Automating Checking of Models built using a Graphically Based Formal Modelling Language

Robert John Walters
Declarative Systems and Software Engineering Group,
Department of Electronics and Computer Science,
University of Southampton, Southampton, UK.
R.J.Walters@ecs.soton.ac.uk

Abstract

RDT is a graphical formal modelling language in which the modeller works by constructing diagrams of the processes in their model which they then join together to form complete systems. Aside from the benefits which accrue as a side effect of building a formal model of a proposed system, these diagrammatic models can be useful as a means of communication between the development team and the users. However one of the greatest benefits of a formal model is that it can be subjected to rigorous examination to ensure that it satisfies properties required of the system.

This paper describes the transformation used by the RDT toolset to generate Promela code (the input language of the SPIN model-checker) automatically from a model.

1. Introduction

As computer and other systems become larger and more complex we need to find methods which enable us to manage this complexity. A winning technique in other areas has been to break the problem into pieces and combine these into systems. In electronic hardware this approach has been spectacularly successful [2-6, 8, 10, 18]. The size of the pieces is a balance. Smaller pieces are easier to handle, but more difficult to assemble into a useful whole

There are two issues which need to be addressed when a system is constructed from components: we need to connect the components, then we have to get the behaviour we want. The question of how to make pieces of software fit together has been the subject of considerable effort and systems and schemes exist which address these issues (COM, EJB, RMI, MSMQ…) [11, 13, 17, 19-21]. Typically, these arrangements work by requiring components to conform to rules about how they interact with the others. Components are prevented from damaging each other [7] and constrained to perform interactions which should be understood. We see this technique applied in the physical world with things like the standardised physical plugs and sockets we use for various applications. The other problem is more subtle and difficult. We need to ensure that the assembled system behaves as required. Outside of software, this is often quite simple because the behaviour at the interface is simple.

Unfortunately, just being able to connect software components does not ensure that the resultant system will do what we want or expect and this is where models can help. They can give us answers to questions about the behaviour of systems before they have been built. However, to get these answers we need to build appropriate models and then analyse them. This analysis could be simple reasoning based on a diagram but, to be really useful, it needs to be more thorough - and for that we need models which have sufficient formality to permit analysis using techniques such as execution or model checking.

The marked reluctance of potential users of these techniques to get to grips with traditional text based formal modelling languages inspired the creation of the RDT modelling language [22]. In RDT, the modeller works with a tool to draw diagrams of their processes and how they are combined into complete systems. The RDT toolset includes an execution tool with which the modeller can experiment with the behaviour of their model, but if the modeller is to make assertions about the behaviour of a model confidently, the analysis needs to be more rigorous. This analysis could have been provided by building model checking into the toolset, but RDT uses the alternative approach of providing an automated translation of models into the input language of a leading model checker.

2. RDT in outline

RDT [22] is a graphically based formal modelling language. It is a small language which does not attempt to emulate the expressive power of more traditional modelling systems. Instead it provides a minimal collection of features inspired by the pi-calculus [14, 15]. The intention is for the language to be small enough for a new user to assimilate its essential concepts in a few hours whilst being powerful enough to describe useful models. A complete model in RDT comprises of a collection of processes which are connected and communicate using channels.

The behaviour of a process is described by a RAD-like diagram [16]. The events of a process are shown as squares and its states (which are named) are shown as circles. An event causes the process to move from one named state to another. An event is joined to the named state which precedes it by a line from above and to the state which follows it by a line from below. These lines are branched or joined as required. All processes start in a distinguished state called "initial".

In addition to the internal change of named state in the process, each is associated with a communication. There are three types, Send, Receive and Create. A Send is shown as a clear square and causes a named value to be placed into a channel. A Create event is a special case of a Send event distinguished in the diagram by a cross in its box in which a new value (or channel) is first created, associated with the local name used by the process for the value being sent and then sent on the channel. The final type of event is Receive which is complementary with the Send and Create events and is shown by a black square. It takes a value from a channel and associates it with the local name specified in the event. In contrast with the pi-calculus in which communications are synchronous, RDT permits the modeller to select the length of channels at runtime.

Figure 1 shows an example of a process description in RDT. To generate a process description, the modeller describes just the events in which the process takes part to the RDT model generation tool. The tool generates and displays the diagram each time an event is added or altered. The process shown represents that of a Barber in a traditional gentleman's barbershop. The process starts in the initial state from which it has choice of two actions. It may send a new value (which it refers to locally by the name, "MyCh") onto the channel it knows as "Custs". In doing so, the process moves to a state named, "Awaiting Instructions". From this state, the Barber receives instructions (from its customer) along the new channel "MyCh" and moves to the named state "Cutting". This is followed by a further pair of interactions concerned with obtaining payment. The Barber is then returned to its initial state and is ready to start again. As in a RAD, an RDT process description permits states which are re-visited to be re-drawn lower in the diagram. RDT uses "=" suffixed to a state name to highlight that the state appears in more than one location on the diagram. From the initial state, the process may alternatively follow the other path along which it places notifications on the channel it knows as "Info" that the Barber is taking a break followed by their return to work.

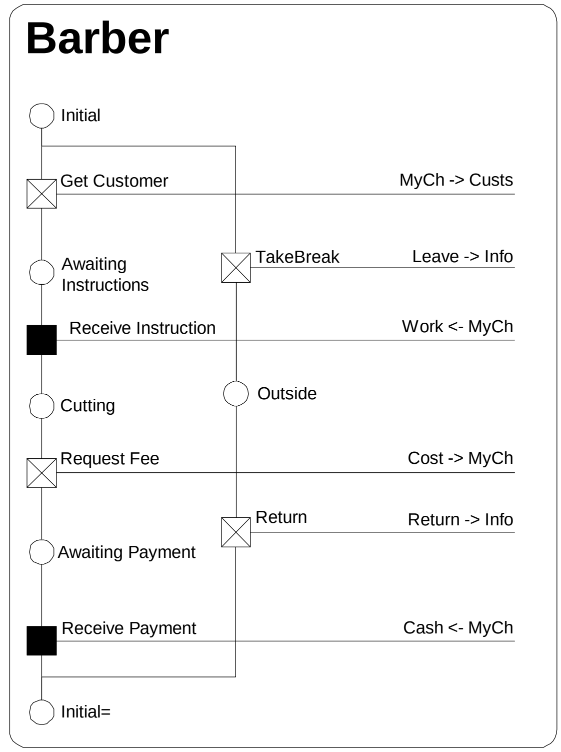

Figure 1: An Example process describing the behaviour of a barber

The second part of the description of a complete RDT system is the "model" diagram in which the modeller specifies a collection of processes and how the channel names they use are connected, if at all. As with the pi-calculus, the values passed along channels may be used as channels so that the initial connections between processes specified in this diagram may be supplemented and changed during execution of the model. Figure 2 shows an example of this type of diagram showing a model in which two instances of the Barber process (named Barber1 and Barber2), two instances of a Customer process (named Customer1 and Customer2) and one of a Sink process (named Info) are connected to form a complete model. To the right of the box for each of the processes, each of the channel names known to it is shown by a filled rectangle joined to the box. Associations or connections between these channels are shown by lines connecting them. To draw this diagram, the modeller uses the model generation tool again to specify the process instances they require and then the connections between them. The diagram is generated automatically by the tool each time the user makes a change.

Notice that RDT makes an important distinction between these two diagrams: the process diagram describes a *type* of behaviour. The "model" diagram deals with *instances* of processes and how they are interconnected.

In addition to the model generator, the RDT tools provide an animation tool in which a modeller may execute their model by hand (and a translation tool which performs an automated conversion into Promela).

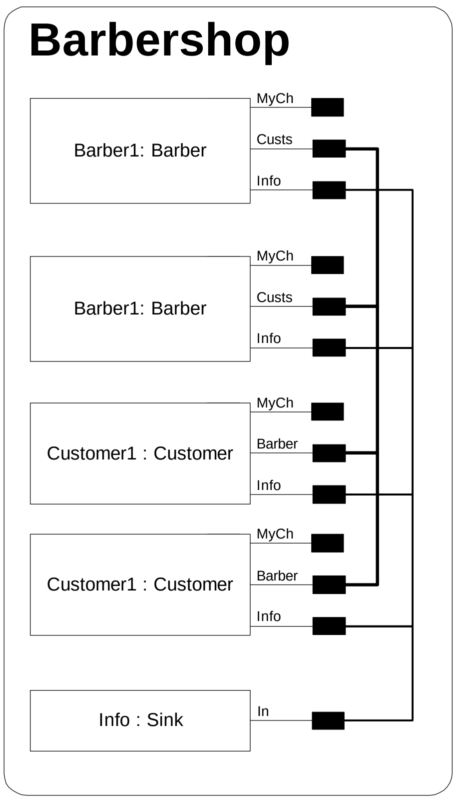

Figure 2: A Barbershop model with two Barbers and two Customers

3. Selecting a target Model Checker

If we are to model check our models using an existing tool, we first need to select a suitable target. The models described by RDT in its diagrams are finite state machines, so it would be possible to use any of the many model checking tools to analyse its models. However, two model checking tools stand out as potential candidates, FDR [1] and SPIN [10]. Both are mature, well established and respected systems, with attractive window based user interfaces, though they differ significantly in their input language and the way that the property to be verified is specified to the system.

FDR uses a variant of CSP [9] as its input language. The language is powerful and fully featured though it would not look familiar to a programmer. Its communication is synchronous along typed channels. By contrast, SPIN uses its own input language which has a syntax reminiscent of, but not the same as, the "C" programming language [12]. Communication in SPIN is also by typed channels, but permits the modeller to specify their length. After consideration, the SPIN model checker was selected for the following reasons:

- At some point the modeller may need to relate the code generated for the model checker to their model. It is felt that our target users are likely to be familiar with programming languages and so will feel more comfortable with the Promela code of SPIN with its superficial similarity to the "C" programming language than the process algebra inspired input language of FDR.
- Although the actual code required is potentially difficult to construct, the notion of giving the property to be checked to SPIN directly is likely to feel more natural to our target audience then the "refinement" based notion used by FDR.
- Promela channels have a more natural correspondence with the channels of RDT.

A final advantage of the SPIN model checker is that it is available free of charge for use on several platforms, including Windows, the platform on which the RDT tools run. This permits the curious potential user to experiment with this the tool without first making an initial financial commitment.

4. Translating from RDT to Promela in outline

Promela is a rich, expressive language in which to specify models for analysis with SPIN so there will be many possible representations of RDT models in the language. This paper describes the one which is performed automatically by the RDT tools.

The description of an RDT model is made in two parts and the translation follows the same pattern. First, each of the processes is transformed into a Promela processes and then a collection of instances of them need is assembled into the completed system as specified in the "model" diagram.

4.1. Processes

During execution of a model by the RDT execution tool, as each event occurs each of the processes in the model reconstructs its list of available events. Whether an event is available depends on the present state of the process (instance) concerned and the willingness of the channel the event interacts with to accept the write or read associated with the event. This suggests a structure for a Promela description of one of our processes as a Promela process with a variable to record its state and a single "do" loop with each branch representing one of its events. Each branch of the loop would be "guarded" by a conditional statement dependent on the current value of the "state variable" of the process and the availability of the required communication. However, this scheme is unsatisfactory for two reasons in particular:

- SPIN would consider such a loop to be a single statement. Consequently it would regard a process created in this manner as having a single statement and one which performed even a single event would appear to SPIN as one which had been thoroughly exercised.

- Promela does not have a string type, so the state of the process would have to encoded into a numeric form which would make interpretation difficult for the human reader. (Promela does have "symbolic constants" which could be used, but just one declaration of this type is permitted in each file, so if it were used, all of the states of all of the processes would have to be declared in a single collection.)

The solution adopted is to use labels and explicit "goto" statements which are permitted in Promela. Each of the labels in the code for a process description corresponds to a named state of the RDT process it represents. Using the process state names for these labels eases the task of relating the automatically generated code to the diagram of the process in RDT.

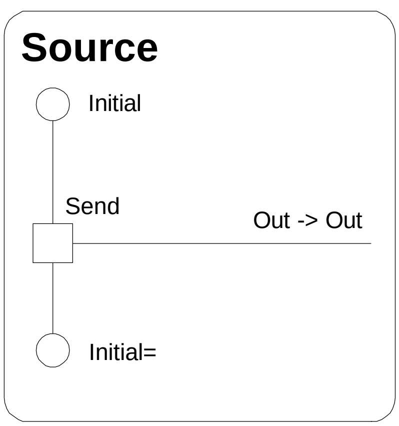

Figure 3: A “Source” process in RDT

Each of the labels in the process corresponds with one of the named states of the RDT process and is normally followed by an “if” statement. Within this statement, there is a branch for each of the possible events which can follow this named state in the RDT process diagram. Each branch starts with an expression which performs the communication associated with the appropriate event followed by a “goto” statement taking execution to the labelled point in the description corresponding to the “after” state of the chosen event, the new named state of the process. In the case of a state which is not the before state of any event, the process is unable to proceed further and the “if” statement is replaced with “skip”.

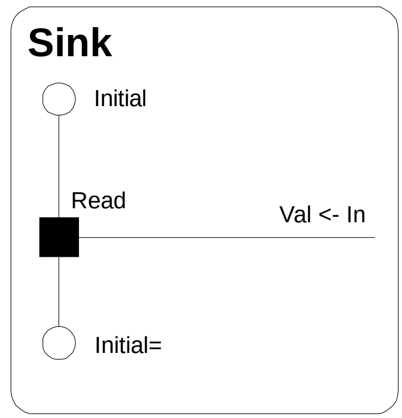

Figure 4: A "Sink" process in RDT

Figure 3 and Figure 4 show two elemental processes in RDT. The process in Figure 3 Sends a value (the name of the channel it is writing to) on a channel it knows as “Out” repeatedly. Figure 4 shows a complementary process which repeatedly reads a value on the channel it knows as “In” which it stores (and is later able to refer to) as “Val”. Using the translation outlined above, these two RDT process descriptions can be transformed into the Promela code shown in Figure 5

```
proctype Source(chan Out)
{
initial:
if
:: Out!Out; goto initial;
fi;
}

proctype Sink(chan In, val)
{
initial:
if
:: In?Val; goto initial;
fi;
}
```

Figure 5: Promela Code for simple Source and Sink Processes

4.2. "Models"

With the transformation of the process descriptions into Promela complete, code is then required to assemble instances of these into the complete system specified in the RDT "model" diagram. The technique adopted was to construct the system required in an "init" process. Where a process with the distinguished name "init" is defined in a Promela file, SPIN's first action on starting the model is to create a single instance of this process and set it running.

Each RDT process has a number of names for channels. Each of these may need to be associated with a channel at the start of execution as a consequence of being connected to another in the "model diagram", but there is no requirement for this to be the case. It is not an error for at least some of the channel names known by a process not refer to a channel initially since they may become associated with channels during execution as a side effect of read and create events.

However, in Promela channels need to be declared (like variables in many programming languages). These declarations may be global, within the process or (declared elsewhere and) passed to the process as a parameter. Where a connection exists at the start of execution, the required channel is declared in the "init" process and then passed to the connected processes as a parameter. This leaves the question of how to handle channels for which processes have names, but are not connected at the start of execution. They need to be declared since otherwise SPIN generates errors. Declaring these channels as local variables within the process is problematic because it requires knowledge of the connections made in the "model" part of the system description to be applied to the general descriptions of processes. It would also mean that coping with a "model" in which different combinations of channels known to a process are the subject of connections in different instances of that process, probably requiring multiple versions of the process – one for each arrangement of initial connections. The solution adopted is not to declare any channels within process descriptions. Instead, all are passed as parameters to the process. Where a process has names which are not initially connected, it is supplied with placeholder channel names. The "init" process generated for a particular RDT "model" is generated as follows:

1) Channels of the required length are declared for each of the required connections between the process instances.
2) Placeholder channels are declared to be supplied to process instances as placeholders for any channel names they know, but are unconnected initially. Separate channels are needed for each such parameter of each process instance to guarantee that no communication can occur on these channels inadvertently.
3) Process instances are brought into existence by a sequence of "run" statements.

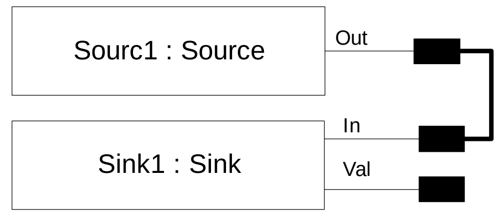

Figure 6: A model showing a Source process with its "Out" channel connected to the "In" channel of a Sink process

All of the statements in the "init" process are enclosed in an "atomic" statement to instruct the model checker to execute them all as if they were a single indivisible action. This ensures that the whole of the model system and its (initial) interconnections are in place before any part of the system starts to operate.

Figure 7 shows the code generated from the RDT model shown in Figure 6. Since the RDT notation itself is silent on the length of channels, the RDT model execution tool elicits this information from the modeller at runtime. This information is also needed when the model is translated to Promela so it is also elicited from the modeller by the translation tool. The chosen value (4 in the example) is defined as a constant at the start of the generated Promela code to permit the modeller to change the channel length easily without regenerating the whole file. Notice also that a channel (ch0) is created and passed to both processes to make the connection shown in the diagram and that, since the channel known to the process "Sink1" as "Val" is initially unconnected, a placeholder channel (nch0) is declared and passed to the process.

```
#define CHLEN 4
chan ch0 = [CHLEN] of {chan};
chan nch0 = [0] of {chan};

/* Process definitions here */

init
{ atomic {
run Source(ch0);
run Sink(ch0, nch0);
} };
```

Figure 7: The "init" process

4.3. Channels and Values

Communication is RDT is inspired by the pi-calculus [15] in which there is just one type of value (referred to as a "name"). RDT takes the same view: values passed in communications are all of the same type. In some contexts, a value passed between processes may represent a value such as the result of a computation. In others the value passed may be a channel which may be used for later communications. It is this ability to pass channel typed values along channels which permits the dynamic re-configuration of RDT models.

In contrast with the pi-calculus and RDT, Promela channels are typed according to the kind of values they carry. One of the permitted types of value that a Proemela channel is permitted to carry is a channel and, since potentially an RDT process may use any value it knows as a channel, it is this type of channel which is used throughout the Promela code generated from an RDT model.

5. Issues

Two issues remain which have not been addressed in the transformation described so far. The first concerns a difference between the acceptable use of the "Read" event in RDT and the action of reading from a channel in Promela. The second concerns the "Create" event in RDT. This event is useful as it permits processes to create the new channels needed to create new connections between processes at runtime (although they may also be used just as simple values).

5.1. A Special case of a Read event in RDT

RDT permits a process to read a value on a channel and assign the name received to the name used as shown in Figure 8.

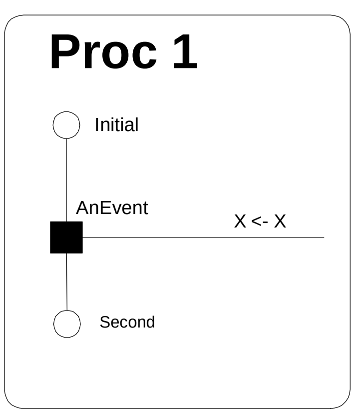

Figure 8

According our scheme for translating models to Promela, this would cause the following code to be generated:

```
if
:: X?X; goto second;
fi;
```

Unfortunately, "x?x" causes SPIN to generate an error, so the tool generates the following alternative code where necessary:

```
chan tmp;
...

if
:: atomic{X?tmp; X = tmp; } goto
second;
fi;
```

5.2. "Create"

The problem deriving code for the behaviour of the "RDT" Create event is not so easily addressed. An interim workaround has been implemented in the translation tool by which any process which contains a create-type event is given a supply of channels. The process then allocates a channel from this supply whenever it needs one for a "create" event. When the supply is exhausted, the process will be unable to carry out another "create" event. This supply of channels is declared as part of the description of each process. So long as the number of channels in this supply is sufficiently large in the context of the model, this solution does not impact on the behaviour of the model. (The size of this cache of channels is elicited from the modeller at the same time as the channel length.)

A complete solution to this problem which is not yet implemented would be based on the following observation:

In an RDT model, each process knows some number of channels which it refers to using its own collection of local names. The assignment of these channels to names changes at runtime when a process reads a channel or uses "Create" to generate a new channel – and if the name to which the new value is assigned already refers to a channel, the existing value is overwritten. A consequence of overwriting channel names is that, unless the process has taken explicit steps to prevent it, knowledge of the overwritten channel is lost at the same time. Processes in RDT are unable to locate channels by any method other than being told of them by other processes (and creating new ones). Consequently, should a channel ever reach a condition where none of the executing process instances has it associated with any of their names, the channel is irretrievably lost to the model and the system could safely destroy that channel (together with any values stored in it).

Since, for a channel to be used by a process instance, it must “know” the channel by having it associated with one of its channel names, no running RDT model can possibly have more channels in use than there are local names for them in all of the process instances of the model. Consequently, the translation tool could generate code which, by the reclaiming of channels which are no longer visible to any of the process instances could guarantee to always have a channel available to allocate to a process which sought to perform a “Create” event. (A complete implementation of this scheme would need to note any values found in recovered channels as their presence may be an indication of a fault in the model.)

6. Conclusion

The RDT modelling language together with its model generation and execution tools demonstrates that it is possible to construct useful formal models using a graphical idiom in place of the usual text based input. However, to make the best use of these models, their behaviour needs to be much more rigorously examined than the modeller can hope to achieve by hand using an execution tool. This might have been achieved by the construction of a model checking tool to supplement the existing RDT tools. However, model checking software is already available which is known to be accurate, powerful and efficient so it was felt that a better approach would be to find a translation which could transform an RDT model into a form suitable for input into an existing model checker.

The model checking software chosen was SPIN with its programming-like input language, Promela. The motivation in the development of RDT is to make formal modelling as easy as possible for the inexperienced user so, the translation of an RDT model into Promela code had to be performed automatically. We cannot expect the user to apply the transformation manually. At the same time, the transformation has to be into Promela code which is sufficiently readable for the modeller to be able to identify its relationship to the original features of the RDT model.

The transformation described above can be performed mechanically and has been implemented in a tool which is able to take a model built using the RDT model generation tool and transform it into correct Promela code automatically. Using this code, the modeller is able use SPIN to perform “standard” analysis (e.g., unreachable code and deadlock detection) of their model without learning the syntax of Promela and with an absolute minimum of knowledge of SPIN itself.

7. References:

[1] "FDR2 User Manual,".: Formal Systems (Europe) Limited, 2000.

[2] M. Barjaktarovic, S.-K. Chin, and K. Jabbour, "Formal Specification and Verification of Communication Protocols Using Automated Tools," presented at First IEEE International Conference on Engineering of Complex Systems (ICECCS'95), Fort Lauderdale, Florida, USA, 1995.

[3] B. Beizer, N. Juristo, and S. L. Pfleeger, "Cleanroom process model: A critical examination," *IEEE Software*, pp. 114-118, 1997.

[4] E. M. Clarke, O. Grumberg, and D. E. Long, "Model Checking and Abstraction," *ACM Transactions on Programming Languages and Systems*, vol. 16, pp. 1512-1542, 1994.

[5] A. M. Gravell and P. Henderson, "Executing formal specifications need not be harmful," *IEE/BCS Software Engineering Journal*, vol. 11, 1996.

[6] O. Grumberg and D. Long, "Model Checking and Modular Verification," *ACM Transactions on Programming Languages and Systems*, vol. 16, pp. 843-871, 1994.

[7] P. Henderson, "Laws for Dynamic Systems," presented at International Conference on Software Re-Use (ICSR 98), Victoria, Canada, 1998.

[8] P. Henderson and R. J. Walters, "System Design Validation Using Formal Methods," presented at Tenth IEEE International Workshop on Rapid System Prototyping (RSP99), Clearwater, Florida, 1999.

[9] C. A. R. Hoare, *Communicating sequential processes*: Prentice-Hall International, 1985.

[10] G. J. Holtzmann, "The Model Checker SPIN," *IEEE Transactions on Software Engineering*, vol. 23, pp. 279-295, 1997.

[11] IBM, "MQSeries Family,"., 2001.

[12] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*: Prentice Hall, 1988.

[13] Microsoft, "Microsoft Message Queuing Services,".: Microsoft, 2001.

[14] R. Milner, *Communication and Concurrency*: Prentice Hall, 1989.

[15] R. Milner, "The Polyadic pi-Calculus: a Tutorial," in *Logic and Algebra of Specification*, F. L. Hamer, W. Brauer, and H. Schwichtenberg, Eds.: Springer-Verlag, 1993.

[16] M. A. Ould, *Business Processes - Modelling and Analysis for Re-engineering and Improvement*: John Wiley and Sons, 1995.

[17] D. S. Platt, *Understanding COM+*: Microsoft Press, 1999.

[18] K. Sullivan, J. Socha, and M. Marchukov, "Using Formal Methods to Reason about Architectural Standards," presented at 19th International Conference on Software Engineering, Boston, 1997.
[19] Sun Microsystems, "Enterprise Java Beans,".
[20] C. Szyperski, *Component Software*: Longman, 1998.
[21] A. Thomas, "Enterprise JavaBeans Technology," Patricia Seybold Group, White Paper prepared for Sun Microsystems Inc December 1998.
[22] R. J. Walters, "A Graphically Based Language for Constructing, Executing and Analysing Models of Software Systems," presented at 26th Annual International Computer Software and Applications Conference (COMPSAC 2002), Oxford, 2002.