First publ. as paper in: Proceedings of the Second International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '99), 2 - 5 May 1999, Saint-Malo, France

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

Stefan Leue[*]
Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario N2L 3G1, Canada
sleue@uwaterloo.ca

Gerard Holzmann
Bell Laboratories
600 Mountain Ave
Murray Hill, NJ 07974-0636, USA
gerard@research.bell-labs.com

Abstract

We describe the design of VIP*, a graphical front-end to the model checker* SPIN. VIP *supports a visual formalism, called* v-Promela *that connects the model checker to modern hierarchical notations for the specification of object-oriented, reactive systems. The formalism is comparable to formalisms such as UML-RT, ROOM, and Statecharts, but is presented here in a framework that allows us to combine the benefits of a visual, hierarchical specification method with the power of LTL model checking provided by* SPIN. *Like comparable formalisms,* VIP *can describe hierarchies of behaviour and of system structure. The formalism is designed to be transparent to the* SPIN *model checker itself, by allowing all central constructs to be translated mechanically into basic* PROMELA*, as already supported by the existing model checker.*

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 [3] 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 [14], OMT [12] and SDL [8] 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 the abstraction mechanisms necessary to build comprehensive tractable requirements models that would avail themselves to automated analysis. At the other end, there are languages like Promela [5] and SMV [2] which provide the abstraction mechanisms required to obtain mathematically tractable models, but fail to provide support for more than the early stages of the life cycle as well as to support the modularization mechanism needed to cope with largely complex requirements specifications.

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].
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

[*] The work of this author was mainly performed while visiting Bell Laboratories.

of concurrent objects, inter-object communication, inheritance and polymorphism.

5. All structural and behavioural concepts of v-Promela should be *implementable in Promela*. A compilation of v-Promela into Promela allows us the use of the XSPIN model checker to validate v-Promela models.

6. The success of a visual modeling notation hinges upon the availability of suitable *CASE tool* support. We sketch the design of the VPI 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.

Furthermore, we expect that v-Promela will avail itself to compositional verification methods, e.g., assume-guarantee reasoning, as well as to design-by-contract like requirements capture by pre-/post-condition pairs and invariants. We also foresee that v-Promela - based modeling 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 for our visual notation as well. However, we feel that the support that Promela offers for inter process 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) [15]. UML-RT is primarily based on two other techniques. First, it is an extension of the *Unified Modeling Language* (UML) [11] 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 [15]), based on the ROOM method [14]. It is expected that based on its broad support UML and its derivative methods will gain substantial momentum in the software engineering community. The current state of the UML-RT language definition is rather preliminary. As one of the authors puts 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 [13]. In cases both ROOM and UML-RT do not provide sufficient guidance in the design of v-Promela constructs, we have taken the liberty of defining our own language features.

UML, ROOM and UML-RT have themselves been greatly influenced by extensive precursory work. Most notably, work on Statecharts [4] has greatly influenced all these methods. Consequently, a subset of the Statecharts language can also be found in v-Promela. There is also a large body of work on Promela [5], including a web site that provides access to many Promela and XSPIN related resources[3].

A translation from pure Statecharts into Promela has been suggested in [10]. This work translates a much more faithful interpretation of Statecharts than the one used in v-Promela, which is why our work and the one in [10] are not directly comparable.

Overview. We assume that the reader has some familiarity with the basics of the Promela language [5, 6]. 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 VIP (Visual Interface for Promela), 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 [15], and inherits many important ideas from ROOM [14].

Models and Packages. Models and Packages are high-level organizational constructs in v-Promela that help to organize 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, similar to the way this is accomplished in ROOM. For open systems, during simulation in the VIP tool the user may interact with the system to inject environment events. For val-

[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`.

idation purposes, a system must be closed, and VIP offers support for randomized closing of an open v-Promela system.

Capsules. Capsules represent structural and behavioural abstraction in v-Promela. 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. 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, which can be thought of describing the architecture of a very simple *plain old telephony system* (POTS) in which calls from an originating party will be forwarded to a terminating party. In Figure 1 this stucture is captured by the definition of the capsule class `POTS`. This capsule class is structurally refined as consisting of instances of capsule classes `Originator` and `Terminator`. We use an iconic representations of state machine diagrams and collaboration diagrams placed in the upper right hand corner of a capsule definition to indicate that the capsule has either a structural refinement, or that it has a state machine associated with it.

Replication, Optionality and Destruction of Capsule Instances. To account for the dynamic nature of concurrent real-time systems a varying number of capsule *instances* may exist any given point in time in a v-Promela model. The state machine of a composite capsule in v-Promela is capable of dynamically creating and destroying instances of contained capsules. A replication factor for a contained capsule of the format `[` *minimum-number* `..` *maximum-number* `]` indicates the number of instances generated at instantiation time of the containing capsule, and the maximum number of possible instances, respectively. If the lower limit is 0, we say that the capsule is *optional*. When no instance replication factor is given, the at the creation time of a composite capsule one instance of all its subcapsules will also be generated. As an example, when an instance of the capsule class `POTS` is created, one capsule instance of class `Originator` will be created, but no instance of class `Terminator`. The strings before the colon define the names of the capsule instances, for replicated capsules the instance name refers to an array. A capsule instance `Q` may destroy any contained non-replicated capsule instance `P` in a hierarchy-downwards direction by the command `destroy P`. If `P` is replicated, the instantiation

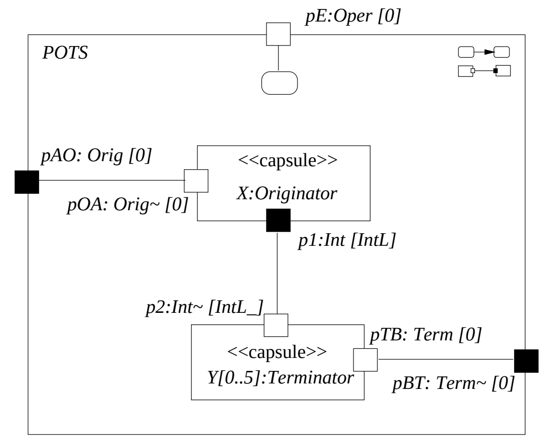

Figure 1. Composite capsule with replicated contained capsules

number of the replica to be destroyed is attached in squared brackets. If the capsule instance to be destroyed is itself a composite capsule, then all contained capsule instances need to be destroyed as well.

Data Classes. In v-Promela, data objects are instances of data classes. Data classes implement the basic Promela data types (i.e., *bit*, *bool*, *byte*, *short* and *int*) as well as any data structure that can be derived from the basic data types in Promela, with the exception of channels. 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. In ROOM and UML-RT, inter-process communication is exclusively message passing and there is no shared variable communication. For v-Promela we decided to be more permissive than ROOM and UML-RT in order to accommodate low-level modeling needs. Hence, we define the visibility attributes *private*, *protected* and *public* to limit visibility to the scope in which an object is declared, to be global in the name scope of the capsule in which they are declared, and to be globally accessible to all capsule instances, respectively. If more than one capsule uses the same name for a protected or private variable, the innermost re-definition of that name will overlay the more global definition. When referring to a visible object in a different name scope v-Promela uses path names to identify this object.

Capsule Replication and Data Objects. Data objects may belong to replicated capsule instances and every replica of a capsule instance hosts replicas of all data objects declared for this capsule class. 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 a capsule of type `Terminator` provides a public data object `digit`, then this could be referred to as `Z.Y[3]::digit` where `Z` is the name of the containing instance of class `POTS`. v-Promela provides for a global array variable `_procid` which stores the information whether a particular capsule instance exists at any point in time, or not.

Messages. A message in v-Promela consists of two parts, a *message identifier* and an optional *message body*. The message identifier is an abstraction of the contents of the message, which may be conveyed in the message body. The type of the message identifier corresponds to a Promela `mtype`, the message body is of the type of any valid data class or of a basic data type. The semantics of message reception in v-Promela is such that the message identifier name in a receive statement must match the identifier of the message at the head of the queue, and the type of the variables that follow must match the type of the message body. The following table defines some of the message types used in the example in Figure 1:

Message identifier	*Message body type* (optional)
`start`	
`off_hook`	`switchhook-msg`
`dial`	`dial-msg`
`connect`	`internal-msg`
`ring_tone`	`term-msg`

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, which describes the message exchanges between originators and terminators as seen from the originator:

Protocol definition: `Int`	
out-message identifiers	*in-message identifiers*
`connect`	`disconnect`
	`party-busy`

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 message exchange interfaces of a capsule are called ports. A port is an abstraction for the concepts of recipient and sender addresses, queueing discipline and queue capacity of the communication infrastructure that the system uses. Ports are represented by either white or black filled boxes on the boundaries of capsules, c f. the boxes labeled '`pAO:Orig [0]`' and '`p2:Int~[IntL_]`'. The string before the colon defines the name of a port, optionally followd by the replication factor of this port, enclosed in brackets. The string after the colon and before the blank denotes the protocol type of the port. The string in the squared brackets that follows the type 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. 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 `pE` in Figure 1.

Connectors. Connections between ports indicate the fact that messages can flow between capsules. 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. 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., [14]), 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 and Synchronizers. In order to remain more consistent with the modeling capabilities of 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. In v-Promela we allow

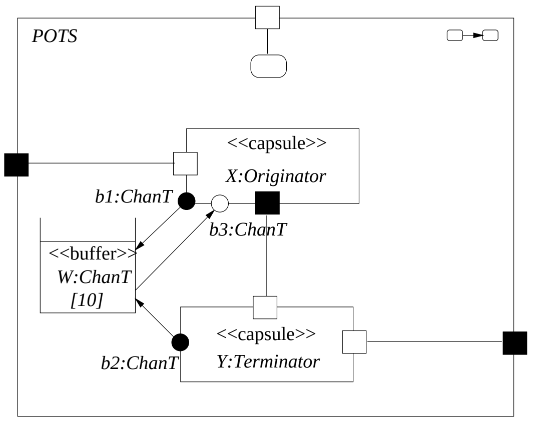

Figure 2. Composite capsule with contained buffer

an arbitrary set of capsule instances to write objects into or to read from a designated buffer object. The concept is illustrated in Figure 2 where the originator and terminator processes place tokens representing channel hardware resources in a buffer instance of name `W` and type `ChanT`, which is the type of the objects that they can host. 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. A *synchronizer* is similar to a buffer except that it has a capacity of zero. Any interaction with a synchronizer is via synchronous rendez-vous communication.

Implementation in Promela

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 1 we use a byte type instance counter variable `POTS_Terminator_instnr` to count the number of instances of capsule class `Terminator`. The instance counter variable is used as an additional guard for the executability of a `run` statement enforcing the maximum replication constraint for a capsule class. 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. We keep a global array variable `POTS_Terminator_procid` 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 `POTS` was containing a `run Terminator` statement which would be translated into the following Promela code sequence:

```
d_step {
(POTS_Terminator_instnr < 5) ->
  _tpid = run Termonator(POTS_Terminator_instn);
  POTS_Terminator_procid
         [ POTS_Terminator_instn ] = _tpid;
  POTS_Terminator_instnr++ }
```

The declaration of the proctype corresponding to the replicated capsule class `Terminator` is `proctype Terminator(short _me)`. Inside `Terminator`, access to a global data object like `digit` would need to be translated into access to a corresponding field in a global array data structure that keeps the values of this data object for all replications of the capsule, for instance `POTS_Terminator_digit [ _me ]` for access to data object `digit` inside instance `_me` of capsule `Terminator`.

To illustrate dynamic destruction of capsule instances, 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 } } }
proctype Q() { ...
  P__destroy = 1 ...}
```

In the implementation of private data objects, local Promela variable declarations are used, and for protected and public data objects, global variable declarations are used. To disambiguate non-private data object names we use name prefixing in the Promela code, where 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 `digit` in the 5th instance of capsule instance `Y` within capsule instance `Z`, we obtain the statement `Z_Y_digit [5]` in the corresponding 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 different messages so that their message bodies are of different incompatible types. However, Promela channels accept only 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 is obtained by using a Promela `typedef` for a Cartesian product construction over the message body types of all messages of the same direction in a protocol. For instance, the protocol half of protocol `Int` containing all messages flowing from the `Initiator` to the `Terminator` would be defined as

```
typedef Int_msgtype {internal_msg internal;
                     error_msg error}
```

where `internal_msg` is the message body type of `disconnect` messages, and `error_msg` is the type of `busy` messages. A channel implementing port `p1` in Figure 1 would be defined to be of type `Int_msgtype`.

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

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. Behaviour in v-Promela is specified using hierarchical communicating extended finite state machines (HCEFSMs) which have some similarity with Statecharts [4]. However, while Statecharts know the concept of "concurrent" states, in v-Promela control within each capsule is strictly sequential and concurrency is expressed using concurrent composition of capsules.

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. Figure 3 contains the state

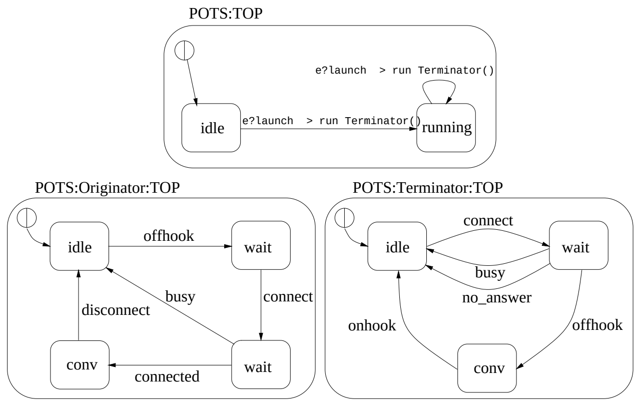

Figure 3. Behaviour specification for example `POTS`

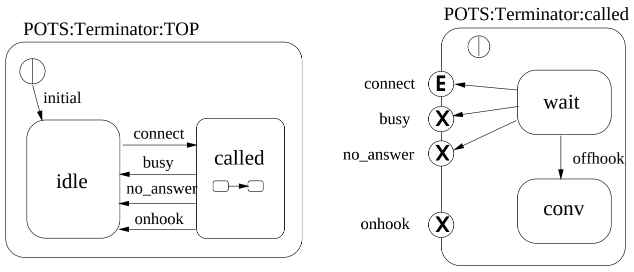

Figure 4. Hierarchical decomposition of the `POTS:Terminator:TOP` state

machines for the capsules in Figure 1. Every state contained in a state diagram may be refined by another state diagram. We call the resulting state machine hierarchical. Figure 4 presents the hierarchical decomposition of the `POTS:Terminator:TOP` state. State refinement in v-Promela can be represented in various forms. A hierarchical state machine may be drawn as a multi-layered state machine in 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 4. In this notation, a circle with an 'E' on the boundary of a composite state indicates a *state entry point* and a circle with an 'X' on the boundary of a composite state represents a *state exit point*. To explain the hierarchical state machine concept in v-Promela in more detail, we will from now on refer to the more generic example in Figure 5.

As most hierarchical state machine models v-Promela knows the concept of group transitions. These occur when the system is in a lower-level state and a higher-level transition is enabled and then executed. As an illustration, assume that the state machine in Figure 5 is in state *(S2, S22,*

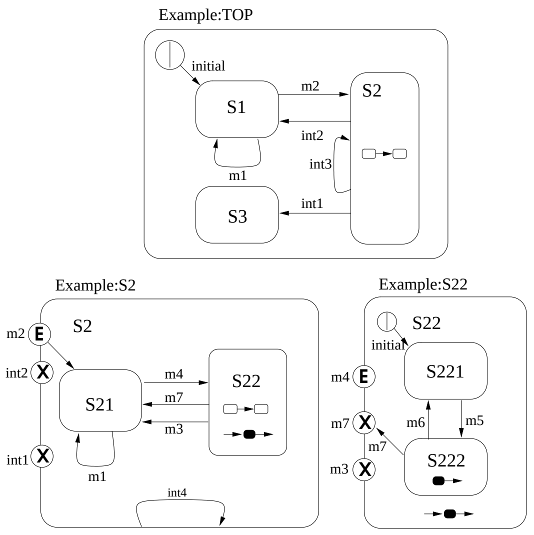

Figure 5. Generic HFSM example

S222) and an event `int1` is happening, then the system will transition into state *(S3)*. A transition into a hierarchical state can be driven by two policies. First, *return to history* drives the system into the substate 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 5 would lead into `S222` if this state was last pre-empted. Alternatively, the *return to explicit state* policy leads back to a contained state irrespective of where the composite state was left. As an example consider transition `m2` in Figure 5 which will always lead into `S21`. The choice of the return policy is implicit in the specification and depends on whether entry points are connected to an internal state, or not.

When an active capsule is instantiated, its contained state machine will be driven into the initial state. An initial state is denoted by a circle with a vertical bar. The system will also carry out an initial transition labeled `initial` as part of the instantiation process.

Transition Enabling and Transition Code. Transition code in v-Promela consists of a set of sequential v-Promela statements that will be executed while a capsule is performing the transition from one state into a successor state, or while executing an initial transition. 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 cases 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. We take advantage of the fact that Promela is a guarded command language - every statement in Promela 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 for the transitions of the `POTS:Originator:TOP` state machine in Figure 3 the corresponding transition code is edited in a separate table relating the symbolic transition name to a piece of v-Promela code. As an example, the `connect` labeled transition could be mapped to the code sequence `pOA?dial -> p1!connect`, indicating that the originator requests establishment of a connection to the terminator. Alternatively, the transition code can be directly attached to the transition label, as shown for the `POTS:TOP` state machine in Figure 3. 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.

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* ˆ *send-clause*

To remain consistent with these notations, v-Promela offers an optional transition code table format that is consistent with the above structure.

Promela Code within v-Promela Transitions. The choice of a suitable subset of the Promela language that can appear as part of v-Promela transition or state code is determined by a compromise between sufficient expressiveness and an avoidance of clashes with other v-Promela constructs. All references to variables must be to *data objects* declared within the scope of the state machine that the transition belongs to. *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. 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, with the exception of `enabled`, `name`, `np_` and `pc_val` which only have a meaning within Promela `never` claims. The set of predefined expressions in v-Promela code is 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. Send and receive statements in v-Promela refer to *port* names in order to specify a potential source or sink of the communication.

Semantic Assumptions. v-Promela is interpreted based on an interleaving model and on the assumption of a *run-to-completion* semantics. To ensure atomicity and run-to-completion semantics 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`[4] statement.

Entry and Exit Code. v-Promela offers the definition of *entry* and *exit* code sequences for states which will be executed whenever a state is entered or exited, respectively. While being only a syntactic shorthand for transition code, entry and exit code bear potential for nice abstractions and may greatly increase readability of a specification.

Implementation in Promela

v-Promela states are translated into Promela *labels*, and transitions are implemented as Promela *goto* statements. Transition code will be executed prior to the `goto` representing the respective transition. Group transitions are implemented using the Promela `unless` construct. To implement run to completion semantics transition code is embraced by a `d_step` statement, unless the statements in the sequence require a combination of `atomic` and `d_step` to ensure atomicity. To implement the return-to-history semantics we use a byte size one place 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 complementary 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 cascading series of exit and entry code executions. Entry and exit code is implemented as non-parameterized *inline* procedures in Promela. Putting all the concepts together, the following example illustrates the implementation of a portion of the hierarchical state machine defined in Figure 5:

```
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")}
```

5. Inheritance and Polymorphism

Object-oriented approaches to system design like ROOM and UML-RT have suggested splitting system models into classes, and allowing classes to *inherit* attributes of other classes. In v-Promela, we envision two sorts of inheritance: 1) *structure and behaviour* inheritance, and 2) *procedural* or *abstract data type* inheritance. At the current state of the v-Promela definitions we concentrate on the former aspect of inheritance, which is closely related to the capsule concept as defined abot, and leave the latter aspect of inheritance up for future research.

Our approach towards structure and behaviour inheritance is guided by the way that ROOM addresses this issue, see in particular the discussion in Chapter 9 of [14]. In v-Promela, we may define a new capsule class as a sub-class

[4]`d_step` is a statement modifier in Promela that asserts atomic execution of sets of *deterministic*, *non-blocking* and *terminating* Promela statements.

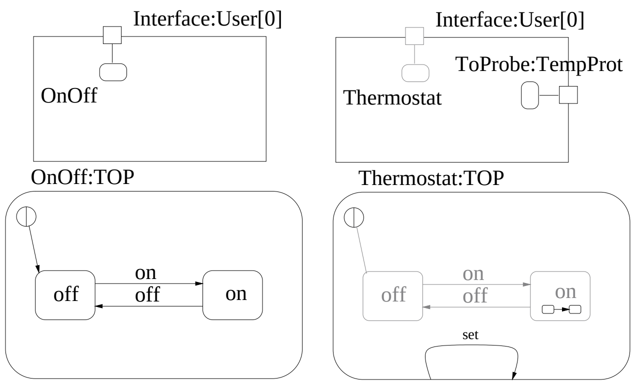

Figure 6. Abstract definition (left) and refinement (right)

of an existing class (either in the same model, or from a different model using a module identifier). We use single inheritance, i.e., a capsule class may only have one parent class. 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 states, transitions, entry- and exit code will be inherited. All inherited attributes can be overwritten or deleted within the subclass. The effect of this type of polymorphism is strictly local to the subclass.

To illustrate inheritance, consider the definition of an capsule named `OnOff` in Figure 6. 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 6 is a sub-class refinement 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 the `Interface` port from `OnOff`. At the same time, a port named `ToProbe` has been added to the subclass. 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.

Implementation in Promela

In the way that we have defined it, inheritance is a compile-time mechanism. Hence, the implementation of the inheritance concepts is in essence a question of suitable user interface design in VIP (see Section 6). For every attribute 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 superclass will be immediately reflected in any subclass that inherits the change.

6. VIP Tool Architecture

As a visual notation v-Promela can only be useful if it is supported by a graphical toolkit. In this Section we describe the architecture of *VIP*, the *Visual Interface to Promela* tool, c.f. Figure 7. VIP provides a visual editor for v-Promela models and allows them to be stored in an internal textual format, called *v-Promela Textual Form*[5]. 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 Promela code, based on the implementation strategy that we have described in previous Sections. The Spin model checker [6] is used to check properties of the resulting Promela model. The analysis results that Spin produces will be fed back into VIP, and VIP will interpret them in the context of the original v-Promela model. As an example for the interpretation step, 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. Note that the existence of v-Promela is completely transparent to Spin, which allows us to use the existing Spin verification engine without changes. VIP 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 also be implemented in VIP.

Promela models are relatively short-lived and rarely survive the requirements stage. Compatibility of v-Promela with the UML-RT standard is therefore motivated by the wish to use v-Promela models throughout a larger number of stages of the design cycle. Consequently, VIP is capable of synthesizing ROOM models using the ROOM linear

[5]A preliminary syntax definition of the textual form is included in [7].

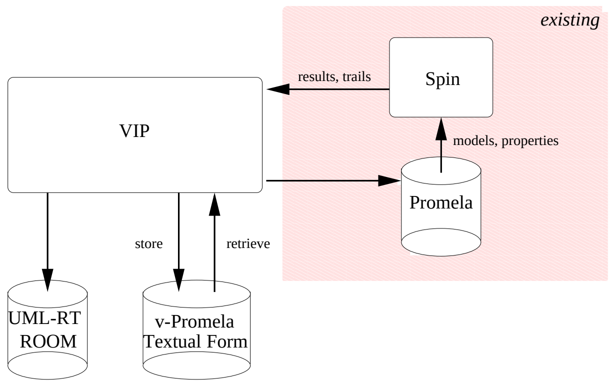

Figure 7. Architecture of VIP tool

form notation. In some places we made semantic assumptions that violate either ROOM or UML-RT semantics, and some constructs of v-Promela do not have counterparts in ROOM or UML/RT. 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. VIP will be equipped with compatibility modes that limit the editing of v-Promela models so that compatibility with ROOM or UML/RT is maintained.

7. Conclusion

We have presented v-Promela, a visual, object-oriented modeling language for concurrent reactive systems. v-Promela relies on the premise that system modeling requires both structural and behavioural views on the system, relying on the concepts of capsules and hierarchical state machines. 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 VIP tool which is currently being implementing. Because of the difficulty of manually editing v-Promela models and translating them into Promela, comprehensive case studies await completion of the implementation of VIP. However, we have partially edited a telephone switch model in the spirit of v-Promela.

Several research issues need to be addressed in the future. v-Promela does not yet support a notation to capture properties that could be checked against the specification. The Spin system takes advantage of various formats in which properties can be specified. States can be labeled as *end* or *progress* states in Promela, and this concept can be easily transferred to states in v-Promela. The Spin tool allows properties to be attached to the model using *assertions*. These can be incorporated readily 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. Spin allows state-oriented temporal properties to be specified as omega automata or as LTL formulae, e.g., [9]. Future research addresses notational support for specifying LTL properties within VIP.

Acknowledgements

The authors thank Margaret Smith for insightful comments during the design of v-Promela. We also thank Bran Selic for sharing with us his views on how UML-RT will evolve.

References

[1] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison Wesley, 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 *LNCS*. Springer Verlag, 1994.
[3] A. Davis. *Software Requirements: Objects, Functions and States*. Prentice-Hall, 1993.
[4] D. Harel. Statecharts: A visual formalisation for complex systems. *Science of Computer Programming*, 8:231–274, 1987.
[5] G. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.
[6] G. Holzmann. The model checker spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997.
[7] G. Holzmann and S. Leue. Towards v-Promala, a visual, object-oriented interface for spin. Unpublished manuscript, 1998.
[8] ITU-T. Recommendation Z.100: Specification and Description Language (SDL). Geneva, Switzerland, 1993.
[9] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer Verlag, 1992.
[10] E. Mikk, Y. Lakhnech, M. Siegel, and G. Holzmann. Implementing statecharts in promela/spin. In *Proc. Workshop on Industrial Strength Formal Specification Techniques WIFT'98*, Boca Raton, Fl., USA, October 1998. IEEE Computer Society.
[11] Rational Software Corporation. UML notation guide. Research report, 1997. See `http://www.rational.com/uml`. Published by The Object Management Group, AD/07-08-05.
[12] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.
[13] B. Selic. Various Personal Communications, May – August 1998.
[14] B. Selic, G. Gullekson, and P. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, Inc., 1994.
[15] B. Selic and J. Rumbaugh. Using UML for modeling complex real-time systems. `http://www.objectime.com/new/uml/index.html`, March 1998.