v-Promela: A Visual, Object-Oriented Language for SPIN

Stefan Leue[*] and **Gerard Holzmann**
Lucent Technologies, Bell Laboratories
600-700 Mountain Ave.
Murray Hill, NJ 07974
[stefan|gerard]@research.bell-labs.com

August 20, 1998

Abstract

We describe the design of *v-Promela*, a graphical specification notation that reconciles Promela with state-of-the art graphical notations for the design of concurrent, object-oriented real-time systems, in particular with UML-RT and ROOM. Amongst other features, v-Promela introduces a graphical notation to describe both the behaviour and the structure of a system in a hierarchical fashion. We also sketch an implementation of the language constructs in Promela - in the ideal case v-Promela will be compiled into Promela and its existency will be completely transparent to the (X)SPIN model checker. We finally present an architecture for a graphical interface tool supporting v-Promela, and define the concepts of inheritance and polymorphism for v-Promela.

1 Introduction

To manage the complexity of the software engineering process the software life-cycle has been split up into a number of stages, and software process models associate well-defined activities with each one of these stages. While individual process models vary in detail, they largely agree on the existence of an analysis or requirements stage, followed by a design stage, an implementation and unit testing stage, an integration testing stage, and finally a maintenance stage. In [4] Davis cites various studies that analyze the impact that early life-cycle software engineering has on the overall cost of software projects. In particular, according to these studies removing a bug at the requirements stage can be 200 times less costly than removing it at the maintenance stage. This suggests that software engineering should pay particular attention to the early stages of the life-cycle in which software requirements are elicited, captured, and automatically analyzed for consistency and correctness.

We observe two trends in software engineering methodologies for concurrent real-time systems. On one side of the spectrum, there is a class of practical, object-oriented languages like ROOM [16], OMT [14] and SDL [10] that tend to provide a homogeneous, seamless coverage of the life-cycle down to the code generation, testing and even maintenance stages, but which mostly lack

[*]On leave from the University of Waterloo, Department of Electrical and Computer Engineering, Waterloo, Ontario N2L 3G1, Canada, sleue@swen.uwaterloo.ca.

the abstraction mechanisms necessary to build comprehensive requirements models. At the other end, there are languages like Promela [7] and SMV [2] which provide the abstraction mechanisms required to obtain mathematically tractable models that avail themselves to automated analysis, but fail to provide support for more than the early stages of the life cycle. Furthermore, it appears that Promela fails to support the modularization mechanims needed to cope with largely complex requirements specifications, while object-oriented techniques like the ones we mentioned above fail to deliver tractable models that would avail themselves to automated analysis.

It is the goal of our paper to demonstrate that the gap caused by this dichotomy can be narrowed, if not closed. We introduce a visual notation called *v-Promela*. The notation is based on some common cornerstones of the object-oriented modeling paradigm for real-time systems while remaining translatable into Promela, which accounts for its mathematical tractability. In the design of v-Promela we were guided by the following set of desiderata:

1. In keeping with a trend in software engineering notations, v-Promela should be a visual notation.

2. v-Promela should be capable of expressing both structure and behaviour, key elements in description techniques designed to capture software architecture at early life-cycle stages [1] and in many of the object-oriented methodologies mentioned above.

3. The notation should add abstraction and hierarchical layering mechanisms to Promela.

4. It should implement a "reasonable" subset of the object-oriented features, including the representation of concurrent objects, inter-object communication, inheritance and polymorphism.

5. All structural and behavioural concepts of v-Promela should be implementable in Promela and should not require any major changes to the Promela language and its current semantics.

6. A compilation of v-Promela into Promela allows us the use of the XSPIN model checker to validate v-Promela models. By insisting on a complete translatability into Promela we ensure that the presence of v-Promela can be completely transparent to SPIN.

7. A visual modelling notation hinges upon the availability of suitable CASE tool support. We sketch the design of the VOOPI toolset that is intended to provide for visual editing of v-Promela models, compile v-Promela models into Promela code, and allow v-Promela models to be simulated and validated using SPIN technology.

8. v-Promela should add support for compositional verification methods, e.g., assume-guarantee reasoning, as well as for design-by-contract like requirements capture by pre-/post-condition pairs and invariants, and finally it should facilitate the exploitation of symmetries that are typical for object-oriented systems.

Other model checking languages, like for instance SMV, could be used as a translation target as well. However, we feel that the support that Promela offers for inter process communication by either synchronous or bounded buffer asynchronous message passing communication makes it most suitable as a target language for v-Promela.

2 Related Work and Overview

Related Work. We decided to develop v-Promela as a notation that is largely consistent with what we expect to evolve into the *Unified Modeling Language for Real-Time Systems* (UML-RT) [17]. UML-RT is primarily based on two other techniques. First, it is an extension of the *Unified Modeling Language* (UML) [3] that was developed by Rational Corporation and has become adopted by the Object Management Group (OMG)[1] into its Object Management Architecture (OMA)[2]. More recently, Rational Corporation and ObjecTime Limited have teamed up to define UML-RT as an extension of UML (see [17]), based on the ROOM method [16]. It is expected that based on their broad support UML and its derivative methods will gain significant momentum in the software engineering community in the nearer future. The current state of the UML-RT language definition is rather preliminary and incomplete. As one of the authors put it, UML-RT is ROOM "*in UML clothes*". However, we expect the UML-RT notation to be expanded and eventually to become adopted by OMG as well [15]. If both ROOM and UML-RT do not provide sufficient guidance, we take the liberty of defining our own language features.

UML, ROOM and UML-RT have themselves been greatly influenced by extensive precursory work. Most notably, Harel's seminal work on Statecharts [6] has greatly influenced all these methods. Consequently, a subset of the Statecharts language can also be found in v-Promela. And, of course, there is a large body of work on Promela [7], including a web site that provides access to many Promela and XSPIN related resources[3].

A translation from Statecharts into Promela has been suggested in [13]. In this work, a fairly low-level semantics representation called 'extended hierarchical automata' is used to represent statecharts, and this representation is then translated into Promela. The work implements a much more faithful interpretation of Statecharts than we do. For instance, v-Promela does not use 'concurrent states' ('and' composition in Statecharts), instead it models concurrency through concurrent objects. As a consequence, many of the problems that [13] address are not relevant to v-Promela, and conversely we have to deal with issues that are not relevant to their work.

Overview. In Section 3 we define how structure can be described in v-Promela and illustrate its implementation in Promela. Next, in Section 4 we explain behaviour specification in v-Promela and discuss the corresponding Promela implementation. In Section 5 we show how inheritance and polymorphism can be fitted into v-Promela, and how these concepts influence implementation in Promela. Section 6 discusses the architecture of VOOPI, the CASE tool that is to support v-Promela. We conclude and define directions for future research in Section 7.

3 Structure in v-Promela

The structural description component of v-Promela, which we describe in this Section, follows largely the UML-RT notation as described in [17], and inherits many important ideas from ROOM [16].

[1] See `http://www.omg.org`.
[2] See `http://www.omg.org/news/pr97/umlpr.htm`.
[3] See `http://netlib.bell-labs.com/netlib/spin/whatispin.html`.

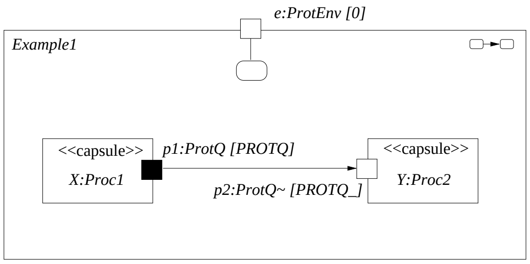

Figure 1: Composite capsule definition

Models and Packages. Models and Packages are organizational constructs in v-Promela that help organizing a large specification into different modules. The highest level structuring element is the model, which can be refined into an arbitrary number of packages. Both packages and models are not instantiable, and have no run-time semantics, they merely hold v-Promela specifications. A package may be refined by exactly *one* capsule, a concept we define in more detail in the next section. Capsule structures may themselves be refined by other capsules.

v-Promela models describe either *open* or *closed* systems. Closed systems do not have any interaction with any entity outside the model, i.e., there are no environment interactions. If environment behaviour is required inside a closed system, it has to be represented inside the model. Open systems are those that offer an interface to entities outside the model. In v-Promela, this is accomplished by defining interfaces (called 'public ports', see below) at the boundary of the outermost capsule in the nesting hierarchy. We will interpret these unbound ports differently in different modeling contexts:

- During *model simulation*, the v-Promela interface allows the user to tag ports of the model to be simulated as "injection points" that can be used to inject environment messages at run-time.
- The finite-stateness constraints of model checking-based *model validation* mandates closed system models. The v-Promela interface offers automatic randomized closing of an open v-Promela model.

Capsules. Capsules represent structural and behavioural abstraction in v-Promela. Capsules represent the concept of composite objects: a capsule can be decomposed into a set of contained capsules, in this case we call the capsule *composite*. The capsule decomposition defines a hierarchy, the leaf nodes of this hierarchy do not have an internal decomposition, and we call them *elementary* capsules. Capsules describe active objects which have an independent thread of control. Capsules represent dynamic aspects of the system. In v-Promela, there can be at most one state machine associated with a capsule to represent the dynamic aspects of the observable capsule behaviour.

A definition of a capsule in v-Promela entails a class definition, and major portions of the class definition are visualized graphically using an UML-RT type internal view collaboration diagram as in Figure 1. This Figure defines a capsule class `Example1` which is structurally refined as consisting of instances of capsule classes `Proc1` and `Proc2`. Sometimes it may be necessary to represent the capsule class definitions in isolation. Figure 2 represents an external view of each of the three class definitions contained in Figure 1, but naturally the information content of the isolated view is lower

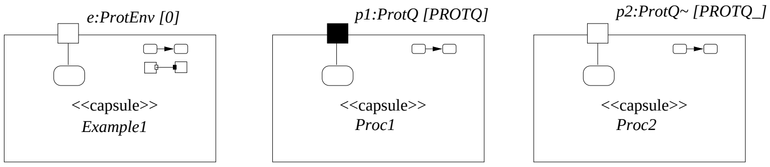

Figure 2: Internal view of capsules

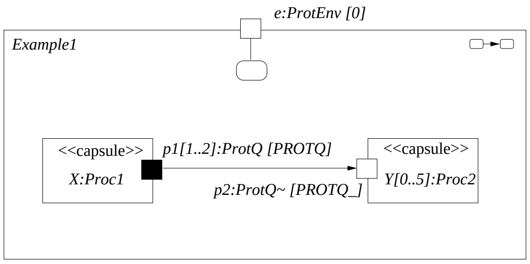

Figure 3: Composite capsule with replicated contained capsules and replicated ports

than that of the internal view collaboration diagram. Capsules are reminiscent of the *actor* object in ROOM. We use an iconic representations of state machine diagrams and collaboration diagrams placed in the upper right hand corner of a capsule definition. The icons in in capsule `Example1` in Figure 2 indicate that this capsule has a structural refinement and a state machine definition, respectively.

Replication, Optionality and Destruction of Capsule Instances. A varying number of capsule instances may exist any given point in time in a v-Promela model. The state machine of a capsule in v-Promela is capable of dynamically creating and destroying instances of contained capsules. When no instance replication factor is given, when an instance of a capsule is generated one instance of all its sub-capsules will also be generated. In other words, in the diagram of Figure 1 one instance of `Proc1` and `Proc2`, named `X` and `Y`, respectively, will be instantiated.

It is possible to deviate from this default by indicating a replication factor for a contained capsule. As shown in Figure 3, the replication factor `[0..5]` specifies that at instantiation time of capsule `Example`, no instance of `Proc2` will be generated. This expression also constrains the number of `Proc2` instances that `Example1` can generate to 5. In general, we use replication expressions of the format `[` *minimum-number* `..` *maximum-number* `]`. If the lower limit is 0, we say that the capsule is *optional*. The upper bound has to be an integer literal or an integer-type constant.

Capsule instances in v-Promela may also be destroyed at run-time. Destruction of a capsule instance is accomplished by a `destroy` statement. A capsule instance `Q` may destroy any contained non-replicated capsule instance `P` by the command `destroy P`. If `P` is replicated, the instantiation number of the replica to be destroyed is attached in squared brackets, e.g., `destroy P[3]` to destroy the 3rd replication instance of `P`. If the capsule instance to be destroyed is itself a composite capsule, then all contained capsule instances need to be destroyed as well. The `destroy` operation can only be applied in a hierarchy-downwards direction.

Data Classes. In v-Promela, data objects are instances of data classes. Data classes implement the basic Promela data types as well as any data structure that can be derived from the basic data types in Promela, with the exception of channels. v-Promela data classes build on the predefined Promela data types *bit*, *bool*, *byte*, *short* and *int*. In v-Promela, *one-dimensional arrays* can be defined using a tabular structure that defines a data class name, a dimension, a base type, and an initial value. In a similar fashion, structures can be built based on basic data classes and one-dimensional arrays. The scope of the data class definitions is global for the entire model.

Data Objects. Data objects can be instantiated from defined data classes. It arises the question, what scope a data object declaration should have, and what its visibility should be. ROOM and UML-RT are very restrictive with respect to visibility of data objects. In essence, inter-process communication is exclusively message passing, hence there is no shared variable communication. v-Promela is intended to be a modeling language much rather than a design language attached to a particular inter process communication paradigm, and hence we decided to be more permissive than ROOM and UML-RT in order to accommodate low-level modeling needs. Hence, we define the following visibility rules for data objects:

Private: The visibility of data objects declared private is confined to the scope of the capsule in which they are declared. Private data objects will not be visible from the scope of contained capsules. Private visibility is the default when no visibility attribute has been explicitly given.

Protected: A protected data object will be global in the name scope of the capsule in which it is declared. In other words, contained capsules will see and be able to modify protected variables declared in a directly or transitively containing capsule.

Public: A data object declared as public is visible to all other capsules of the model.

If more than one capsule use the same name for a protected or private variable, the innermost re-definition of that name will overlay the more global definition of that name. When referring to a visible object in a different name scope v-Promela uses path names to identify this object. The path name is composed of a sequence of capsule instance identifiers which, starting from the top most instance, describe a traversal of the capsule nesting hierarchy to reach the desired object. The name of the object will be appended, preceded by a double colon. If the name is to be found in a different package, the name of that package will be pre-pended to the path name, seperated by a colon.

Data objects may belong to replicated capsule instance, and every replica of a capsule instance hosts replicas of all data objects declared for this capsule type. When replicated capsule instances have non-private data objects an index notation will be used to refer to a particular instance of that capsule. Consider that the third instance of the public `round` data object is to be addressed, then this could be done by writing `Z.Y[3]::round`. Capsule instances may or may not exist at different points in the computation. Therefore, for every non-private replicated data object a global array variable named *hierarchy-navigation* `_procid` is kept that can be queried for the system process id of any replicated capsule instance. If this instance is not equal to zero, the queried instance of a data object does exist.

Messages. A message in v-Promela consists of two parts:

1. *Message identifier.* This is a name for the type of a message and an abstraction of its contents. Message identifiers are implicitly defined constants with global scope.

2. *Message body.* This is an *optional* part of the message which contains data and has the type of any valid data class or basic data type.

The following table defines a number of message types:

Message identifier	*Message body type* (optional)
`start`	
`request`	`request-msg`
`grant`	`grant-msg`
`denial`	`denial-msg`

The semantics of message reception in v-Promela is as in Promela for the case in which the format of the receive statement is *channel-name*`?`*mtype-name*`,`*list-of-variable-names*. I.e, the mtype name must match the message identifier, and the type of the variables that follow must match the type of the message body.

Protocols. Protocols define lists of message identifiers exchanged between instances of two capsules. A protocol has a name which allows one to refer to the message identifier lists. A protocol definition has global scope. The message identifier lists can be split up in identifiers for messages sent to another capsule instances (out-messages), and messages received from another capsule instance (in-messages). The capsule `Example1` in Figure 1 contains, for instance, the following protocol:

Protocol definition: `ProtQ`	
in-message identifiers	*out-message identifiers*
`grant`	`request`
`denial`	

Communication is typically symmetric, i.e., the in-messages as seen from one capsule are out-messages for another capsule, and vice versa. We adopt from UML-RT and ROOM the notion of the *conjugate* of a protocol which is the same as the non-conjugated protocol except that the in and out lists are inverted. We denote the conjugation of a protocol by appending a tilde at the end of the original protocol name.

Ports. The interfaces of capsules are called ports. They permit a capsule to interact with its environment through message passing. A port is an abstraction for the concepts of recipient/sender address, queueing discipline and queue capacity of the communication infrastructure that the system uses. Port references are used when sending or receiving messages. Ports are represented by either white or black filled boxes on the boundaries of capsules, c.f. the boxes labeled '`e:ProtEnv [0]`', '`e1:ProtQ [PROTQ]`' and '`e2:ProtQ~ [PROTQ_]`'. The string before the colon defines the name of a port. Note that there can be many instances of a port, depending on how many instances there are of the capsule that the port belongs to. The string after the colon and before the blank denotes the protocol type portion of the type of the port. The string in the squared brackets that follows denotes the capacity offers this port, i.e., the maximum number of messages that can at any point in time be sent to and not yet received from this port. A zero port capacity indicates synchronous communication along this port, non-zero capacity indicates buffered asynchronous communication with finite buffer capacity. Independent from the replication of the capsule to which a port belongs the port itself can have a replication factor attached to it.

In addition to the protocol type, two more attributes define the type of a port. First, a port can be of the base type, which is the case if the attached protocol is a non-conjugated protocol. This is denoted by a black, filled box at the boundary of a capsule. If the protocol of a port is conjugated, we say that this port plays the conjugated role and depict it by a white box. The second attribute is the distinction between end-ports, which pass messages to and from a behaviour describing state machine, and relay ports which pass messages to and from capsules. End-ports are depicted by an elongated, round-cornered rectangle which is connected to the port box by a line, c.f. the port `e:ProtEnv` in Figure 1.

Connectors. By connecting ports connectors represent the fact that messages can flow between capsules. As an example, consider the arrow connecting ports `p1:ProtQ` and `p2:ProtQ~` in Figure 1 and the line between the same ports in Figure 4. The presence of a connector between two ports entails that there must be some communication infrastructure between two capsules, although the particulars of this infrastructure are hidden. All that is visible from inside these capsules is the name and type of the respective ports. There are two types of connectors in v-Promela. An arrow represents unidirectional communication between the associated ports in the direction of the arrow, while a line indicates bi-directional communication. Two syntactic conditions constrain when a connector can be drawn between two ports. First, the ports myst be protocol type compatible (c.f., [16]), and second, any directly or indirectly connected pair of end ports must have different roles (i.e., one must be the base role, the other must be the conjugated role).

Buffers. For modeling purposes and in order to remain more consistent with basic Promela it might be desirable to allow active capsule instances to access buffers in a much more flexible fashion than it is possible using the standard v-Promela message passing mechanism through ports and connectors. We want to allow an arbitrary set of capsule instances to write objects into or to read from a buffer. To obtain this expressiveness we introduce the concept of a *buffer* into v-Promela. The concept is illustrated in Figure 5.

Buffers have a type, which is the type of the objects that they can host. Note that while ports can handle messages of different types, buffers are restricted to handling objects of one type. A buffer has a capacity attribute which determines the maximum number of objects it may hold at any given point in time. The access relationship between active capsules and buffers is described in terms of buffer access points (white or black filled dots) and directed buffer access paths which are represented by arrows. The latter describe buffer access rights in the obvious fashion depending the direction of the arrow.

Synchronizers. The concept of a *synchronizer* in v-Promela is derived from the concept of a buffer. The main difference between these concepts is that a synchronizer has no capacity, hence any communication that happens through a synchronizer involves a synchronous rendez-vous of two capsule instances. The semantics of read and write operations for a synchronizer can be derived

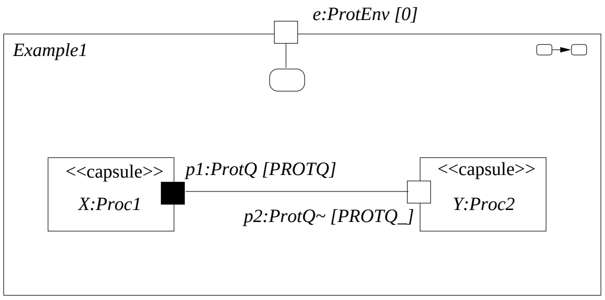

Figure 4: Composite capsule, bi-directional connector

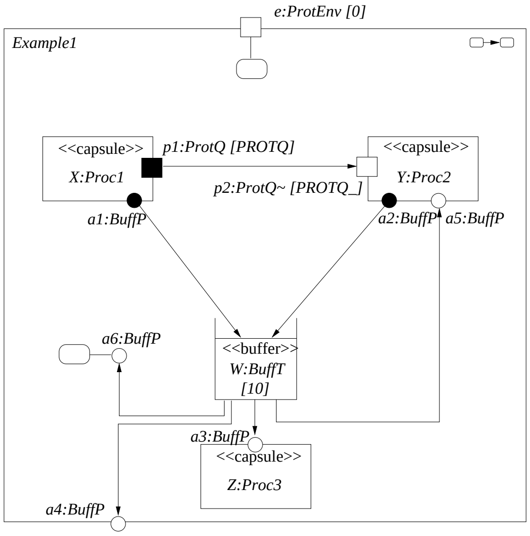

Figure 5: Composite capsule with contained buffer

from the semantics of rendez-vous channel communications in Promela. The description of the relationships of a synchronizer with state machines and other capsules is largely similar to that for buffers. One difference, though, is that a capsule which writes to a synchronizer may not at the same time read from it.

Implementation in Promela

We now discuss how structural concepts of v-Promela will be compiled into Promela code. First, capsule definitions correspond to Promela `proctypes`. Special attention needs to be paid to optional and replicated capsule instances, where more than one `proctype` may be incarnated. We introduce a counter variable which tracks the number of currently active instances of a particular capsule class. In the example of Figure 3 we use a byte type instance counter variable `Exampl1_Proc2_instnr` to count the number of instances of capsule class `Proc2`. The instance counter variable is used as an additional guard for the executability of a `run` statement in order to enforce the maximum replication constraint for a capsule class. Second, the instance counter will be passed as an argument to the object instance as a parameter named `_me`. This allows every object instance to know what its instance counter is. Third, we keep a global variable `Example1_Proc2_procid` which is defined as an array of length 5 and used to store the process id that is returned as a result of the `run` statement in the Promela code. This variable allows us to identify which process id has been associated with which instance of a replicated capsule. As an illustration, assume that a state machine within capsule class `Example1` was containing a `run Proc2` statement. This statement would be translated into the Promela code sequence

```
d_step { (Example1_Proc2_instnr < 5) -> _tpid = run Proc2(Example1_Proc2_instnr);
         Example1_Proc2_procid [ Example1_Proc2_instnr ] = _tpid;
         Example1_Proc2_instnr++ }
```

The declaration of the proctype corresponding to Proc2 is `proctype Proc2(short _me)`. Inside Proc2, access to a global data object like `round` would need to be translated into access to a corresponding field in an array. Accessing the own copy of `Example1_Proc2_round` is accomplished within Proc2 by indexing the global data structure:

```
Example1_Proc2_round [ _me ] = Example1_Proc2_round [ _me ] + 1;
```

The dynamic destruction of a process instance is implemented using the following scheme. Assume that proctype P corresponds to a capsule contained within a capsule represented by proctype Q. The containing capsule Q, which in the first place is responsible for instantiating proctype Q, will use a global guard variable `P__destroy` to terminated the life of P. This is accomplished by an `unless` construct at the highest level within the code structure of P, as in the following example:

```
proctype P() {  { ... P's code ...}
  unless { P__destroy -> d_step { P_procid [ P_instnr ] = 0; P_instnr--; R__destroy = 1; skip } }
}
proctype Q() { ...
  P__destroy = 1
  ...}
```

Data class definitions translate into Promela type declarations in the obvious way. Data object instantiation is translated into Promela variable declaration. For private data objects, local variable declarations are used. For protected and public data objects, global variable declarations are used in the Promela code. To disambiguate non-private data object names we use name prefixing in the Promela code. The prefix is composed based on the positioning of the object in the capsule hierarchy. Non-private replicated data objects are implemented as arrays. For instance, to access the public copy of a data object `round` in the 5th instance of capsule instance Y within capsule instance Z, we obtain the statement `Z_Y_round [5]` in the Promela code. Due to Promela constraints, non-private data objects of replicated capsules may not be arrays themselves.

Message identifiers are translated into `mtype` objects, and message bodies into Promela data types. The pairing of two ports by a connector results in one Promela channel for each receiving end of the connection. The names of the channels need to be disambiguated. In other words, if capsules C3 and C4 with bi-directionally connected ports p and q and a protocol P are contained within a capsule C2 which is contained within C1, then we will define the two channels `C1_C2_p_P` and `C1_C2_q_P_` to represent the connection between p and q. A v-Promela send to port p will be implemented as a Promela send to the channel at the receiving end, i.e., a send to `C1_C2_q_P_`. A v-Promela receive from port p is translated into a Promela receive from channel `C1_C2_p_P`.

Protocols can have messages the bodies of which are of different incompatible types. However, Promela channels accept messages that have identical base types. To solve this problem we define the channel type as consisting of two parts: the *message identifier* which it is of type `mtype`, and a second component which has structured type so that its components are made up from all the different message body types that any message traveling over this port can have. The message body type for some protocol `ProtQ` is defined as:

```
typedef ProtQ_msgtype {grant_msg grant; denial_msg denial; request_msg request
   };
```

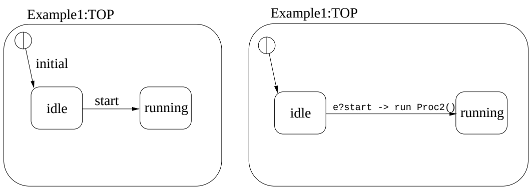

Figure 6: Two variants of representing the top level state in `Example1`

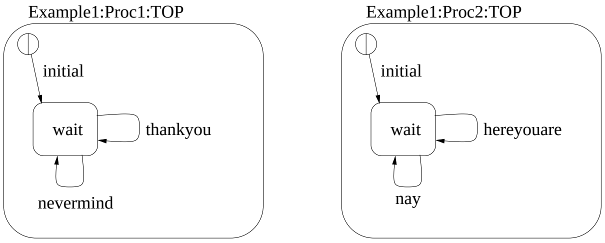

Figure 7: State machines for `Proc1` and `Proc2` – state label notation

A channel that is supposed to carry messages belonging to protocol `ProtQ` is defined as `chan Example1_p1_ProtQ = [PROTQ_] of { mtype, ProtQ_msgtype }`. Within `Proc1` we define a local variable `in_msgvar`that works as a message buffer of type `ProtQ_msgtype`. The v-Promela statement `p1?request` is translated into the Promela code `Example1_p1_ProtQ?ProtQ_request, p1_msgvar.request`.

v-Promela buffers and synchronizers are implemented as Promela channels, where synchronizers have a zero capacity.

4 Behaviour Specification in v-Promela

Both elementary and composite capsules in v-Promela may or may not have behaviour descriptions attached to them. We call a capsule which possesses a behaviour definition an *active* capsule, whereas one that does not possess behaviour is called a *passive* capsule. Behaviour in v-Promela is specified using hierarchical, extended finite state machines (HEFSMs), very much in line with many real-time systems engineering methods (e.g., ROOM, UML and UML-RT), bearing some similarity with Statecharts [6]. However, in v-Promela, control within each capsule is strictly sequential, but capsules are composed in a concurrent fashion.

State Diagrams. A *state* in v-Promela is an abstraction for a point of control within the computation of a capsule. We use state name labels of the form *capsule-name* : *state-name* to identify states and the capsules to which they belong. `TOP` is a special keyword denoting the top-level state within a capsule. Figures 6 and 7 present a finite state machine example related to the structure specification in Figure 1. Every state contained in a state diagram may be refined by another state diagram. We calle the resulting state machine hierarchical, an Example can be found in Figure 8. States can be *elementary* states that have no decomposition and *composite* states that have a state refinement. When we look into a state hierarchy and relate states within one hierarchy to each other, we often distinguish *sub*states and *super*states. Substates are either composite or elementary states that are part of the refinement of a superstate. Superstates are always composite. State refinement in v-Promela can be represented in various forms. Figure 8 shows an example of a hierarchical state machine in complete representationin which state hierarchy is expressed by graphical inclusion. For editing purposes it is convenient to have a partial representation for just one level of the hierarchy available, as in Figure 9. In this notation, a circle with an 'E' on the boundary of a composite state indicates a *state entry point* and acircle with an 'X' on the boundary

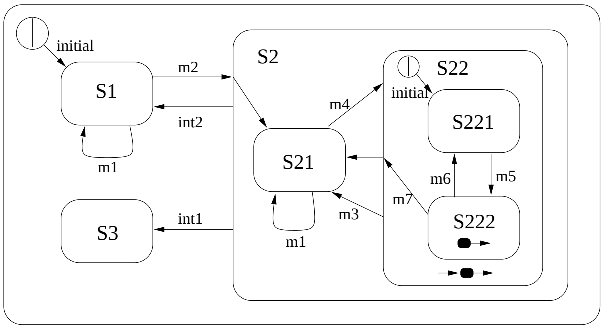

Figure 8: HFSM with return to history semantics and state code labels, complete representation

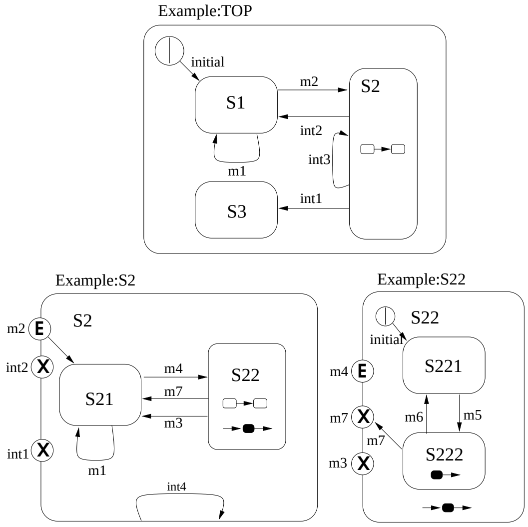

Figure 9: HFSM with exiting and non-exiting state transitions, partial representation

of a composite state represents a *state exit point*. The hierarchical state principle introduces group transitions. These occur when the system is in a lower-level state and a higher-level transition is executed. As an illustration, assume that the state machine in Figures 8 and 9 is in state *(S2, S22, S222)* and an event `int1` is happening, then the system will transition into state *(S3)*. Two policies are possible when a system returns to a previously visited composite state. First, *return to history* drives the system into the state of the composite state from which it was last pre-empted, or into the initial state if the composite state wasn't visited before. As an example, a return to `S22` through `m4` in Figures 8 and 9 would lead into `S222` if this state was last pre-empted. Alternatively, *return to explicit state* leads back to a contained state irrespective of where the composite state was left. Consider transition `m2`.

When an active capsule is instantiated, it's contained state machine will be driven into the initial state. An initial state is denoted by a circle with a vertical bar (c.f. the symbol in the upper left corner of state `Example1:TOP` in Figure 6). The system will also carry out an initial transition labeled `initial` as part of the instantiation process.

States will be identified by names, c.f. "`running`" in Figure 6. UML suggests that state names are mere labels, but that states neither need to have a name, nor that different states within one diagram need to have different names. Transitions describe the transition of the state machine from one state to a successor state. They are represented by states leaving one state and leading into another state.

Transition Enabling and Transition Code. Transition code in v-Promela consists of a set of sequential Promela statements that will be executed while a capsule is performing the transition from one state into a successor state, or while executing the initial transition into an initial state. Transition code can appear either as code associated with the occurrence of a state transition, or as state-entry and state-exit code (we refer to the latter two as 'state code'). In principle, with the exception of some restrictions that we will discuss below, any sequence of Promela expressions forms valid transition and state code in v-Promela. The description of behaviour of a v-Promela model relies entirely on sequences of possible system states. Hence, the model has to determine under which conditions the transition from one state into a successor state is possible. We call these conditions transition enabling conditions. We take advantage of the fact that Promela is a guarded command language - every statement in Promela, be it basic or not, has an either explicitly or implicitly defined enabling condition. It follows that labeling a state transition with *any* sort of Promela statement either implicitly or explicitly defines an enabling condition for this transition. There are two ways of associating transition code with transitions. First, as shown on the left side in Figure 6 and in Figure 7, transitions can be labeled with a symbolic name, and the corresponding transition code is edited in a seperate table. Table 1 gives a transition code table for the `Example1:Proc1:TOP` state in Figure 7. Alternatively, the transition code can be directly attached to the transition label, as shown on the right side of Figure 6. The first solution scales up nicely to models with complex transition code, the second facilitates comprehension of the v-Promela model when the transition code is relatively short.

Transition label	*Transition enabling and code*
initial	`p1!request`
thankyou	`p1?grant -> printf("Thank you!\n"); p1!request`
nevermind	`p1?denial -> printf("Never mind ...\n");p1!request`

Table 1: **Example1:Proc1:TOP**

Some real-time systems modeling languages are more restrictive in their choice of a syntactic form for transition enabling condition and transition code. A typical transition specification format in notations like UML or OMT looks as follows:

event-signature `[` *guard* `]` *action-expression* `^` *send-clause*

To remain consistent with these notations, v-Promela offers an alternative table format for transition code, as shown in Table 2.

Transition label	*event*	*guard*	*actions*	*send-action*
initial				`p1!ProtQ_request`
thankyou	`p1?grant`		`printf("Thank you!\n")`	`p1!request`
nevermind	`p1?denial`		`printf("Never mind ...\n")`	`p1!ProtQ_request`

Table 2: **Example1:Proc1:TOP**, condition code table in UML style

UML does not possess an unambiguously defined model-theoretic underpinning for terms like *event* and *action*, therefore our interpretation is just one possible way of understanding UML. We hope that this illustrates that a formal interpretation can easily be given for these UML constructs UML into a semantically largely well-defined language like Promela. However, this requires commitment to a model theoretic, well-defined meaning of these terms. For a more extensive discussion of this issue see [8].

Promela Code within v-Promela Transitions. The choice of a suitable subset of the Promela language that can appear as part of a v-Promela transition is determined by a compromise between sufficient expressiveness and an avoidance of clashes with other v-Promela constructs. The following discusses admissible constructs and required properties of transition and state code. Note that the with the exception of the `destroy` statement the described language is a subset of Promela. All references to variables must be to *data objects* declared within the scope of the state machine that the transition belongs to, i.e., the containing capsule, or to any data object that is visible within the scope of the containing capsule. *Meta terms* that are not allowed include `ltl` and `inline`. Also, *declarators* are not allowed inside transition code. In principle, all *control flow* constructs shall be allowed with the exception of `label` definitions, `goto`s, and `break` statements. The rationale behind this limitation is that to express branching control flow, control states in the v-Promela model should be used. Furthermore, `d_step` definitions will not be allowed because every piece of transition of state code will implicitly be made a `d_step` declaration. The `unless` statement of Promela is not allowed - exceptions with priorities can be expressed through higher-level group transitions. All Promela *basic statements* are all allowed in v-Promela's transition code. However, there are some exceptions with respect to keywords that may not appear in v-Promela transition code while being allowed in Promela expressions. These include `enabled`, `name`, `np_` and `pc_val` which only have a meaning within Promela `never` claims. Of the *predefined* expressions, only those with meaning inside v-Promela transition code will be allowed. The set of predefined expressions in v-Promela is therefore restricted to `_`, `_pid`, `cond_expr`, `else`, `empty`, `full`, `len`, `nempty`, `nfull`, `poll`, `run` and `timeout`. We add the predefined expression `destroy` which represents the dynamic destruction of capsule instances. `destroy` takes a capsule instance identifier as an argument.

Special attention has to be paid to the correct syntactic form of send and receive statements. Instead of referring to a channel name like in Promela, send and receive statements refer to *port* names in order to specify a potential source or sink of the communication. The set of ports on the boundary of a capsule defines its message passing interfaces. For example, in order to send a message of type `m` from capsule instance `X` in Figure 1 to capsule instance `Y` a `p1?!m` statement would be used. Variants of the sending and receiving primitives denoted by `!!` and `??` are also allowed, for more see the Promela documentation.

Semantic Assumptions. Concurrency in v-Promela is interpreted based on an interleaving model. Furthermore, v-Promela is based on the assumption of a *run-to-completion* semantics. I.e., transition code is executed atomically and will not be interleaved by events in other capsules. v-Promela transition code is required to be largely '`d_step`-safe', i.e., it has to obey the constraints applying to Promela statement sequences appearing within the body of a `d_step` statement. This means, amongst other things, that the statements need to be *deterministic*, *non-blocking* and *terminating*. As we will see later, this constraint is motivated by the way we implement transition and state code, namely by enclosing every transition and state code sequence with a `d_step` statement.

Entry and Exit Code. v-Promela offers the definition of *entry* and *exit* code sequences for states. When a distinction between entry and exit is not necessary, we sometimes refer simply to *state* code. Entry and exit code sequences will be executed whenever a state is entered or exited, respectively. While being a syntactic shorthand for transition code, entry and exit code bear potential for nice abstractions and may greatly increase readability of a specification. Entry and exit code is subject to same syntactic rules as transition code. In VOOPI, state code will be edited in seperate code tables.

Implementation in Promela

v-Promela states translated into Promela *labels*, and transitions are implemented as Promela *goto* statements. Transition code will be executed prior to the `goto`. Group transitions are implemented using the Promela `unless` construct. To implement run to completion semantics transition code is embraced by a `d_step` statement. Some statements, like the `run` statement, may not appear inside `d_steps`, therefore we occasionally have to use a combination of `d_step` and `atomic`.

To implement the return-to-history semantics we use a byte history variable for every non-elementary state for which there exists the possibility of a pre-emptive exit. A value 0 of this variable indicates that the state was either never visited before or that it was left properly, a value k for $k = 1 \ldots n$ indicates that the state was pre-empted in state k out of a total of n substates. When jumping into a non-elementary state that has a substate history variable we have to first check the value of that variable and perform a jump to the appropriate substate label.

Entry and exit code in a hierarchical state machines is executed whenever a simple or a group transition occurs that involves states for which entry or exit code has been defined. A group transition from a composite state into another state will execute the exit code for all substates and cause a group transition for all composite substates. In case the target state of a group transition is itself a composite state, the target state will return to its proper state according to the state re-entry rules. This means that that group transitions may entail a series of exit and entry code executions. Entry and exit code is implemented as non-parameterized *inline* procedures in Promela. As an example, assume the exit code definition for state `S222` in the above example is as follows:

```
inline HFSM_S222_exit () {
printf("Exit-code of S222\n");
}
```

Putting all the concepts together, the following example illustrates the implementation of a portion of the hierarchical state machine defined in Figures 8 and 9.

```
S22: atomic { HFSM_S22_entry();
   if
   :: (S22_last == 0) -> S22_last = 1; goto S221
   :: (S22_last == 1) -> S22_last = 1; goto S221
   :: (S22_last == 2) -> S22_last = 2; goto S222
   fi; }
   S221: { atomic {
      if
      :: d_step {in?m5 -> S22_last = 2 }; goto S222
      fi;}
   S222:   atomic {
      if
      :: d_step {in?m6 -> HFSM_S222_exit(); S22_last = 1 }; goto S221
      :: d_step {in?m7 -> HFSM_S22_exit(); S22_last = 0 }; goto S21
      fi;}
   } unless { atomic {
      if
```

```
        :: d_step {in?m3 -> HFSM_S22_exit()}; goto S21
        fi } }
```

Group transitions may cause exit code executions at lower levels. To ensure that a group transition at higher level leads to the execution of lower level code we need to invoke the lower level exit code procedures in the higher level ones. For example, the following exit procedure will be defined for S22:

```
inline HFSM_S22_exit () {
if
   :: S22_last == 1 -> skip
   :: S22_last == 2 -> HFSM_S222_exit()
fi;
printf("Exit-code of S22\n")}
```

Apart from executing genuine S22 exit code, the exit code procedure for state S22 it also looks at the last internal state of the state machine, and if the exit originates from S222, then the corresponding exit code will be invoked as well. Note that we first check for exit code to be invoked for lower level states, and then second execute the code belonging to this state.

5 Inheritance and Polymorphism

Object-oriented approaches to system design like ROOM have suggested splitting up system models into classes, and allowing classes to *inherit* other classes. Including an inheritance concept into v-Promela is mainly motivated by the following points. First, inheritance allows to stay compatible with techniques like ROOM and UML-RT, and with an emerging dominant way of thinking amongst engineers and scientists in real-time systems design. Second, the abstraction may help in dealing with the complexity of large requirements specifications, including re-use of parts of a v-Promela model and facilitated maintenance. Finally, class structuring and inheritance relationships may reflect inherent symmetry features in the problem domain. It is the hope that these symmetries can be later exploited to improve efficiency of the formal analysis process. In v-Promela, we envision two sorts of inheritance: 1) *structure and behaviour* inheritance, and 2) *procedural* or *abstract data type* inheritance. The first is a concept based on capsules and permits their refinement through inheritance. The second concept relies on the ability to define classes and operational abstractions (or "methods") on these classes. At the current state of the v-Promela definitions we concentrate on the former aspect of inheritance and leave the latter up for future research.

Structure and Behaviour Inheritance. Our approach towards structure and behaviour inheritance is guided by the way that ROOM addresses this issue. Space limitations do not permit us to repeat the rationale for the choice of inheritance in ROOM and we refer to the discussion in Chapter 9 of [16] for a more detailed treatment, and to [8] for a more detailed discussion.

In v-Promela, we may define a new capsule class as a sub-class of an existing class (either in the same model, or from a different model using a package identifier). A capsule class may only have one parent class, i.e., single inheritance is used in v-Promela. With the exception of protocol and data classes, a capsule subclass inherits all structural attributes of the superclass. Behavioural definitions are additional attributes of capsules. Hence, all behavioural attributes like

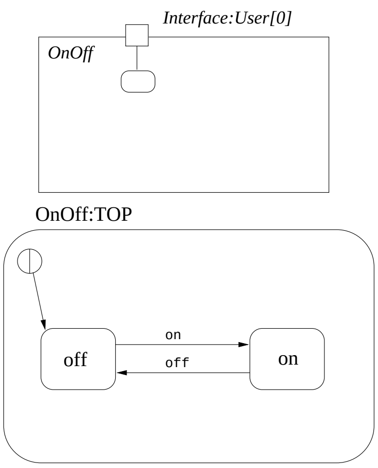

Figure 10: Capsule structure and behaviour definition

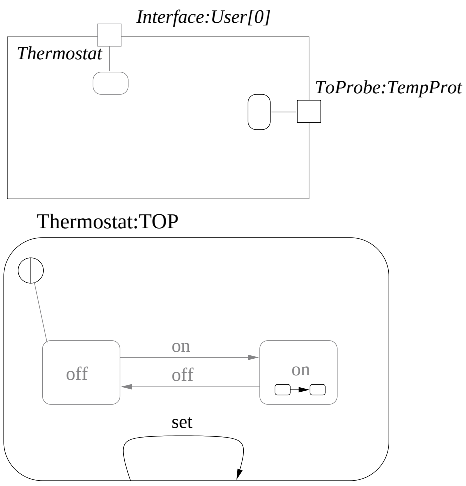

Figure 11: Capsule structure and behaviour definition with inherited attributes

states, transitions, entry- and exit code will be in herited as well. In v-Promela, all inherited attributes can be overwritten or deleted within the subclass. This has an effect on the definition of these attributes in the subclass, but does not affect the original definition in the superclass.

To illustrate inheritance, consider the definition of an capsule named `OnOff` in Figure 10. It features an external interface through a port called `Interface` of protocol type `User`. The interface permits external entities to provide `on` and `off` messages to the switch. A subclasses should be used to define more specific aspects of the system, and this is usually done by adding attributes. The capsule class `Thermostat` in Figure 11 is a sub-class of the capsule class `OnOff`. All inherited, non-local attributes are shown in greyed colour, whereas local attributes are shown in black. Structurally, `Thermostat` has inherited from `OnOff` the `Interface` port. At the same time, a port named `ToProbe` has been added. It allows interactions with the probe so that temperature values can be sampled and heating and cooling can be controlled. The behaviour definition of `Thermostat` inherits the `on` and `off` states as well as transitions between them from `OnOff`. However, the behaviour of `Thermostat` is more complex than that of the simple `OnOff` switch. First, a non-exiting self-transition processes the set target value for the thermostat that is received from the environment through the `Interface` port. Second, the `on` state has a refinement and is now responsible for regulating the temperature of the probe to a target value.

The implementation of the inheritance concepts is in essence a question of suitable user interface design in VOOPI. For every attributes in every capsule definition it must be determined whether it is a local definition, or whether it has been inherited from a different capsule. In case of an inherited definition, there must be a pointer to the original definition of the attribute, and this pointer must be used at editing time to display inherited attributes, as well as at compile time in order to produce suitable Promela code. During the editing of a v-Promela model, changes to a superclas will be immediately reflected in any subclass that inherits the change. After compile time the nature of the inheritance hierarchy is irrelevant - the determination of an capsule instances' attributes is exclusively done at compile time.

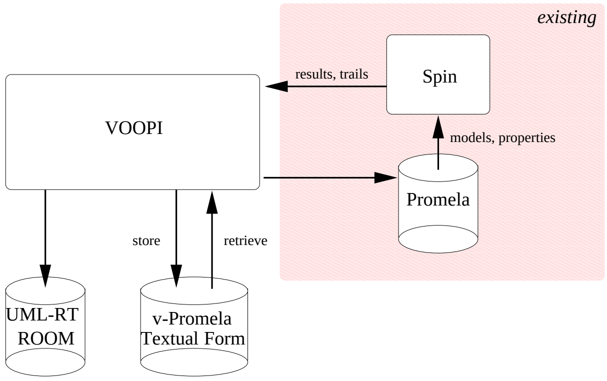

Figure 12: Architecture of VOOPI tool

6 VOOPI Tool Architecture

v-Promela is a visual notation intended for automated model analysis and comprehensive tool support vor v-Promela is needed. At the heart of our tool architecture in Figure 12 is the *VOOPI* (Visual Object-Oriented Promela Interface) tool. It provides for visually editing v-Promela models and storing them in an internal format, called *v-Promela Textual Form*[4]. The textual form captures logics and layouts of the v-Promela models.

For the purpose of model analysis, v-Promela models will be translated into plain Promela and the Spin model checker [9] is used to check properties of the Promela model. Once Spin has produced analysis results these will be fed back into VOOPI. Analysis results need to be filtered and interpreted inside VOOPI in order to make sense in the context of the original v-Promela model. As an example, if Promela reports an assertion violation relating to a global data object, then the name of this object needs to be interpreted to determine, which capsule instance this variable belongs to. VOOPI has some functionality that resembles the current Xspin system. Note that the existence of v-Promela is completely transparent to Spin, which allows us to use the existing Spin verification engine without changes. VOOPI also contains a modified version of the simulation capability of Xspin which allows individual capsule instances, data objects and inter-object message exchanges to be observed and traced. Simulation capabilities that go beyond Xspin (animation of state machines and structural elements) will be added later to VOOPI.

Compatibility with the UML-RT standard is motivated by the use of v-Promela models at subsequent stages of the design cycle. In comparison, Promela models are relatively short-lived and rarely survive the requirements stage. Therefore, VOOPI is capable of synthesizing ROOM models using the ROOM linear form notation. In some places we made semantic assumptions that violate either ROOM or UML-RT semantics. Examples are the queueing strategy for ports where v-Promela uses one queue per port, while ROOM has one input queue for all ports, and the outer level priority resolution of concurrently enabled HFSM transitions where ROOM sets a priority on lower level transitions. VOOPI will be equipped with compatibility modes that allow ROOM-compatible models to be synthesized.

[4]For a syntax definition of this of the textual form see [8].

7 Conclusion

We have presented v-Promela, a visual, object-oriented modeling language. v-Promela relies on the premise that system modeling requires both structural and behavioural views on the system. The concept of a capsule defines the Because v-Promela can be compiled into Promela, v-Promela models avail themselves to formal analysis using the Spin model checking tool. To facilitate the handling of v-Promela specifications we have sketched the design of the VOOPI toolset. VOOPI remains to be implemented.

Several research issues need to be addressed in the future. v-Promela does not currently support a notation to capture properties that could be checked against the v-Promela model or the resulting Promela code. The Spin system, which VOOPI uses for validation purposes, takes advantage of various formats in which properties can be specified in Promela. States can be labeled as *end* or *progress* states in Promela, and this concept can be easily transferred to states within v-Promela. Promela allows state properties to be attached to the model using *assertions*. We envisage that these can easily be incorporated in the v-Promela model. For instance, it would be possible to attach assertions to either states or to transitions so that pre- and post-conditions, as suggested in UML, can be specified. Promela allows state-oriented temporal propeties to be specified as *never claims*. Never claims are rather unintuitive to the unexperienced modeler and it has therefore been suggested to use Linear Time Temporal Logic (LTL) [12] to capture temporal requirements. LTL formulas can be interpreted as never claims and there is an automatic translation from LTL formulas into Promela never claims that is part of the current Xspin implementation. Future research addresses notational support for specifying LTL properties. We expect that the patterns-based approach of [5] will prove to be very helpful in defining an intuitive notation. Note that LTL formulas will be capable of specifying the invariant properties that the UML standards suggests, as far as behavioural aspects of the system are concerned. Finally, event-oriented temporal properties can be specified in Promela using *trace* assertions. Like never claims, trace assertions do not appear to be a real end-user notation, hence we look for alternative capturing means. First, LTL can be interpreted based on events, which would provide for a sufficient high-level interface. Second, Message Sequence Charts (MSCs) [11] are inherently an event-oriented notation, and we will investigate how properties expressed in MSCs can be translated into trace assertions so that MSCs can be checked against v-Promela models.

A further area of research addresses the question how the formal analysis process can benefit from the symmetries and the modularization that v-Promela offers, and how compositional verification techniques can exploit the modularization provided by v-Promela.

Acknowledgements

The authors wish to thank Margaret Smith for insightful comments during the early design stages of v-Promela. Bran Selic shared his view on how UML-RT will evolve with the authors.

References

[1] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison Wesley Longman, 1998.

[2] E. Clarke, O. Grumberg, and D. Long. Verification tools for finite-state concurrent systems. In *A Decade of concurrency – Reflections and Perspectives*, volume 803 of *Lecture Notes in Computer Science*. Springer Verlag, 1994.

[3] Rational Software Corporation. UML notation guide. Research report, 1997. See also `http://www.rational.com/uml`. Also published by The Object Management Group, document number ad/07-08-05.

[4] Alan M. Davis. *Software Requirements: Objects, Functions and States*. Prentice-Hall, Inc., 1993.

[5] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Property Specification Patterns for Finite State Verification. In *Proceedings of the 2nd Workshop on Formal Methods in Software Practice*, March 1998. For access to the patterns catalog see URL `http://www.cis.ksu.edu/~dwyer/spec-patterns.html`.

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

[7] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall International, 1991. Also available online via URL `http://cm.bell-labs.com/cm/cs/who/gerard/popd.html`.

[8] G. J. Holzmann and S. Leue. Towards v-Promala, a visual, object-oriented interface for Xspin. Unpublished manuscript, 1998.

[9] G.J. Holzmann. The model checker spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.

[10] ITU-T. Recommendation Z.100: Specification and Description Language (SDL). Geneva, Switzerland, 1993.

[11] ITU-T. Recommendation Z.120, Message Sequence Chart (MSC). ITU - Telecommunication Standardization Sector, Geneva, Switzerland, May 1996. Review Draft Version.

[12] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.

[13] E. Mikk, Y. Lakhnech, M. Siegel, and G. J. Holzmann. Implementing statecharts in promela/spin. Technical Report BL011272-971203-25TM, Murray Hill, NJ 07974, USA, 1997.

[14] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall International, 1991.

[15] B. Selic. Various Personal Communications, May – August 1998.

[16] B. Selic, G. Gullekson, and P.T. Ward. *Real-Time Object-Oriented Modelling*. John Wiley & Sons, Inc., 1994.

[17] B. Selic and J. Rumbaugh. Using UML for modeling complex real-time systems. `http://www.objectime.com/new/uml/index.html`, March 1998.