for a similar example will be:

With the Present in Same option, a generated event can be sensed as present only in the same small-step it is generated in. Therefore, each event will be a macro, rather than a Boolean variable, and its value will be a disjunction of the execution variables of the transitions that

```
default
{next(event1) := t1_execute | t2_execute;
 next(event2):= t3_execute | t4_execute;}
in
 {
  if(stable)
  {
    [Based on the Big-Step Maximality option]
  }
 }
```

generate the event. For example, for the similar example, the events are defined in the snapshot module as followed:

```
event1 := t1_execute | t2_execute;
event2 := t3_execute | t4_execute;
```

6.3 Enabledness Memory Protocol

The Enabledness Memory Protocol option specifies the values of variables that a transition reads for its guard condition (GC). Table 5 lists the three possible semantic options of this aspect.

Options	Definition
GC SMALL-STEP	The value of a variable is its up-to-date value, obtained from the beginning of the small-step.
GC BIG-STEP	The value of a variable during a big-step is obtained from the beginning of the big-step.
GC COMBO-STEP	The value of a variable during a combo step is obtained from the beginning of the combo-step.

Table 5: Enabledness Memory Protocol semantic options [13]

We are not considering BSMLs with combo-steps, therefore, we do not discuss the third

option. Before discussing how to apply the Enabledness Memory Protocol semantic options, we talk about assigning the next value to the variables.

6.3.1 Next Value Assignment to Variables

Value assignment to a variable is done by a transition execution. The problem that may arise by value assignment is "race condition". Race condition occurs when different executing transitions assign different values to the same variable. When this happens, one of the values are assigned to the variable non-deterministically [13]. To apply this in the SMV model, the translator generates the next statements of the variables in a way that if more than one transition assign values to the same variable, one of them will be assigned non-deterministically. For example, if t1, t2, and t3 assign val1, val2, and val3 to var1 respectively, then, the next statement of var1 in the snapshot module will be:

```
default
{next(var1) :=
   case {
    t1_execute & t2_execute & t3_execute : {val1, val2, val3};
    t1_execute & t2_execute : {val1, val2};
    t2_execute & t3_execute : {val2, val3};
    t1_execute & t3_execute : {val1, val3};
    t1_execute : val1;
    t2_execute : val2;
    t3_execute : val3;
    1 : var1;
   };}
in
 {if(stable){
    next(var1) := input.var1;
    next(var2) := var2;
  }}
```

In the above example, var1 is an environment variable, while var2 keeps its value, when the model resets.

6.3.2 Applying the Enabledness Memory Protocol Options

With the Small-Step option, no change is required in the SMV model, since the value of a variable in GC is its up-to-date value. However, with the Big-Step option, we need to find a way that the value of variables in GC are obtained from the beginning of the big-step. With this option, a variable, which we call copy variable, is defined in the snapshot module for each variable in GC. For each variable var1, the copy variable is named var1_copy. The variables in GC, which are used in the transition enabledness macros in the SMV model, are replace with their copy variables.

6.4 Assignment Memory Protocol

The Assignment Memory Protocol semantic aspect specifies the values of variables that a transition reads when evaluating the righthand side (RHS) of an assignment. The semantic options are exactly similar to the Enabledness Memory Protocol options. The options are also applied in a similar way to the SMV model with the difference that the variables in RHS are replaced with their copy variables.

6.5 Order of Small-Steps

At a snapshot, when it is possible to execute more than one set of transitions, there are different options to choose a set for execution. Some BSMLs non-deterministically choose one set of transitions, while others have an order among sets for execution. Table 6 lists the three possible semantic options of this aspect.

When the Order of Small-Steps semantic option is None, one set of transitions is executed non-deterministically. With this option, the SMV model needs no change, since having invariants on the non-allowed combinations of transitions, one of the allowed combinations will be chosen by SMV non-deterministically.

Options	Definition
NONE	Small-steps are not ordered.
EXPLICIT ORDERING	Execution of small-steps is ordered syntactically.
DATAFLOW	Small-steps are ordered so that an assignment to a variable happens before it is being read.

Table 6: Order of Small-Steps semantic options [13]

6.5.1 Explicit Ordering

The Explicit Ordering order of small-steps is relevant for the transitions within the scope of an Andstate. Based on the Explicit Ordering formalization in [13], a transition is enabled, only if its predecessors are already executed, or they are not enabled. Moreover, it is assumed in [13] that the Explicit Ordering is always chosen together with the Take One option. Based on Take One option, if a transition is already executed, it cannot be enabled at the current snapshot. Therefore, we can say that a transition is enabled, only if none of its predecessors are enabled.

The execution of transitions in an Andstate children are ordered according to their graphical order. With the Explicit Ordering, ordering is imposed on the transitions execution using invariants. Each Andstate has a set of invariants for enforcing the order of execution among its direct children. For example, if the Andstate has three substates, S1, S2, and S3, and the order is S1, S2, S3, then, the following invariants are added to enforce the ordering:

```
S1.enabled -> ~S2.execute & ~S3.execute;
S2.enabled -> ~S3.execute;
```

6.5.2 Dataflow

With the Dataflow order, small-steps that assign values to a variable happen before the ones that read the variable's value. Based on the formalization of this option in [13], a transition, t, is enabled if all variables in prefix_new(t) are assigned values during the current big-step. prefix_new(t) is the set of variables prefixed by new that are used in the guard condition (GC) of t or in the righthand side (RHS) of an assignment in t.

For applying the Dataflow option in the SMV model, we define a Boolean variable for each variable prefixed by new in the GC or RHS of a transition. The variable is called v_assigned for the variable v, and its value is 1, if the variable has been assigned a value in the current big-step. Therefore, the next value of the variable is 1, if it is already 1, or a transition that assigns value to it gets executed. When the model resets, the variable gets the value 0. For example, for the variable var1 that is assigned value by t1 and t2, the next value of var1_assigned in the snapshot module will be:

```
default
{next(var1_assigned) := var1_assigned | t1_execute | t2_execute;
in
 {
  if(stable)
  {
    next(var1_assigned) := 0;
  }
 }
```

The variable will be used in the enabledness macro of transitions with this variable in their GC or RHS, so that they cannot get executed before the variable is assigned value in the current big-step.

6.6 Next Value Assignment to Control States

In the SMV model, a Boolean variable is defined for each basic control state. Other control states are defined using a macro, based on the value of their direct children. In this section, we discuss the next value assigned to the Boolean variables of basic control states.

The next value assigned to control states (defined in the snapshot module) can be considered as a fixed dynamic parameter. A basic control state may be entered, exited, and interrupted in the same small-step. So, we should find the precedence between the entering, exiting and interrupt transitions. The interrupt transitions have the highest precedence, since when an interrupt transition exits a control state, its next value will be 0, regardless of other entering and

exiting transitions. Between entering and exiting transitions, entering transitions have higher precedence. The reason is that some transitions are both entering and exiting a control state. For example, t1 in Fig. 25 is in both sets of entering and exiting transitions for the control state S1, and as can be seen, S1 is actually entered.

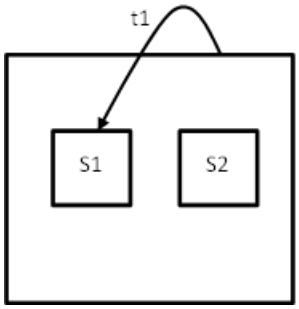


Figure 25: An example of a transition both entering and exiting a control state

For each basic control state, the translator finds the set of transitions that enter the state, the set of transitions that exit the control state, and the set of interrupt transitions that exit it. The next statement of the control state will be defined based on these three sets of transitions. The precedence among the three sets is applied by the order of checking these transitions. SMV checks for conditions in a top-down manner, and for the first condition that is satisfied, the corresponding statement will be applied. For example, if t1 exits S1, t2 and t3 enter it, and t4 is an interrupt transition exiting S1, then, the next statement for the basic control S1 will be:

```
next(S1) := case {
             t4_execute : 0;
             t2_execute | t3_execute : 1;
             t1_execute : 0;
             1 : S1;
            };
```

7 Breaking Apart CHTS Models

One of the contributions of this research is to make SMV models with modular structures similar to the original CHTS model structures. One of the benefits of modular translation is that having the resulting SMV model, the user can have the SMV model corresponding to a submodel of the original CHTS model.

By submodel extraction, we mean taking one module with all its submodules, and also cross transitions. The steps for finding the corresponding SMV submodel are as follows:

- Ignore the state modules corresponding to the control states that are removed from the original CHTS model.
- For every transition, t1, removed from the CHTS model, add:

  ```
  init(t1_execute) := 0;
  next(t1_execute) := 0;
  ```

  to the snapshot module. Cone of influence (COI) reduction [14] in model checking will remove these variables and any control states that become unreachable in model checking.
- Add:

  ```
  ss.stable := enabled;
  enabled -> execute;
  ```

  to the new root module.

8 Related Work

Model checking is a popular method for verification of models designed in the process of software development. The problem is that the model checking tools have their own languages for model description. It is not an easy and straightforward process to translate the models designed using different modeling languages to the input languages of model checkers. We can categorize the work in this area into three major groups; direct translation, using intermediate languages, and semantics-based translation.

In the first method, a model in a model-based requirements notation is translated directly to the input language of a verifier (such as a model checker) (e.g., SCR to EMC model checker [15], RSML to SMV [16], SCR to SPIN and SMV [17], and Stateflow to SMV [18]). It is obvious that these work cannot solve the problem permanently, since new modeling languages with new syntaxes and semantics are created everyday. To reduce the number of required translations, intermediate languages have been designed, such as SAL [4], Action Language [5], and Bandera Intermediate Representation (BIR) [19]. In these approaches, the models designed in different modeling languages are translated to an intermediate language, and from that language to the input language of a verifier.

The third approach is to generate an analysis tool based on a notation's semantics (e.g., Day *et. al* in [20] and Pezze *et. al* in [21]). The problem with these approaches is that it is not easy to define the semantics of different modeling languages. To overcome this problem, Lu *et. al* in [6] used a set of parameters, called template semantics, to describe the semantics of modeling languages.

Niu originally created template semantics for her PhD thesis [22]. In template semantics, a modeling language is described using a set of values for 22 parameters with a choice of composition operators. Lu *et. al* in [6] described how to use the semantic decomposition provided by template semantics to facilitate notation-specific analysis. They developed a translator for CHTSs from template parameter values into SMV in a tool called Express. Express was originally built on top of a tool called fusion that Day created for her PhD thesis [23]. Express takes as input a specification in CHTS notation and a set of template parameters detailing the notation's semantics; the translator combines these inputs with the template's common-semantics definitions, to generate an SMV model of the specification.

Esmaeilsabzali has come up with a new deconstruction for the semantics of BSMLs that is more abstract and uses fewer parameters [13]. The goal of this project was to propose the algorithm of translation to create an "Express"-like translator for Esmaeilsabzali's parameters.

9 Conclusion

We proposed a semantics-based, parameterized algorithm for translating big-step modeling languages to SMV. The translator takes the options for structural and dynamic semantic aspects and a specification in the CHTS notation, and it produces an SMV model. Using this translator, it is possible to model check specifications written in big-step modeling languages. As future work, we will be implementing the proposed algorithm to automate the procedure of translation. The implementation can lead to more evaluation of the algorithm.

References

[1] K. McMillan, *Symbolic Model Checking*, Kluwer Academic Publishers, 1993.

[2] G.J. Holzmann, "The model checker SPIN", *IEEE Transactions on Software Engineering*, pp. 279–295, 1997.

[3] S. Esmaeilsabzali, N.A. Day, J.M. Atlee, and J. Niu, "Deconstructing the semantics of big-step modelling languages", *Requirements Engineering*, pp. 235–265, 2010.

[4] S. Bensalem, V. Ganesh, Y. Lakhnech, C. Munoz, S. Owre, H. Rueß, J. Rushby, V. Rusu, H. Saıdi, N. Shankar, et al., "An overview of sal", in *NASA Langley Formal Methods Workshop*, 2000.

[5] T. Bultan, "Action language: A specification language for model checking reactive systems", in *International Conference on Software Engineering*, 2000, pp. 335–344.

[6] Y. Lu, J.M. Atlee, N.A. Day, and J. Niu, "Mapping template semantics to smv", in *International Conference on Automated Software Engineering*, 2004, pp. 320–325.

[7] J. Niu, J.M. Atlee, and N.A. Day, "Template semantics for model-based notations", *IEEE Transactions on Software Engineering*, pp. 866–882, 2003.

[8] D. Harel, "Statecharts: A visual formalism for complex systems", *Science of Computer Programming*, pp. 231–274, 1987.

[9] F. Maraninchi and Y. Rémond, "Argos: an automaton-based synchronous language", *Computer Languages*, pp. 61–92, 2001.

[10] R. Alur and T.A. Henzinger, "Reactive modules", *Formal Methods in System Design*, pp. 7–48, 1999.

[11] C.L. Heitmeyer, R.D. Jeffords, and B.G. Labaw, "Automated consistency checking of requirements specifications", *ACM Transactions on Software Engineering and Methodology*, pp. 231–261, 1996.

[12] S. Esmaeilsabzali and N.A. Day, "Prescriptive semantics for big-step modelling languages", *Fundamental Approaches to Software Engineering*, pp. 158–172, 2010.

[13] S. Esmaeilsabzali, *Prescriptive Semantics for Big-Step Modelling Languages*, PhD thesis, Univeristy of Waterloo, 2011.

[14] O. Grumberg and D.A. Peled, *Model checking*, The MIT Press, 1999.

[15] J.M. Atlee and J. Gannon, "State-based model checking of event-driven system requirements", *IEEE Transactions on Software Engineering*, pp. 24–40, 1993.

[16] W. Chan, R.J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J.D. Reese, "Model checking large software specifications", *IEEE Transactions on Software Engineering*, pp. 498–520, 1998.

[17] R. Bharadwaj and C.L. Heitmeyer, "Model checking complete requirements specifications using abstraction", *Automated Software Engineering*, pp. 37–68, 1999.

[18] A.L. Juarez-Dominguez, N.A. Day, and R.T. Fanson, "Translating models of automotive features in matlabs stateflow to smv to detect feature interactions", in *International Systems Safety Conference*, 2008.

[19] J.C. Corbett, M.B. Dwyer, J. Hatcliff, S. Laubach, C.S. Pasareanu, and H. Zheng, "Bandera: Extracting finite-state models from java source code", in *International Conference on Software Engineering*, 2000, pp. 439–448.

[20] N.A. Day and J. Joyce, "Symbolic functional evaluation", *Theorem Proving in Higher Order Logics*, pp. 839–839, 1999.

[21] M. Pezze and M. Young, "Creating of multi-formalism state-space analysis tools", in *Proceedings of the International Symposium on Software Testing and Analysis*, pp. 172–179.

[22] J. Niu, *Template Semantics: A Parameterized Approach to Semantics-Based Model Compilation*, PhD thesis, Univeristy of Waterloo, 2005.

[23] N.A. Day, *A Framework for Multi-Notation, Model-Oriented Requirements Analysis*, PhD thesis, University of British Columbia, 1998.