VIP: A Visual Editor and Compiler for
v-Promela[*]

Moataz Kamel[1] and Stefan Leue[2]

[1] Department of Electrical and Computer Engineering, University of Waterloo
Waterloo, Ontario N2L 3G1, Canada
m2kamel@uwaterloo.ca
http://fee.uwaterloo.ca/~m2kamel
[2] Institut für Informatik
Albert-Ludwigs-Universität Freiburg
D-79110 Freiburg, Germany
http://www.informatik.uni-freiburg.de/~leue
leue@informatik.uni-freiburg.de

Abstract. We describe the *Visual Interface to* Promela (VIP) tool that we have recently implemented. VIP supports the visual editing and maintenance of v-Promela models. v-Promela is a visual, object-oriented extension to Promela, the input language to the Spin model checker. We introduce the v-Promela notation as supported by the VIP editor, discuss Promela code generation, and describe the process of property validation for the resulting models. Our discussion centers around two case studies, a call processing system and the CORBA GIOP protocol.

1 Introduction

As Davis argues in [2], a significant return of investment can be expected when investing resources into the early stages of the software design cycle: in particular, fixing design flaws at the requirements stage can be 200 times less expensive than fixing them at the maintenance stage. Even corrections at the architectural design stage can be 50 times more cost efficient than at the maintenance stage. The inherent complexity of concurrent real-time systems makes it necessary to employ mechanized, formally supported methods to analyze early life-cycle artifacts. In this context the main questions to be answered are whether the requirements are consistent and correct with the intended behavior of the system, and whether the system's design correctly implements the requirements.

It has been shown that state-based modeling and automatic model checking is an effective tool for answering these questions for concurrent reactive systems. Recent advances in model checking research have made verification based on state space exploration more feasible for realistic software problems [6,7]. However, the introduction of formal methods in the software engineering process is

---

[*] The work documented in this paper was largely performed while the second author was with the University of Waterloo. We are currently working on a public release version of the VIP tool, interested parties are requested to contact the authors.

often hampered by textual interfaces laden with mathematical notations. Visual formalisms, on the other hand, appear to enjoy broad acceptance in engineering practice.

With the ever increasing complexity of concurrent, reactive systems that continue to be designed, the requirements and high-level design models are becoming sizeable artifacts themselves. In order to facilitate the model development process, we propose the alignment of the modeling language of a state-of-the art formal analysis tool with the state of the art visual, object-oriented hierarchical notations used in current software development. This has the benefit of fostering maintainability and evolvability of these models while increasing the chances that the models actually express the intentions of the designers and increasing the acceptance of formal analysis in the practical software engineering community.

In this paper we present the prototype of a graphical user interface-based tool called the *Visual Interface for Promela* (VIP) that we have developed. VIP supports a visual language called v-Promela which is the graphical extension of PROMELA, the input language for the model checker SPIN [8]. The tool provides graphical editing capabilities for v-Promela models and generates standard PROMELA code from the graphical representation. In the process of describing VIP we also show how modeling of complex real-time systems for the purpose of formal analysis can be based on a state-of-the art visual object-oriented notation, and that efficient tool support can be provided.

Related Work. VIP supports visual modeling using v-Promela. The v-Promela notation has first been described in [13] and [5]. The design of v-Promela was guided by a number of desiderata. First, we desired to use PROMELA/SPIN validation technology without any changes to the existing model checker. Hence every feature of v-Promela had to be compilable into Promela. Next, we were interested in a visual notation that would capture both structural and behavioral modeling aspects. We were also interested in providing hierarchical modeling and object-oriented concepts. Finally, we have attempted to comply as far as possible with existing or emerging software design methodology standards for concurrent real-time systems. As a consequence, the v-Promela notation inherited largely from the UML-RT notation [16]. UML-RT, which evolved from the ROOM notation [15], is supported by an industrial-strength case tool (ROSE-RT) and is expected to be a prominent player in the real-time systems domain in coming years. Some of the syntactic as well as some of the semantic aspects of UML-RT are not completely specified at this time. The authors of UML-RT suggest that these missing aspects can be derived from the definitions of the syntax and semantics of ROOM as given in [15]. The development of the VIP tool is described in more detail in [10] which also discusses some modifications of the original v-Promela proposal.

Organization of the Paper. In Section 2 we describe the architecture of the VIP tool, and we illustrate the use of VIP in Section 3 through an example. In Section 4 we discuss the v-Promela compiler implemented in VIP. In Section 5

we show how to perform property validation in the context of our approach. Finally, in Section 6 we discuss further issues related to the implementation of VIP, and we conclude in Section 7.

2   VIP Architecture

To support the editing and maintenance of v-Promela models we have developed the VIP (*Visual Interface to Promela*) tool. Figure 1 illustrates the functional architecture of VIP. We will describe the functionality of the VIP editor in the following section. The edited v-Promela models are compiled into Promela code by VIP, and the resulting Promela models can be validated by the Spin model checker. Spin error traces can then be re-interpreted in the context of the original v-Promela model. Currently, the re-interpretation has to be done manually. To store v-Promela models, we currently use JAVA class serialization. The use of this feature of the JAVA Development Toolkit saved considerable development time, however, to allow better future expandability we are currently working on implementing storage and retrieval functionality based on XML [17] schema definitions and an XML parser.
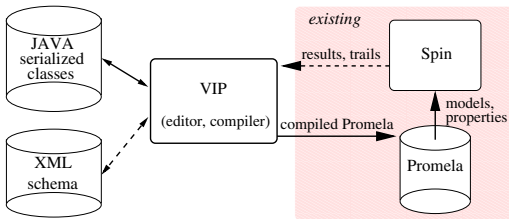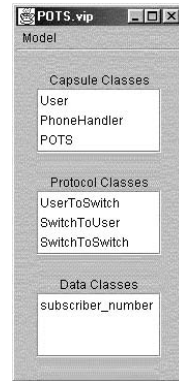

Fig. 1. VIP tool architecture.

Fig. 2. POTS Model editor.

3   Modeling in VIP

In this Section we describe the main features of the VIP graphical user interface. As a running example we use a simplified *Plain Old Telephony System* (POTS) call processing problem. The example consists of two `User` processes and two `PhoneHandler` processes contained in the environment of the POTS system. [1]

---

[1] In order to obtain models that can be translated into Promela all v-Promela models must be closed systems. Therefore, the environment behavior must be modeled as part of the system.

The `User` processes represent the behavior of the telephones which communicate with `PhoneHandler` processes which represent the call processing software inside the switch. The `PhoneHandler` processes are responsible for responding to events from the `User` processes as well as communicating with other `PhoneHandler`s in order to establish a voice connection.

3.1   Structural Modeling

Model Editor. The Model Editor is the starting point for creating v-Promela models. It allows the user to define the basic elements of a v-Promela model: capsule classes, protocol classes, and data classes. From the model editor the user can open editors for each one of the above mentioned basic elements. From the model editor, the user can also save the model or generate Promela code. Figure 2 illustrates the model editor for the POTS example. It specifies three capsule classes, three protocol classes, and a data class.
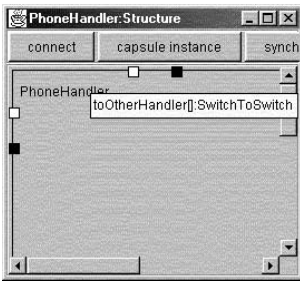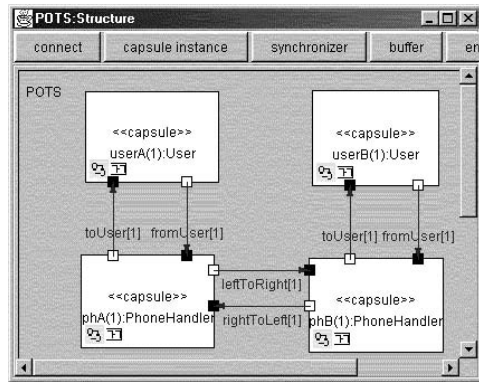

Fig. 3.   `PhoneHandler` capsule structure.

Fig. 4. Structure of POTS system.

Structure Editor The Structure Editor displays the internal structure of a chosen capsule class using the v-Promela graphical notation. The structure may consist of other capsule instances, buffers, synchronizers, ports, and connectors. Changes to the structure of the capsule class are automatically reflected in other views that contain an instance of the capsule class.

Figure 4 represents the POTS system structure. Concurrent objects are called *capsules* in accordance with the UML-RT terminology. The POTS system consists of a high-level capsule class called `POTS`. It is decomposed into four contained capsules as indicated by the capsule references in Figure 4. None of the contained capsule classes is further refined. However, Figure 3 illustrates the internal structure definition of the `PhoneHandler` capsule class. The black and white boxes at the border of the capsule denote in and outbound `ports`, respectively. Ports represent message passing interfaces for capsules. In contrast to v-Promela where ports are either uni- or bi-directional, all ports in VIP are uni-directional. Incorporating bi-directional ports into VIP is part of future improvements which are

in-progress. The type of a port is a protocol class, see below for a discussion of their definition. Ports need to be connected to ports in other capsule instances to enable messages to be exchanged. *Connectors*, as indicated by the labeled arrows in Figure 4, are used to establish connections between ports. The connector label shows the name of the connector and the message buffering capacity of the connector within brackets. Only ports of identical type can be connected, and a connector must join an out-port to an in-port.

A capsule instance can have an associated replication factor that is greater or equal to 1. The replication factor specifies the number of capsule instances that will be generated at instantiation time. For simplicity of presentation all capsule instances in the POTS example have a replication factor of 1.

Fig. 5. Protocol class definition for POTS model.

Protocol Classes. A Protocol class consists of a name and a list of message classes. Each message class has a name which identifies the message type and an associated message body type. Figure 5 illustrates the protocol class definition for the POTS example. The names of the protocol classes as well as the message class names are indicative of signals passed in a telephone switch.
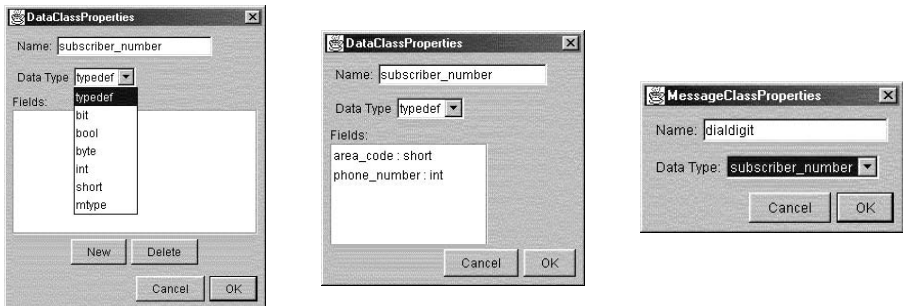

Fig. 6. Dataclass definition and usage in protocol class definitions.

Data Classes. Figure 6 illustrates the definition of data classes in VIP based on the available Promela data types. If a data class is mapped onto a basic

Promela data type it merely serves as an alias for that type. However, a data class definition can also take advantage of the `typedef` construct in Promela. Figure 6 shows the definition of the data class `subscriber_number` as a record consisting of a short integer field `area_code` and an integer field `phone_number`. Compared to the data type capabilities of languages like UML-RT the v-Promela possibilities are rather limited. However, this restriction is necessary to allow exhaustive model-checking by SPIN. The right side of Figure 6 shows how a data class definition is used to define a message body type. The `dialdigit` message of the `UserToSwitch` protocol is defined to have a message body type of `subscriber_number`.

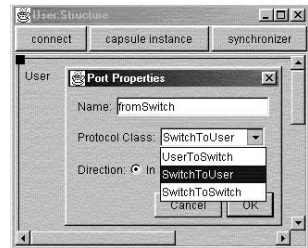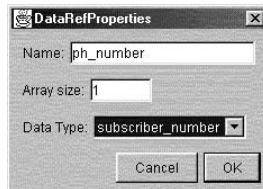**Fig. 7.** Data object definition.

Fig. 8. Defining ports based on protocol class definitions.

Data Objects. Data objects are instances of data classes that can be used in expressions. They are defined as attributes of capsule classes. Figure 7 illustrates the definition of a data object `ph_number` within the `User` capsule class as an instance of the `subscriber_number` data class. A data object may also be defined to be an array.

Buffers and Synchronizers. UML-RT is very rigid in the way that it allows inter-process communication to happen. Communication between processes takes place exclusively through point-to-point message exchanges via ports. In contrast, Promela allows arbitrary sets of processes to share channels or to synchronize via rendez-vous channels. In order to permit the more general modeling approach that Promela makes possible, v-Promela introduces the concepts of *buffers* and *synchronizers*. These concepts have been used in the CORBA GIOP example that is described in Section 5 and we have used them to model producer-consumer problems as well as semaphores.

3.2   Behavior Modeling

Behavior in v-Promela is modeled using hierarchical, communicating extended finite state machines.

Hierarchical State-Machines. The behavior editor in VIP allows for editing the state machine associated with a capsule class. In the POTS example, only the `User` and `PhoneHandler` capsules have state machines associated with them, as illustrated in Figures 9 and 10. The top-level state of any state machine has the name `TOP`. To illustrate the hierarchical nature of state machines in VIP, the refinement of the `await_digit` state of Figure 9 is shown in Figure 10. It should be noted that in the current implementation of VIP we consider two states with identical name labels to denote different states. We use various icons inside the state symbols to express attributes of the states. As an example, the circle in the lower left corner of the `idle` state in Figure 9 indicates that this state has been marked as a valid end state, and the icons in the `await_digit` state indicate that this state has a refinement and that entry code has been defined for it.

Circled `X` and `E` symbols indicate exit and entry points for multi-level transitions, respectively. Typical for the use of hierarchical state machines is the occurrence of *chained* and *group* transitions. For instance, if the `PhoneHandler` capsule is in any state contained within the `await_digit` state this state may be exited if the transition labeled `onhook_` in its super state executes. v-Promela and VIP allow for explicit return or return to history semantics for hierarchical state machines. If an entry point is not connected to a contained state, the return to history semantics will be chosen in the event of an incoming transition. Otherwise, the explicit return to a contained state is indicated by an arrow from the entry point to the contained target state.
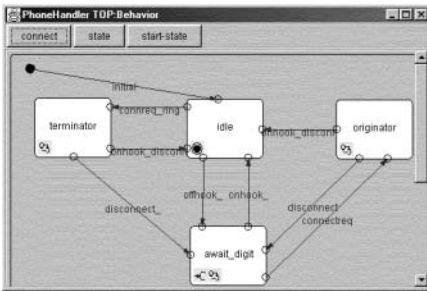

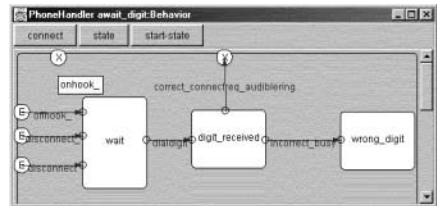**Fig. 9.** PhoneHandler TOP state.

Fig. 10. PhoneHandler TOP:await_digit state.

Transition code. The labels on the state transitions in the previous state transition diagrams have no executable semantics, they are merely used to identify the transition and to enhance readability of the state machine model. To attach enabling conditions and executable code to a state machine transition we open the transition's property editor, as shown in Figures 11 and 12. There are two formats for specifying transition code. Figure 11 shows the UML style of defining transition code. The code consists first of an *event* specification which in our implementation consists of the reception of a message from a port. In the editor we use pull-down menus that allow the user first to select a port from which the

message is to be received, and second to select a message type from the list of all messages that are allowable for the selected port. A *guard* can be specified as a side-effect free Promela expression. The chosen semantics is that only if the specified message is receivable and the guard is true will the transition be taken. The *action* can be an arbitrary Promela code fragment and will typically contain a send statement for a message. Care has to be taken that the code specified here is always executable and does not contain internal control flow loops. The current version of VIP does not parse the action code and hence it is the responsibility of the modeler to ensure that the code is meaningful. The second format, illustrated in Figure 12 shows the more liberal way of defining a transition. Unlike UML-RT, v-Promela transitions can be triggered by the executability of any Promela statement, in this case a boolean expression specified in the event clause. If the event clause evaluates to true then the action part will be executed. In this example a message of type `busytone` is sent to the port `toUser`.

As illustrated in Figure 13, state entry and exit code will be executed whenever a state is entered or exited, respectively. In our example this means that from whichever state we enter the `await_digit` state we will apply a `dialtone` signal to the user. This has bearing on the format in which transition code is specified. It could be argued that transition code could be specified by simply allowing an arbitrary Promela statement to be attached to a transition. However, exit code should be executed prior to executing the transition code, which would be impossible if we were not distinguishing a triggering event in the transition code.

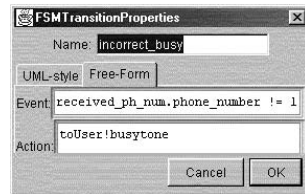**Fig. 11.** Promela code in action portion of transition definition.

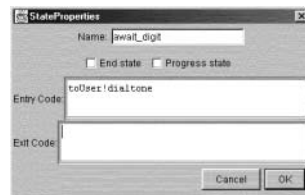**Fig. 12.** Free-form transition definition.

Fig. 13. Entry code definition.

4   The VIP Compiler

The basic principles of the Promela code generation are that capsules are mapped onto proctypes, protocols onto `mtype` declarations, and that ports with their connectors as well as buffers and synchronizers are mapped onto channel declarations. Message bodies are implemented as record structures (`typedef`s in Promela parlance) with one field for every message type that the protocol comprises. Data objects are mapped onto PROMELA variables. For a more extensive discussion we refer to [14,5,10].

Ports. Ports form part of the interface to capsule classes. Accordingly, the VIP compiler generates ports as channel type parameters in the parameter list of the corresponding capsule class proctype. On instantiation of a proctype, the VIP compiler generates the proper arguments to bind the correct connectors to the ports. For instance, in the POTS example the `POTS` proctype contains a channel declaration of the form `chan toUser2105717128 = [1] of SwitchToUser` [2]. The definition of the `User` proctype declares two ports (fromSwitch and toSwitch): `proctype User(chan fromSwitch, toSwitch )`. In the body of the POTS proctype, the User proctype is instantiated and its ports are bound by the code `run User( toUser2105717128, fromUser2016588168 )`. This scheme has the desirable property that ports can be referenced within the proctype without regard for which channel may be connected to it.

State Machine Encoding. The v-Promela control states could be implemented in two ways: a) by using Promela variables, or b) by using Promela control state labels and `goto` statements. Measurements documented in [10] suggest that, while the state vector size for both variants is identical, the state space size for variant a) is about twice the size for variant b). We therefore implemented variant b) in VIP. The state name label is chosen such that it starts with the state name in the v-Promela model and the VIP compiler adds a numeric sequence to disambiguate states with identical v-Promela names.

Transition Code. The code generated for transitions specified in the UML style starts with checking whether the specified message reception event is available by polling the relevant channel. The result is conjoined with the evaluation of the guard which yields the transition's enabling condition. The firing of the transition will cause the sequential execution of the following code fragments: first the message reception is performed, next the exit code of the current state, then the action code associated with the transition, followed by entry code for the new state and finally the `goto` into the successor control state. Promela does not allow polling the state of synchronous rendez-vous channels. Therefore, in such cases the first part of the transition code consists only of the rendez-vous communication, and any specified guard will be ignored. Figure 14 illustrates

---

[2] The VIP compiler disambiguates element names by concatenating a unique identifier to the name.

this mechanism for the `offhook` transition from the `idle` state of Figure 9. To enhance comprehensibility of the code, the compiler automatically inserts meaningful comments even if no entry or exit code has been specified. Note that the transition modeled here is a chained multi-level transition from the `idle` state into the `wait` sub-state contained in the `await_digit` state (c.f. Figure 10), and that the entry code defined for `await_digit`, i.e., `toUser!dialtone` is executed during this transition.

```
if
idle1723158139:
:: fromUser?[offhook] && true->
   fromUser?UserToSwitch_msg;
   /* exit idle */
   /* action offhook_ */
   /* entry await_digit */
   toUser!dialtone;
   /* entry wait */
   goto wait2091208315
...
fi
```

Fig. 14. Transition code for UML-style.

```
if
/* correct_connectreq_audiblering */
:: received_ph_num.phone_number == 1->
   /* exit digit_received */
   /* action
   correct_connectreq_audiblering */
   toOtherHandler!connectreq;
   toUser!audiblering;
   /* exit await_digit */
   /* action connectreq */
   /* entry originator */
   /* action untitled */
   /* entry party_ringing */
   goto party_ringing1956295048
```

Fig. 15. Transition code for free-form chained transition

Figure 15 illustrates the generated code for the VIP free form format for transition code specification. In this case the enabling condition is specified by an equality test on the value of a variable. The example also illustrates a chained transition, i.e., one that crosses nesting levels in the hierarchical state machine. All relevant entry and exit code specified along the transition chain is inlined into the transition code which, in certain cases, allows it to be processed as one atomic action[3].

Priority Schemes for Group Transitions. The implementation of group transitions depends on the priority model the user wishes to adopt. Three possible transition priority schemes are possible. In the first scheme, higher-level (group) transitions could take priority over lower-level transitions. Alternatively, lower-level transitions could take priority over higher-level transitions. Finally, both high and low-level transitions can be given equal priority. In VIP, all three priority schemes have implemented with the user having control over which scheme is used. Equal priority is the default in VIP. It is implemented simply by combining both high-level and low-level transition code as separate conditions in the same

---

[3] Promela allows non-blocking statements to be grouped into an atomic clause which can improve model checking efficiency.

```
ringing62399654:                    ringing62399654:
{if                                 {if
:: fromUser?[offhook] ...           :: fromUser?[onhook] ...
fi } unless {                       :: fromOtherHandler?[disconnect] ...
if                                  fi } unless {
:: fromUser?[onhook]...             if
:: fromOtherHandler?[disconnect] ...:: fromUser?[offhook] ...
fi}                                 fi}
```

Fig. 16. Priority on grouptransition   **Fig. 17.** Priority on local transition

```
ringing2063158907:
if
:: fromUser?[offhook] ...
:: fromUser?[onhook] ...
:: fromOtherHandler?[disconnect] ...
fi
```

Fig. 18. Transition code with equal priority

`if ... fi` statement as illustrated in Figure 18. Promela semantics dictate that multiple enabled conditions in an `if` statement are chosen non-deterministically resulting in equal priority among alternatives. The other two priority schemes are implemented using the Promela `{A}unless{B}` construct which pre-empts statements in A if the statement in B becomes executable. The first priority scheme is implemented by placing high-level transition code in B and low-level code in A. The second scheme implements the reverse. Figures 16 and 17 illustrate both of these schemes. Note that only the non-pre-emptive scheme complies with the *run-to-completion* semantics of UML-RT as described in [15].

5   Property Validation

Property validation of VIP-synthesized models currently relies on using the SPIN model checker to analyze the generated Promela models. The interpretation of the validation results that SPIN produces in the context of the v-Promela model currently relies on manual interpretation. We discuss two validation case studies using VIP-generated Promela code.

Validation of POTS. The previously presented POTS model was designed with the intention of revealing most of the significant features of v-Promela as supported by VIP. As a consequence, little attention was paid to developing a flawless model of POTS. The described POTS model is not free of deadlock. We have labeled the `idle` state in the `PhoneHandler` process and the `on_hook` state in the `User` process as end-states and an end-state check in SPIN easily shows a trace that terminates with one process an invalid end-state. This is mainly

due to the fact that we have not synchronized the `User` and `PhoneHandler` interactions. Thus, the `User` can repeatedly generate `offhook` and `onhook` event sequences that will eventually fill up the channel to the `PhoneHandler`. Also, call processing software is rarely "live", i.e., it only satisfies trivial liveness properties. A progress test in SPIN easily shows `offhook` and `onhook` cycles that do not imply system progress.

We therefore decided to answer the question of whether our POTS model was at all capable of doing it's very raison d'être, namely to connect two phones. In order to show that such a scenario exists, we formulate the converse property (namely, that the scenario does not exist) and hope that SPIN would refute the claim by showing us a trail to the contrary. The property we seek to prove is: "there exists a scenario in which both `PhoneHandler` processes are in the respective conversation states." The converse of the property is: "it is never the case that, at the same time, one `PhoneHandler` process reaches the conversation state for an originator and the other reaches the conversation state for a terminator." This property is represented by the LTL formula: `!<>(p && q)` where p and q are defined in SPIN by the state propositions:

```
#define p (PhoneHandler[2]@conversation_orig1985130888)
#define q (PhoneHandler[3]@conversation_term2034323067)
```
These expressions are referred to as *remote references* in Promela parlance. The expression `PhoneHandler[2]@conversation_orig1985130888` is a boolean expression that evaluates to true if the process named `PhoneHandler` with process id equal to 2 is at the control state labeled `conversation_orig1985130888`.

Shortly after running the model checker on the above claim, an error trace was found. As expected, the error trace that SPIN found showed a scenario in which both `Phone Handler` processes were in the respective conversation states. The validation required matching appr. 448,000 states, 680,000 transitions and 45.5 MByte of memory.

Validation of CORBA GIOP. In a previous work we modeled and formally validated the Common Object Request Broker Architecture (CORBA) *General Inter-ORB Protocol (GIOP)* [4] using PROMELA/SPIN validation technology [11]. In that work, a hand-built model of GIOP was developed and validated in Promela. Subsequently, a v-Promela model of GIOP was created using the VIP tool. The v-Promela model of GIOP has the equivalent functionality of the scaled-down, hand-built model that was validated in [11]. It comprises two `User`, two `Server`, one `GIOPClient` and two `GIOPAgent` processes. The model structure is shown in Figure 19. Behavior of the various capsules is defined using non-hierarchical v-Promela state machines.

Certain limitations of the VIP tool caused difficulty in expressing the structure of the model in a natural way. For example, replication of capsule instances was not implemented in the tool at the time the experiments were run and therefore, multiple instances of capsules had to be explicitly shown in the model. Similarly, replicated ports and channels were also not available in the tool and thus, buffers were used to emulate the desired communication structure.
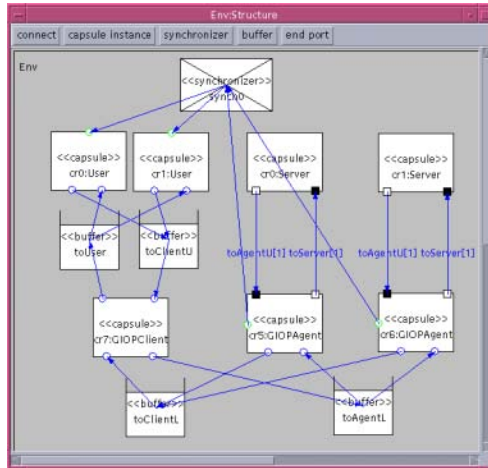
Fig. 19. Structure of the v-Promela GIOP model.

In the v-Promela model of GIOP, messages destined to the `GIOPClient` from either of the `User` processes are merged into a single buffer called `toClientU`. Similarly, messages destined for the `GIOPClient` from either of the `GIOPAgent` processes are merged into a buffer called `toClientL`. In the opposite direction, the `GIOPClient` may send messages to the `User` or `GIOPAgent` by placing them in the `toUser` and `toAgentL` buffers respectively. The messages are tagged with the Promela process id (pid) of the receiving process which only receives messages that contain its pid as a tag.

Table 1. Verification of safety and progress properties of hand-built versus automatically generated code.

Model	Property	State vector	Depth	States	Transitions	Memory usage
hand-built	safety	244 byte	119	77,261	92,566	17.697 Mb
VIP	safety	256 byte	171	8,704	13,236	4.590 Mb
hand-built	progress	248 byte	229	237,157	534,157	49.032 Mb
VIP	progress	260 byte	223	13,641	36,376	5.819 Mb

A basic safety properties verification run was carried out on the two models. This checked for invalid end-states and assertion violations. Equivalent assertions were placed in both models to check for invalid conditions such as reception of a `Reply` when no `Request` was outstanding. Also, end-state labels were placed in both models to identify valid end-states in each process. To validate progress properties a progress label was inserted into the models where the `User` process reaches the `UReplyRecvd` state. Both models ran with no violations on the safety as well as the progress properties. The results are shown in Table 1. As can be seen, the VIP generated code, although it required a larger state vector, resulted

in a significantly smaller state space. This can be attributed to two factors. First, the state encoding of the VIP generated model uses `goto` statements while the hand-built model uses an event loop construct in which control states were represented through variables. It was shown in [10] that this difference alone can account for a doubling in state space size. Second, the hand-built model uses global variables for all channels whereas, the VIP generated code declares channels as being local variables of the `Env` process. In [10] it was also shown that the use of global variables can reduce the effectiveness of the partial order reduction algorithm and thus also contributes to a larger state space.

To model the occurrence of events in the state-based model checker SPIN we used the previously described concept of remote referencing. For example, the remote reference:

```
#define p (GIOPAgent[5]@SRequestSent)
```

refers to a label `SRequestSent` introduced into the action part of the transition code within a state-machine in the `GIOPAgent` capsule. It corresponds to the state after the `SRequest` message has been sent.

For the hand-built GIOP model, ten high-level requirements (HLR) were formulated and verified in [11]. Of the ten requirements, two were explicitly checked on the VIP generated code for the GIOP model. Some other requirements were checked implicitly through the use of assertions. All requirements that were checked were exhaustively verified successfully on the VIP generated model. This serves to confirm that the required behaviors present in the hand-built model are also present in the VIP model and that the VIP-generated code does not cause a prohibitive state space size penalty.

6   VIP Implementation

VIP was implemented in the Java programming language using Sun Microsystem's freely available *Java Development Toolkit* version 1.2. This allowed us to achieve a highly portable tool while leveraging Java's extensive GUI support. In developing VIP we adhered to a strict model-view separation which has enhanced flexibility and reuse in the design. To achieve maintainability, a quintessential requirement in the academic environment in which VIP was built, all class structures have been documented in UML. The graphical editors used in VIP are based on a separately developed component called Nexus. Other components such as windows and dialog boxes are based on standard Java class libraries.

As discussed in [10], VIP contains a set of approximately 30 well-formedness rules, the majority of which are checked whenever a model component is changed as a result of changes in the view. An example rule is that *"... a connector can only connect Ports that are protocol compatible..."*

7   Conclusions

We introduced the VIP tool which permits the creation and editing of v-Promela models as well as the compilation of these models into Promela code. We showed that the resulting models are analyzable using standard SPIN model checking technology.

The current version of VIP supports many features of v-Promela. A major thrust in research and development effort will be needed to develop VIP into a comprehensive CASE tool for requirements and high-level design. First, the aspect of property specification is currently not supported very strongly. We hope that an incorporation of ideas stemming from the temporal logic specification pattern approach [3] and from graphical interval logics [12] will facilitate the specification of requirements. We will also design ways of relieving the user from having to build hooks inside the synthesized Promela code, for example by introducing labels, by allowing property formulae to refer to states and variables in the v-Promela model. Next, we plan to feed the SPIN validation results back into the VIP environment including an animation of simulation and error traces inside VIP[4]. We also intend to explore linking the v-Promela models to other model checking tools by suitable intermediate representations as for instance the IF representation [1]. The question of the different priority schemes for implementing transition code has highlighted the need for parametric semantics in order to remain compatible with other modeling tools and methods. We plan, in particular, to develop a set of semantic options that will allow analyzing models which have semantics identical to UML-RT. Finally, some concepts from v-Promela such as structural and behavioral inheritance as well as data object scoping have not yet been implemented and we plan to add these as well.

We hope that by reconciling an industrial standard visual modeling language like UML-RT with the input language of a model checker, and by providing suitable tool support we can contribute to increasing the acceptance of formal methods in the practical software engineering community.

Acknowledgements

References

1. M. Bozga, L. Ghirvu, S. Graf, L. Mounier, and J. Sifakis. The Intermediate Representation IF: Syntax and semantics. Technical report, Vérimag, Grenoble, 1999. 485

2. A. M. Davis. *Software Requirements: Objects, Functions and States.* Prentice Hall, Upper Saddle River, New Jersey, USA, 1993.   471

---

[4] The feasibility of this has been demonstrated quite convincingly in [9]

3. 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`. 485

4. Object Management Group. The Common Object Request Broker: Architecture and Specification. Revision 2.1, August 1997. 482

5. G. J. Holzmann and S. Leue. Towards v-Promala, a visual, object-oriented interface for Xspin. Unpublished manuscript, 1998. 472, 479

6. G. J. Holzmann and Margaret H. Smith. A practical method for the verification of event-driven software. In *Proc. ICSE99*, pages 597–607, Los Angeles, CA, USA, May 1999. invited. 471

7. G. J. Holzmann and Margaret H. Smith. Software model checking. In *Proc. FORTE/PSTV 1999*, pages 597–607, Beijing, China, October 1999. Kluwer. invited. 471

8. 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. 472

9. W. Janssen, R. Mateescu, S Mauw, P. Fennema, and P. van der Stappen. Model checking for managers. In *Theoretical and Practical Aspects of SPIN Model Checking, Proceedings of the 5th and 6th International SPIN Workshops*, volume 1680 of *Lecture Notes in Computer Science*, pages 92–107. Springer Verlag, September 1999. 485

10. M. Kamel. On the visual modeling and verification of concurrent systems. Master's thesis, University of Waterloo, 1999. Available from URL `http://fee.uwaterloo.ca/~m2kamel/research/thesis.ps`. 472, 479, 484

11. M. Kamel and S. Leue. Formalization and Validation of the General Inter-ORB Protocol (GIOP) using Promela and Spin. *Software Tools for Technology Transfer*, 1999. To appear. 482, 484

12. G. Kutty, Y. S. Ramakrishna, L. E. Moser, L. K. Dillon, and P. M. Melliar-Smith. A graphical interval logic toolset for verifying concurrent systems. In C. Courcoubetis, editor, *Computer Aided Verification, 5th International Conference, CAV'93*, volume 697 of *Lecture Notes in Computer Science*, pages 138–153. Springer Verlag, 1993. 485

13. S. Leue and G. Holzmann. v-Promela: A Visual, Object-Oriented Language for SPIN. In *Proceedings of the 2nd IEEE Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'99), Saint Malo, France*, pages 14 – 23. IEEE Computer Society Press, May 1999. 472

14. S. Leue and G. Holzmann. v-Promela: A Visual, Object-Oriented Language for SPIN. In *Proceedings of the 2nd IEEE Symposium on Object-Oriented Real-Time Distributed Computing ISORC'99*, pages 14–23. IEEE Computer Society, May 1999. 479

15. B. Selic, G. Gullekson, and P.T. Ward. *Real-Time Object-Oriented Modelling*. John Wiley & Sons, Inc., 1994. 472, 481

16. B. Selic and J. Rumbaugh. Using UML for modeling complex real-time systems. `http://www.objectime.com`, March 1998. 472

17. W3C. Extensible Markup Language (XML) - W3C Recommendation. `http://www.w3.org/TR/REC-xml`, February 1998. 473