Formal Verification of the A-7E Software Requirements using Template Semantics

Eunsuk Kang Nancy A. Day

David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1
{ekang,nday}@uwaterloo.ca

Technical Report: CS-2006-35
September 2006

Abstract. Template semantics is a template-based approach to ease the process of identifying the essential differences among model-based notations. In this approach, a template captures semantics that are common among notations and allows users to specify only the distinctive features of a notation. In this paper, we illustrate the method of describing requirements in Software Cost Reduction (SCR) using the Metro toolkit, which is the framework for the modelling and analysis of notations in template semantics. Furthermore, we demonstrate the usage of Metro to verify the A-7E software requirements and compare our verification effort to an alternative method of requirements analysis, which does not use template semantics.

1 Introduction

Requirement writers must be able to clearly understand the specification notations that they use and identify the differences among the semantics of these notations. However, comparing different notations can be a difficult and time-consuming process for users with limited expertise in the notations. In order to ease this process, the researchers at the University of Waterloo have developed a template-based approach to describe the semantics of model-based notations [9]. In this approach, semantics that are common to model-based notations are pre-defined in a template, and users specify only the distinctive features of a notation as input parameters to the template.

The Metro toolkit [2] is an extensive set of tools developed to support the modelling and analysis of software specifications in template semantics. Metro includes tools that allow users to specify notation-specific semantics in a textual format or using a graphical tool such as MagicDraw. In addition, the toolkit currently provides translation from a template-based notation to the input language of SMV. In effect, Metro is an unified environment under which a variety of tools can be constructed for the analysis of requirements in template-based

notations. As a result, users of Metro do not need to create a separate translator from a notation to the input language of an existing formal analysis tool.

In order to demonstrate the robustness of the Metro toolkit, we have chosen to model the *Software Requirements for the A-7E Aircraft* [9] using template semantics. The A-7E requirements were originally written in Software Cost Reduction (SCR) [6]. After mapping the SCR requirements to a template, we translated the specification in template semantics to an equivalent SMV model using Metro Express [7], the translator to SMV. In comparison to previous case studies that have been done on Metro, the A-7E requirements specification is the largest in size. Therefore, the A-7E case study also serves as a test for the scalability of the Metro toolkit.

Previously, Sreemani performed the direct translation of the A-7E requirements into an SMV model and verified the properties that had been specified in the original requirements document [10]. Sreemani's translator does not involve template semantics. Since our template-based approach is geared toward the generality of semantics, we expect a loss of efficiency in the analysis phase. Thus, our SMV model and Sreemani's model are equivalent in semantics, but they significantly differ in size. A comparison between the performances of the model checker on the two SMV models is an useful indicator on the trade-offs between the expressiveness and the efficiency of template semantics.

The goal of this paper is to describe the steps that are involved in mapping the A-7E SCR requirements to a specification in template semantics. Furthermore, we illustrate the translation from a template-semantics specification to an SMV model using Express and discuss our verification effort in comparison to Sreemani's results.

In Section 2, we begin with brief introductions to the SCR specification notation and template semantics. In Section 3, we provide a more detailed descriptions of the main problem, the verification of the A-7E requirements specification. Section 4 contains a summary of our solution to the problem, and Section 5 details the methods and the tools that were used to implement this solution. We discuss the comparison of the two sets of verification results and evaluate our success in Section 6. We conclude our work in Section 7, and provide an outline of future work in Section 8.

2 Background Information

2.1 The SCR Specification Notation

The SCR notation describes a system specification as a collection of mathematical functions that are represented as tables. A system in SCR consists of one or more state machines, called **mode classes**, and in turn, each of these mode classes contains one or more states, which are called **modes**. A variable in a SCR specification may be either **monitored** or **controlled**. The value of a monitored variable is set only by the environment, and that of a controlled variable is modified by the system as output to the environment.

Furthermore, a SCR specification may contain any number of **condition tables** or **event tables**. A condition table is a tabular representation of case-based assignments to variables; a case is determined by a condition on the current variables values. An entry in an event table modifies the value of the current mode for a single mode class, depending on an event and optional enabling conditions. An event is in form `@X(cond)`, where `X` may be one of `T` or `F`. `@T(cond)` means that the condition `cond` currently evaluates to false but is becoming true in the next step. The syntax for optional enabling conditions is in form `WHEN[enable_cond]`. An assignment to the mode in an event table entry can occur if and only if both the event and `enable_cond` for that entry evaluate to true.

2.2 Template Semantics

Template semantics is a parameterized approach to defining model-based notations. A model in template semantics consists of one or more hierarchical transition systems (HTSs). An HTS contains a set of hierarchical control states, a set of transitions between the states, a set of events, and a set of variables. Each transition has the form,

$$< src, trig\_ev, cond, act, dest, prty >$$

where *src* and *dest* are the transition's source and destination states, respectively; *trig_ev* is zero or more triggering events; *cond* is a predicate over variables; *act* is zero or more actions that assign new values to some variables and generate events; and *prty* is the priority of the transition.

The transition semantics of an HTS is defined as a **snapshot relation**. A snapshot is an observable point in an HTS execution, containing the sets of current states, current variable values, current internal events, and current outputs that are communicated to other concurrent HTSs. A **step** in template semantics corresponds to the transition of an HTS from one snapshot to another. A **micro-step** is the execution of exactly one transition by an HTS. On contrary, a **macro-step** is a sequence of micro-steps that begins with new input from the environment and terminates when no more transition is enabled. When the system reaches the end of a macro-step, we refer to the system as being **stable**.

When a model in template semantics contains two or more HTSs, they are composed into a hierarchical binary tree with each leaf node representing a single HTS, and each non-leaf node representing a composition operator. Currently, Metro supports the following 7 composition operators for the execution semantics of concurrent HTSs: parallel, interleaving, environmental synchronization, rendezvous synchronization, sequence, choice, and interrupt. A more detailed description of the composition operators can be found in [8].

3 Problem Description

The software requirements for the A-7E aircraft were written in the SCR notation by researchers at the U.S. Naval Research Laboratory in 1978. The version

of the requirements document discussed in this paper dates back to 1988 [1], and is equivalent to the version that was used for Sreemani's work. The SCR specification consists of three mode classes with an event table for each of them:

- Alignment/Navigation/Test mode class
- Navigation Update mode class
- Weapon Delivery mode class

A more detailed presentation of the mode classes can be found in [10]. The specification contains 84 variables and 703 possible transitions in the three event tables. Each transition depends on the current mode value and the current variable values. Since the specification does not have any condition tables, we assume that all of the variables are modified by the environment.

Sreemani identified five properties from the A-7E requirements document and verified them using model checking. These properties define the allowable combinations of three mode values in the system at a time. For the purpose of comparing our verification effort to Sreemani's, we have chosen to verify the same set of properties using the model checking capabilities in Cadence SMV.

4 Overview of Solution

In our approach to translating the SCR specification into template semantics, we map each mode class to an HTS with a single control state. An entry in the SCR event table for the mode class corresponds to a self-looping transition from this control state. The event in a SCR table entry maps to the triggering event in a state transition, and the entry's `WHEN` conditions correspond to the enabling conditions in the transition. Since a table in a SCR specification is complete by definition, all state transitions in an HTS have the explicit priority of 0, with an exception for an idle transition, whose triggering event is empty, whose condition is the truth value `T`, and whose explicit priority is 1. The source state and the destination state in an HTS's transitions are the same since all transitions are self-looping. Finally, an action in a transition assigns a new value to variable `Mode`, which represents the value of the current mode for the mode class that is represented by an HTS. For example, the following is an excerpt from the event table for the Navigation Update mode class:

```
MODECLASS   NavigationUpdate
  MODE   RadarUpdate
  ...
  MapUpdate @T(update_flyover) WHEN [station_selected]
  ...
```

This table entry represents the change of the mode from `RadarUpdate` to `MapUpdate` when the variable `update_flyover` is becoming true in the next step and `station_selected` evaluates to true at the beginning of the assignment. This table entry translates to the following state transition in the Navigation

Update HTS:

```
<NavigationUpdate, @T(update_flyover), Mode = RadarUpdate &
 station_selected, [Mode := MapUpdate], NavigationUpdate, 0>
```

where `NavigationUpdate` represents the sole control state in the HTS.

The composition operator for the SCR notation is functional composition; when there are more than one HTSs with an enabled transition, only one of them can execute at a micro-step. The order of the execution of the HTSs depends on variable dependencies among the event tables, and can be calculated offline using def-use analysis. Currently, the Metro toolkit does not support functional composition. However, in a previous case study on Metro, researchers have used an extra variable, called `order`, to indicate which one of the HTSs may execute at the current micro-step. An expression involving `order` is conjuncted with enabling conditions in each transition. At the end of a micro-step, the value of `order` is updated so that another HTS can execute in the next micro-step. The sequence of updates for `order` depends on the pre-determined order of functional composition among the event tables.

The A-7E model contains only event tables and thus, all variables are modified only by the environment. In this case, there exist no variable dependencies between the tables, and the order of the execution of the tables can be assigned non-deterministically. We have modelled the composition operator of the A-7E model in Metro using a modified version of interleaving composition, with the variable `order`, to ensure that only one table execute at a micro-step, and that all tables execute exactly once during a macro-step. Furthermore, since SCR tables are complete by definition, at least one transition is enabled in each table at all times. In template semantics, the system reaches the end of a macro-step only when no transition is enabled. We model a SCR snapshot instance without enabled transitions by setting `order` to a value that does not correspond to any of the HTSs' execution priorities. At the beginning of a macro-step, `order` is reset so that the HTS at the top of the ordering can execute.

5 Methods and Tools Used

5.1 SCR2HOL : Translator from SCR to Template Semantics

The first step in our A-7E case study is to translate the A-7E SCR specification into a template-semantics specification in Metro. The input language of Metro is the higher-order logic used in the HOL theorem prover [5], which was constructed on top of Moscow ML. In order to automate the process of the translation from SCR to HOL, we have implemented a tool called SCR2HOL. The input parser used in SCR2HOL is largely based on Sreemani's translator to SMV since both of the programs process the same format of SCR specifications.

For most parts, translation from SCR to template semantics is purely syntactical. SCR2HOL directly maps each table to a single HTS and table entries

to corresponding state transitions, as described in Section 4. The user needs to specify only the order in which the tables in the input SCR specification should execute after they are mapped to HTSs; this order can also be determined automatically using dataflow analysis, but SCR2HOL does not yet support this feature. When the translation is complete, SCR2HOL produces a ML file that can be directly inserted into the Metro toolkit for translation to SMV or further analysis in HOL.

5.2 Express : Translator from Template Semantics to SMV

The second step in our case study is to translate the A-7E specification in template semantics to an equivalent SMV model. Metro Express, written by Lu, is a translator that takes a template-semantics model as an input and produces an SMV file for formal analysis in tools such as Cadence SMV and NuSMV[3]. Besides the input specification, Express also requires the user to specify a set of template parameters that describe the characteristics of the original notation.

One problem that we encountered while using Express was that prior to this case study, Express had not supported triggering events of type `@T(cond)` or `@F(cond)`. We have chosen to abstract each of these SCR events to a boolean variable and assign a value to it at the end of a macro-step, depending on the current value of `cond` and the new input from the environment. For example, the following SMV code illustrates the definition of the variable `NT_Doppler_up`, which corresponds to the SCR event `@T(Doppler_up)`:

```
DEFINE
  NT_Doppler_up := case
    stable : ss.AV.Doppler_up = 0 & I.var.Dopper_up = 1;
    1 : ss.AV.NT_Doppler_up;
  esac;
```

where `ss.AV` indicates the variable values at the current snapshot, and `I.var` refers to a new set of values for the environment variables. Any condition inside `@T` or `@F` must consist of only monitored variables. When `stable` evaluates to true, indicating that the system has reached the end of a macro-step, `NT_Doppler_up` is assigned `T` if and only if `Doppler_up` is currently false but is becoming true due to an update from the environment. If the system has not yet reached the end of a micro-step, we do not modify the value of `Doppler_up`.

Table 1 shows the comparison between the measurements of Sreemani's SMV model and the model that was generated by Express. The measurements in the table are meant to provide only a rough estimate of the complexity of the models. More specifically, we believe that the actual complexity of Sreemani's model is greater than what is represented by the number of lines due to the fact that this model contains large `TRANS` statements that cannot be further broken down into smaller `TRANS` statements. The Express model contains nearly 4 times as many variables as Sreemani's model does. There are a number of reasons that account for the increased number of variables in the Express model. First of all,

during translation, Express creates a duplicate of each variable that is designated as an environmental variable (*i.e.,* a monitored variable in SCR). Since all of the variables in the A-7E model are of type *monitored*, the SMV model generated from Express will contain at least twice as many variables as the original SCR specification. Secondly, for each of `@T` or `@F` event in the SCR specification, Metro outputs a boolean variable that represents the abstraction of the event. Lastly, depending on template-semantics parameters for a particular notation, an Express-generated model contains other auxiliary variables that are used to model snapshot elements of an HTS.

Measurement	Sreemani's Model	Express Model
File size (byte)	258K	617K
Number of lines	1246	16607
Number of variables	65	231
Number of `DEFINE` macros	0	2450

Table 1: Measurements of Sreemani's SMV model and Express SMV model

Researchers at the University of Waterloo have come up with various suggestions for reducing the number of variables in an Express-generated SMV model [4]. One of the suggestions that is applicable to the A-7E model is representing variable `tran` in each `execute` module using a `DEFINE` macro. Since our model has 703 possible transitions in total, we expect that this optimization technique significantly improve the verification performance; the effect of this optimization is discussed in Section 6. Other optimizations, such as removing all of unnecessary duplicate variables in the SMV model, should be pursued as future work on this case study.

6 Verification Effort

6.1 Properties

For this case study, we have chosen to verify the five properties that Sreemani identified in her thesis [10]. It is important to note that the Express model contains two sets of snapshot elements that can be used to specify a property: `pss` and `iss`. `pss` refers to a snapshot at the beginning of a micro-step, and `iss` represents a point in a macro-step after which elements of `pss` are modified by the new input from the environment, `I`. Each instance of `pss` has corresponding `iss`, but the modifications of variables take place if and only if the system is stable. Therefore, `iss` should be used to specify properties at the macro-step level, while `pss` may be used to describe system behaviours that occur at the micro-step level.

For example, we specify the false property that the Weapon Delivery mode class will take on the mode values `AA_Guns` and `Manrip` at the same point in future (Property 4 in Sreemani's thesis) as follows:

```
EF(iss.AV.WeaponDelivery=AA_Guns) &
EF(iss.AV.WeaponDelivery=Manrip) &
EF(iss.AV.WeaponDelivery=AA_Guns & iss.AV.WeaponDelivery=Manrip)
```

Due to the space limit on the paper, we are not able to discuss the four other properties in this section. A detailed description of all the properties can be found in Sreemani's thesis.

6.2 Evaluation

Originally, Sreemani verified these properties using CMU SMV. Since our choice of the model checker for this case study is Cadence SMV, we repeated the verification of Sreemani's model using the latter version of SMV. We performed all of our experiments on a Linux machine with 4GB of RAM and 7GB of swap space. Table 2 illustrates the comparison of the verification times and the number of allocated BDD nodes between Sreemani's model and Express model as well as the optimized Express model where variable `tran` has been replaced with a macro. "X" indicates that no data is available because Cadence SMV terminated with a segmentation fault or ran out of memory before it could complete the verification.

Property	Sreemani's Model		Express Model		Optimized Express Model	
	Time(sec)	BDD Nodes	Time(sec)	BDD Nodes	Time(sec)	BDD Nodes
Property 1	0.76	41397	335.88	13516548	316.25	13516448
Property 2	1.08	90617	X	X	263.61	13516448
Property 3	1.10	100526	X	X	307.78	13516448
Property 4	3.29	168340	X	X	261.78	13516448
Property 5	1.06	89432	X	X	308.56	13516448

Table 2: Verification performances of the three different SMV models

The verification results show significant differences between the performances of Sreemani's model and the two models that were generated by Express. For all verification runs that completed, the SMV models from Express consistently perform worse than the Sreemani's model, in terms of both the verification times and the number of BDDs that were allocated during the verification. This result confirms our expectation for the loss of efficiency as a trade-off for the generality of semantics.

The optimization technique of replacing `tran` in `execute` modules with a macro seems to have a significant effect on the outcome of model checking. In the experimental runs for the last four properties, Cadence SMV failed to complete the verification, likely because it was unable to handle the complexity of the Express model. However, the model checker was able to verify the same set of properties in the optimized model. The drastic improvement in the performance is not surprising if we consider the fact that the `NavAlignTest` HTS contains 302 different transitions, `NavigationUpdate` 79 transitions, and `WeaponDelivery`

322 transitions. Using the equation in [4], the potential saving factor of this optimization can be calculated as $(302 + 1)(79 + 1)(322 + 1) = 7829520$.

As an experiment, we verified the properties after turning on the optimization option "Variable order sifting", which attempts to find an improved variable ordering in Cadence SMV. Surprisingly, the verification took significantly longer than it did without the optimization for both Sreemani's model and the Express models. We believe that the original ordering that is provided in Sreemani's model is the optimal one, and that any attempt to improve this ordering results in negative effects on the model checking performance.

7 Conclusion

In this paper, we presented the steps that were involved in mapping a SCR specification into a template-semantics specification. We showed that the template semantics is expressive enough to capture most aspects of the SCR notation, with an exception being the functional operator for the composition of tables. We also illustrated how Metro Express can be used to automatically generate an SMV model and verify the properties using model checkers such as Cadence SMV.

To demonstrate these steps with a concrete example, we performed a case study on the requirements for the A-7E aircraft control software. We successfully translated the SCR requirements into a template-semantics specification and eventually, into a SMV model. We verified the properties using Cadence SMV and compared the verification results to Sreemani's work. As expected, the SMV model that was generated from Express did not perform as competitively as Sreemani's model, which did not make use of template semantics. We believe that the relatively low performance of Express is inevitable due to a trade-off between the efficiency and the generality of template semantics; however, we also believe that we can significantly improve the performance of Express by exploring different optimization techniques.

8 Future Work

There are a number of potential optimization techniques that can be used to reduce the complexity of an Express model. Most of these techniques involve reducing the number of variables in the model by removing duplicates or replacing variables with macros. Another possibility is to take advantage of the characteristics of a particular model in order to reduce the model size. For example, since each HTS in the A-7E specification contains only one control state, it may be possible to eliminate the snapshot element `CS` from the model. In addition, the template-semantics representation of the A-7E specification does not have any events, so we can remove all SMV modules and variables that are related to events. We plan to investigate these optimization techniques for further improvement of Express.

References

1. Alspaugh T., Faulk S., Britton K., Parker R., Parnas D., Shore J. Software Requirements for the A-7E Aircraft. Technical report, Naval Research Laboratory, 1988.
2. Atlee, J.M., Day, N.A., Niu J., Fung D., Kang, E., Lu Y., Wong L. Metro: An analysis toolkit for template semantics. Technical Report 2006-34, David R. Cheriton School of Computer Science, September 2006.
3. Cimatti A., Clarke E. M., Giunchiglia F., Roveri M. NuSMV: A new symbolic model checker. In *Int. Journal on Soft. Tools for Technology Transfer*, pages 410-425. 2000.
4. Esmaeilsabzli S., Wong L., Day N. A. An evaluation of Metro Express. Technical Report 2005-20, David R. Cheriton School of Computer Science, December, 2005.
5. Gordon M., Melham T., editors. *Introduction to HOL.* Cambridge University Press, 1993.
6. Heitmeyer C. L., Jeffords R.D. The SCR tabular notation: A formal validation. Technical Report NLR/MR/5546-03-8678, Naval Research Laboratory, 2003.
7. Lu Y., Atlee, J.M., Day, N.A., Niu J. Mapping template semantics to SMV. In *ASE*, pages 320-325. IEEE Computer Society, 2004.
8. Niu J. *Template Semantics: A Parameterized Approach to Semantics-Based Model Compilation.* PhD thesis, University of Waterloo, 2005.
9. Niu, J., Atlee, J.M., Day, N.A. Composable semantics for model-based notations. In *FSE*, pages 149-158. 2002.
10. Sreemani T. *Feasibility of Model Checking Software Requirements: A Case Study.* MMath thesis, University of Waterloo, 1996.