Protocol Conformance through Refinement Mappings in Cadence SMV*

Sara Van Langenhove

Abstract

This paper addresses the verification of protocol conformance between two types of state machines in model-driven software design: protocol state machines that enable specifying allowed sequences of signals and behavioral state machines that are intended for implementation specification. The contribution of the present paper, is to provide a methodology, which is based on refinement mappings, to automatically verify on protocol conformance. It helps to turn UML into a powerful and interesting tool for the development of business critical software systems.

1 Introduction

If you are familiar with object-oriented methods, you will be aware of the concept of a class and an interface. A *class* represents an entity of a given system and provides a piece of system functionality, whereas an *interface* is a variation of a class in the sense that it only provides a definition of system functionality. Interface classes are used by classes that claim to implement them.

Nowadays, it is becoming a trend to use the Unified Modeling Language (UML) [9] for modeling a multitude of software systems, even in the early design phases of the software development process. The UML allows that classes use *behavioral state machines* to describe their piece of system functionality. Basically, state charts show the events and the conditions that trigger a transition from one state to another, together with the resulting behavior. The new version of the standard, UML 2.0, allows that interfaces use *protocol state machines* to focus only on allowable sequences of behavior invocation on a class, but, without having to show its behavior.

*Funded by Ghent University (BOF/GOA project B/03367/01 IV1) and the Prof. Dr. Wuytack Fund.

A problem, then, arises: given a protocol state machine and a behavioral statechart, that should be an implementation of it, how to verify that the implementation meets the specification? This problem is called *protocol conformance* verification.

Experience has shown that, even in the design phase, state machines are more and more used for the increasingly successful automatic verification through model checking [5]. In [10] we have introduced a UML based verification method to identify and to remove behavioral faults during the design phase and before entering the code phase. The proposed method makes way for an automatic translation of the state charts to the language of the model checker Cadence SMV (CaSMV) [2] and helps to turn UML into a powerful and interesting tool for the development of business critical systems. The challenge is now to augment the verification method of [10] with a method that automatically decides on protocol conformance in the design phase.

In this survey we first formalize both types of state machines. Thereafter, we explain what protocol conformance really means and we use refinement mappings (or simulations) to formalize the concept. Finally, a methodology to prove the conformance relation is set up, using the existing model checker Cadence SMV [2].

2 Two kinds of State Machines

Behavior in UML 2.0 is defined as a specification of how its context classifier changes state over time [9]. In addition to expressing the behavior of a part of a system, state machines can also be used to express usage protocols of a part of a system. Based on this, UML 2.0 differentiates between two kinds of state machines: *behavioral state machines* and *protocol state machines*.

2.1 Behavioral State Machines ($\mathcal{BSM}$s)

Behavioral state machines extend traditional finite automata by adding hierarchy, concurrency and communication. They are hierarchical automata associated to UML objects (a class instance) to model their behavior (Definition 1). Each $\mathcal{BSM}$ gives an abstract view of all the desired behaviors of an object in its lifecycle; a view that is concerned with what an object *must* do.

Definition 1. *A UML behavioral state machine is a 6-tuple* $\mathcal{BSM} = \langle\ \sigma, \delta, E, C, AC, s^0\ \rangle$ *where* σ *is a finite set of states,* E *is a finite set of events,* C *is a finite set of conditions,* AC *is a finite set of actions,* $\delta \subseteq \sigma \times E \times C \times 2^{AC} \times \sigma$ *is a finite set of transitions where* 2^{AC} *denotes the power set of* AC*, and where* s^0 *is the initial state.*

2.2 Protocol State Machines ($\mathcal{PSM}$s)

A protocol state machine is always defined in the context of a classifier (mostly an interface). In its simplest form, a $\mathcal{PSM}$ (Definition 2) is a state diagram in standard UML notation whose transitions are labeled by events (call event, signal

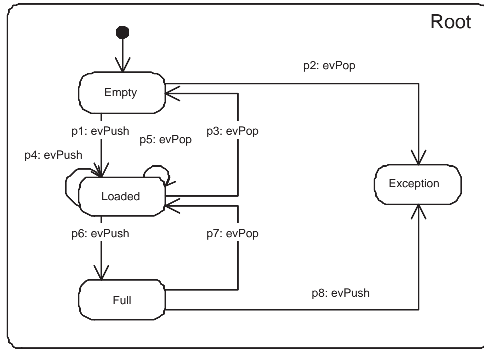

Figure 1: $\mathcal{BSM}$ of a Stack

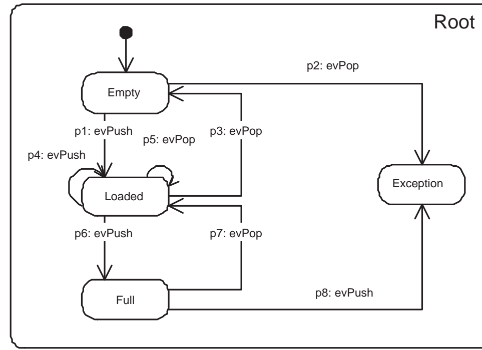

Figure 2: $\mathcal{BSM}$ of a Stack

event, time event, completion event) and do not have actions; i.e. refusing any behavioral implementation.

Definition 2. *A UML protocol state machine is a 6-tuple* $\mathcal{PSM} = \langle\ \sigma, \delta, E, PREC, POSTC, s^0\ \rangle$ *where* σ *is a finite set of states, E is a finite set of events, PREC is a finite set of preconditions, POSTC is a finite set of postconditions,* $\delta \subseteq \sigma \times PREC \times E \times POSTC \times \sigma$ *is a finite set of transitions, and where* s^0 *is the initial state.*

This way, a $\mathcal{PSM}$ captures the *triggering view* of an objects behavior. It presents the possible and the permitted transitions on the instances of its context classifier, by specifying the consummation order of the events and the states through which an object progresses during its life. A $\mathcal{PSM}$ defines *what the instances of its context classifier can do.*

3 Motivating Example: A Bounded Stack

An abstract data type (ADT), or interface, specifies a set of operations (or methods) and the semantics of the operations (what they do), but it does not specify the implementation of the operations. The stack as ADT is defined by specifying the operations (modelled as events *push, pop, isfull, isempty*) that can be performed on it. Using UML 2.0, the stack interface is represented by the protocol state machine given in Figure 1, showing the way how the stack is used in practice.

There are many uses for stacks: providing support for recursive procedure calls, searching structures, computation, and so on. Depending on the use, the methods of the stack can be implemented in many different ways, each leading to different stack classes (array based, list based, ...). A possible overall behavior of such a stack class is given by Figure 2. Note that the behavior of the stack is defined in a reactive manner. The transitions also contain some *guards* – a predicate expression associated with an event – which might change depending on how the stack is implemented. Note that the specified behavior considers the stack when it is almost empty or almost full.

Obviously, a protocol machine specifies what a behavioral machine is allowed to do at any given moment. At this point, the designer only has to worry about whether the implementation – the behavioral state machine – is correct in accord with the interface – the protocol state machine. This problem is known as protocol

conformance verification. It is the purpose of this paper to show how this can automatically be done during the early phases of system design.

4 Protocol Conformance

The UML Superstructure 2.0 Specification [9] explains that there are relationships between the classifier being the context of the specific $\mathcal{BSM}$ and the classifier being the context of the $\mathcal{PSM}$. Generally the former specializes or realizes the latter. In UML 2.0, protocol conformance means that every rule and constraint specified for the general $\mathcal{PSM}$ applies to the specific $\mathcal{BSM}$. This is augmented by specifying that the $\mathcal{PSM}$ can be redefined into the $\mathcal{BSM}$. Clearly, the behavioral view of an object is not independent of its triggering view. Intuitively, the ordered collection of stimuli received by an object's statechart must exist as a sequence of events in its corresponding protocol state machine. That is, every behavior of an object's statechart is also a behavior of its protocol statechart. If not, then obviously, the interface class is wrongly implemented. In order to verify the consistency between both state machines, we need a more formal definition of protocol conformance between a $\mathcal{PSM}$ and a $\mathcal{BSM}$ and adapt a method presented in [1]. This is where refinement mappings, as considered by Abadi and Lamport [4], will show their use.

4.1 Refinement Mappings (or Simulations)

The existence of a refinement mapping proves that a machine implements a given specification. You may know that refinement mappings are the functional cousin of Milner's *simulation relations*. Milner introduced them for the purpose of comparing programs [8]. The simulation guarantees that every behavior of a structure is also a behavior of its abstraction. However the abstraction might have behaviors that are not possible in the original structure. Technically, a simulation (or a refinement) is a function (or a mapping) R between the states of a low-level specification $\mathcal{S}_1$ and a high-level specification $\mathcal{S}_2$ that satisfies conditions [5] as

$$(s, v) \in R \wedge s \rightarrow_{\mathcal{S}_1} s' \Rightarrow \exists v' : v \rightarrow_{\mathcal{S}_2} v' \wedge (s', v') \in R$$

(If a low-level state s and a high-level state v are related, and $\mathcal{S}_1$ can make a transition from s to s', then there exists a matching transition in $\mathcal{S}_2$ from v to a state v' that is related to s'.) The existence of such a mapping implies that any behavior exhibited by $\mathcal{S}_1$ can also be exhibited by $\mathcal{S}_2$.

It can be of no suprise that a refinement mapping is useful to automatically decide on protocol conformance. Each $\mathcal{PSM}$ is equivalent to an abstract specification $\mathcal{PSM}_a$ while each $\mathcal{BSM}$ defines a concrete specification $\mathcal{BSM}_c$ of some classifier. Verifying on protocol conformance is now equal to proving that a low-level specification (i.e. $\mathcal{BSM}_c$) correctly implements a high-level specification (i.e. $\mathcal{PSM}_a$), using a refinement mapping between a $\mathcal{BSM}$ and a $\mathcal{PSM}$. A refinement mapping, denoted as R_f, from a $\mathcal{BSM}$ to a $\mathcal{PSM}$ is a function $f : \sigma_{\mathcal{BSM}} \rightarrow \sigma_{\mathcal{PSM}}$

that provides the following mappings:

1. state mapping: $\forall s \in \sigma_{\mathcal{BSM}} : f(s) \in \sigma_{\mathcal{PSM}}$
2. steps mapping: $\forall (s_i, s_j)_{e[g]/a} \in \delta_{\mathcal{BSM}} : (f(s_i), f(s_j))_{[pre]e[post]} \in \delta_{\mathcal{PSM}}$
3. behav. mapping: $xs \in Beh(\mathcal{BSM}) \Rightarrow f^w(xs) \in Beh(\mathcal{PSM})$

4.1.1 State Mapping

There can be no refinement mapping between both state machines unless the designer has specified a correspondence relation between the states of both state charts. We assume that a $\mathcal{PSM}$ presents the possible and permitted states through which an object progresses during its life i.e. a $\mathcal{BSM}$ cannot reside in states not present in its corresponding usage protocol. Summarized, the state mapping is given by a state transformer (Definition 3) that assumes a 1-to-1 correspondence between the states of $\mathcal{BSM}$ and the states of $\mathcal{PSM}$. Obviously, the initial state s^0_{BSM} corresponds to the initial state s^0_{PSM}.

Definition 3. *The set of states that an object may have during its life is fully defined in its $\mathcal{PSM}$:*

$$\forall s \in \sigma_{\mathcal{BSM}} : \exists! s \in \sigma_{\mathcal{PSM}} : f(s) = s'$$

4.1.2 Steps Mapping

Analogously, Definition 4 specifies the mapping of the steps. And each step is equal to a transition.

Definition 4. *A behavioral transition, with label* ***event[guard]/actions****, from state s_i to state s_j is legal iff the corresponding $\mathcal{PSM}$ defines a protocol transition from state s_i to state s_j with the label* ***[pre]event/[post]****.*

$$\forall (s_i, s_j)_{e[g]/a} \in \delta_{\mathcal{BSM}} : (f(s_i), f(s_j))_{[pre]e[post]} \in \delta_{\mathcal{PSM}}$$

This means that a behavior transition may exist iff there exists a protocol transition with the same source, target and triggering event.

Still, it is not required that every $\mathcal{PSM}$'s state/transition has a counterpart state/transition in its redefined $\mathcal{BSM}$ because a $\mathcal{PSM}$ specifies all the capabilities of a classifier, not all of which may be used in a particular system. That is, $\sigma_{\mathcal{BSM}} \subseteq \sigma_{\mathcal{PSM}}$ and $\delta_{\mathcal{BSM}} \subseteq \delta_{\mathcal{PSM}}$.

4.1.3 Behavioral Mapping

Protocol conformance means that every behavior of an object's statechart is also a behavior of its protocol statechart. Thus, Definition 5 implicitely defines the requested mapping of behaviors (= sets of all sequences of transitions) i.e. $xs \in Beh(\mathcal{BSM}) \Rightarrow f^w(xs) \in Beh(\mathcal{PSM})$.

Definition 5. *Let P be a $\mathcal{PSM}$ and B defined for a class c. B conforms to P with respect to the initial state s^0 if and only if whenever*

$$s^0 \rightarrow^* s' \overset{event[guard]/actions}{\rightarrow} s''$$

(that is, a transition is triggered from some state s' in B that is reachable from the initial state of B) we have a corresponding counterpart transition in P

$$s^0 \rightarrow^* s' \overset{[pre]event/[post]}{\rightarrow} s''$$

where both the pre and the post condition evaluate to true.

The refinement mapping R_f can now be used to prove whether the low-level specification correctly simulates the high-level one. If so, every execution trace of the BSM_c is allowed by the $\mathcal{PSM}_a$ meaning that the $\mathcal{BSM}_c$ implements (conforms) the $\mathcal{PSM}_a$. Obviously, there must exist some verification technique (and corresponding tool) that uses R_f to prove the protocol conformance during system design. Additionally, it is useful if the technique finds some interesting counter examples, helping the modeler to develop the design of her/his system.

5 Methodology

The refinement methodology enforces the verification process of the protocol conformance to proceed through phases:

Phase 1 The first phase automatically decides whether the set of states of the $\mathcal{BSM}_c$ is indeed a subset of the one of the corresponding $\mathcal{PSM}_a$. If not, the low-level specification contains states that are not present in the high-level one i.e. it is impossible to have protocol conformance between both machines. At this point, the developer is informed about her/his mistake(s). As an example, the set of states used to define the behavioral stack machine (Figure 2) is fully defined in its corresponding usage protocol (Figure 1).

Phase 2 The second phase verifies whether each behavioral transition has a corresponding counter protocol transition. If not, then the mapping of the transitions is not correctly followed, and the designer is again informed about her/his mistake(s). For example, it is easy to see that each behavioral transition in Figure 2 has a corresponding counter protocol transition in Figure 1.

Phase 3 The last phase proves the satisfaction of the behavioral mapping. This phase shows that every implementation behavior is allowed by its definition, specified in the $\mathcal{PSM}_a$. Of course, this shall be done by an exhaustive search respecting the run-to-completion step semantics of state charts. At this stage, in order to efficiently perform this proof, a sophisticated tool is needed. We propose the Cadence SMV model checker.

5.1 Behavioral Mapping Auxiliary

The behavioral model checker Cadence SMV (CaSMV) system [2] provides an approach that is geared to proving that an abstract model is implemented by some more detailed system model. The notion of correctness is defined in terms of refinement maps that relate signaling behavior at suitable points in the implementation with events occurring in the abstract model. The verification is based on a circular compositional rule that allows us to assume that one map (as a temporal property) holds true while verifying another map, and vice versa. The construct *layer* is used to provide the refinement maps. A layer is defined as a collection of abstract signal definitions. These are expressed as assignments in the same way the implementation is defined. Inside a layer transition relations are specified as well.

CaSMV gives us the opportunity to tie the verification of behavioral properties together with the protocol conformance proof. Both proofs happen through the model checking technique. Intuitively, to prove the protocol conformance, CaSMV executes the $\mathcal{BSM}_c$ following the execution semantics as close as possible and checks it against the information covered in the refinement mapping defined inside a layer. If no fault is found, we are allowed to say that the $\mathcal{BSM}_c$ complies with its corresponding $\mathcal{PSM}_a$. Otherwise, the model checker returns a useful counterexample, helping the user to develop the design.

Following Definition 5, the layer must contain the transition relation of the $\mathcal{PSM}_a$. The transition relation outside the layer is the one of the $\mathcal{BSM}_c$ resulting in the following structure:

```
-- high-level specification, triggering view
layer protocol : {
  -- PSM's transition relation

}
-- low-level specification, behavioral view
  -- BSM's transition relation
```

Doing so, the behavioral mapping is correctly defined and the model checker is capable of proving the conformance all by itself, as wanted. The main obstacle to face here is the construction of the transition relations. We will now show how to solve this problem using the `stack` as an example (Section 3).

5.1.1 Behavioral Transition Relation outside a Layer

In [10] we have defined a template structure in the CaSMV language [2] in order to be able to model check some behavioral properties of a system under development. Instantiating the template structure on the behavioral stack machine (Figure 2) results in the following CaSMV representation, which is simplified to the parts of main interest. The relation between the code and the graphical representation of the behavior is straightforward.

```
/* Specification of Behavioral View */

...;
t4:=in_Loaded & event_queue[0] = evPush & (size < (m-1));
t6:=in_Loaded & event_queue[0] = evPush & (size == (m-1));
...;
/* Initialization of Behavioral View */
init(st_root) := Empty;
/* Total Transition Relation of Behavioral View */
case {
  ...;
  progress_trigger & ~error: {  -- dispatch an event
     next(st_root) :=
       case {
         t1: Loaded;
         t2: Exception;
         t4: Loaded;
         t6: Full;
         t3: Empty;
         t5: Loaded;
         t7: Loaded;
         t8: Exception;
         default : st_root;
       };
     ...;
  };
  ...;
};
```

The code clearly illustrates that `init/next` statements specify the lifecycle depicted in the behavioral stack machine. The `init` statement defines the initial state of the machine. The `next` statements define parts of the total transition relation. Each transition is assigned a unique identifier and is used inside CaSMV to represent the enabling of the transitions. The activation of transitions is illustrated in Example 1.

Example 1. *Transition `t5` is allowed to fire and re-enters state `Loaded` when (1) the system is in the accepting state for this transition, i.e. `Loaded`, (2) the guarded event occurs (event is first of the queue and is dispatched), i.e. `evPush`, and (3) the guarded predicated evaluates to true, i.e. the stack is not yet almost full. However, if the stack is almost full, transition `t7` fires and the state machine reaches state `Full`.*

5.1.2 Protocol Transition Relation inside a Layer

The transition relation for a $\mathcal{PSM}$ must be specified in such a way that Definition 5 is correctly used within the verification process i.e. the execution of a behavioral transition should lead to the execution of some counter protocol transition. Let's examine the CaSMV representation of the protocol state machine (Figure 1) immediately.

```
/* Specification of Triggering View */

layer protocol: {
  ...
  p4 := in_Loaded & event_queue[0] = evPush;
  p6 := in_Loaded & event_queue[0] = evPush;
  ...
  /* Initialization of Triggering View */
  init(st_root) := Empty;
  /* Total Transition Relation of Triggering View */
  case {
    ...;
    progress_trigger & ~error: {
     next(st_root) :=
       case {
         p1 & t1: Loaded;
         p2 & t2: Exception;
         p4 & t4: Loaded;
         p6 & t6: Full;
         p3 & t3: Empty;
         p5 & t5: Loaded;
         p7 & t7: Loaded;
         p8 & t8: Exception;
         default : st_root;
       };
    };
   ...;
   };
 };
```

Here, `init/next` statements specify the refinement mapping. The `init` statement mentions the state, the behavioral state chart has to reside in at time `t = 0`. The `next` statements represents the allowable state changes at time `t + 1`. Each protocol transition also has a unique identifier, used again to define the condition under which a transition is able to take the state change.

Example 2. *It is allowed to reach state* `Full` *whenever* `t7` *and* `p7` *are enabled. Indeed, a behavioral transition,* `t7`*, is only allowed to execute whenever its abstract counter protocol transition,* `p7` *is executed as well. It is forbidden to reach state* `Full`*, if behavioral transition* `t7` *is enabled to execute, but the conditions of its abstract counter protocol transition are not fulfilled.*

Inside the layer, every protocol transition is connected to its behavioral counterpart. Executing a low-level transition is only possible whenever a corresponding high-level transition is executed as well, as specified by Definition 5. This explains the reason for *conjuctions* inside the layer, as illustrated by Example 2. If we leave out the conjunction and only use the abstract transitions to define the transition relation inside the layer, strange things can happen, i.e. if behavioral transition `t7` is enabled to execute, the abstraction transition `p5` shall be executed as well instead of abstract transition `p7`. Indeed, transition `p5` has the same activation condition as the one of

abstract transition p7 and is evaluated/taken first in the case statement. Leaving out the behavioral transitions inside the layer leads to a bad implementation of Definition 5.

5.1.3 The proof

The stack example shows a layer as a collection of a single abstract signal definition *st_root*. Such an abstract definition entails a verification task to show that every implementation behavior is allowed by this definition. Indeed, the model-checker verifies whether the low-level implementation of a signal is simultaneously consistent with its abstract definition for each possible behavior.

At time `t = 0` the model checker verifies whether the behavior machine is in the correct initial state. The stack machine clearly fulfills this requirement. Next, at time `t + 1`, each behavioral state change is verified against the state changes defined inside the layer. If at time `t + 1` the behavioral stack reaches a state that can never be reached by the layer, a counterexample is returned to the user. This is not the case for the stack example; the behavioral stack is protocol conformant with its protocol state chart.

6 Related Work

In the field of UML, the refinement concept for UML statecharts has been formalized in [7]. Here, refinement maps are defined in terms of configurations and simulations, which is slightly different from our refinement map definition. In [3] an extension of the Temporal Logic of Actions (TLA) is defined in order to identify adequate concepts of refinement for mobile UML state machines. Such an extension has the advantage to work with more complex refinement maps; our refinement maps are limited to the capabilities of CaSMV.

Important Usage Difference The verification of a system against a given property is done in two phases when using refinements. In a first phase, the coherence (=consistency) between the specification and the refinement is verified. If both are consistent with each other, the property is verified on the high-level model. Of course, the high-level model is usually much smaller than the low-level one. This means that a two-phase approach proves to be more efficient than a direct verification [6]. For our purposes, verifying a particular property does not benefit from this two-phase approach. This follows directly from the definition and the use of our refinement maps.

7 Conclusion

In this paper, we have used refinement mappings to define protocol conformance between two types of state machines during the design phase of software development. Based on this definition, we have shown a verification technique (and corresponding tool) to prove whether a $\mathcal{BSM}$, as low-level specification, correctly implements its $\mathcal{PSM}$, as high-level specification. Additionally, the methodology returns interesting

counterexamples, helping the modeler to develop the design of her/his system. We only carried out experiments on small examples, experiments on larger models must be done.

References

[1] Tenzer J. and Stevens P. Modelling Recursive Calls with UML State Diagrams. *Proceedings of Fundamental Approaches to Software Engineering, LNCS*, 2621, 2003.

[2] McMillan K. `http://embedded.eecs.berkeley.edu/Alumni/kenmcmil/smv/`.

[3] Alexander Knapp, Stephan Merz, Martin Wirsing, and Júlia Zappe. Specification and Refinement of Mobile Systems in MTLA and Mobile UML. *Theoretical Computer Science*, 2005. Special Issue AMAST 2004.

[4] Abadi M. and Lamport L. The Existence of Refinement Mappings. *Theor. Comput. Sci.*, 82(2):253–284, 1991.

[5] Clarke E. M., Grumberg O., and Peled D. A. *Model Checking*. The MIT Press, 2002. 0-262-03270-8.

[6] Kenneth L. McMillan. A Compositional Rule for Hardware Design Refinement. In *CAV*, pages 24–35, 1997.

[7] Sun Meng, Zhang Naixiao, and Luís Soares Barbosa. On Semantics and Refinement of UML Statecharts: A Coalgebraic View. In *SEFM*, pages 164–173, 2004.

[8] Robin Milner. An Algebraic Definition of Simulation between Programs. Technical report, Stanford, CA, USA, 1971.

[9] Object Management Group (OMG). Unified Modeling Language Specification. Available from `http://www.omg.org/uml/`.

[10] Van Langenhove S. and Hoogewijs A. Integrating Cadence SMV in the Verification of UML Software. In *Proceedings of the 8th Dutch Proof Tools Day*, July 2004.

Department of Pure Mathematics and Computer Algebra
Galglaan 2, B-9000 Gent
Ghent University, Belgium
Sara.VanLangenhove@UGent.be