UNIVERSITY OF Southampton

University of Southampton Research Repository
ePrints Soton

Copyright © and Moral Rights for this thesis are retained by the author and/or other copyright owners. A copy can be downloaded for personal non-commercial research or study, without prior permission or charge. This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the copyright holder/s. The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the copyright holders.

When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given e.g.

AUTHOR (year of submission) "Full thesis title", University of Southampton, name of the University School or Department, PhD Thesis, pagination

http://eprints.soton.ac.uk

University of Southampton
Faculty of Engineering and Applied Science
Department of Electronics and Computer Science
Southampton SO17 1BJ

A Graphically based language for constructing, executing and analysing models of software systems

by

Robert John Walters

A thesis submitted for the degree of Doctor of Philisophy

December 2002

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING AND APPLIED SCIENCE

ELECTRONICS AND COMPUTER SCIENCE

Doctor of Philosophy

A graphically based language for constructing, executing and analysing models of software systems

by Robert John Walters

With computer systems becoming ever larger and more complex, the cost and effort associated with their construction is increasing. Consequently, it is more important than ever that the developers understand how their systems behave if problems are to be avoided. However, acquiring this understanding is a problem: the systems are sufficiently complex that developers need help to analyse and understand them and, at the time this analysis is most crucial, the system is unavailable because it has yet to be built.

We already have maturing technologies which address issues associated with the interconnection of software components at the procedural level, but they do not address issues related to the behaviour of these compound systems. Formal, executable models can help here by providing developers with a platform on which to establish the feasibility of a proposed design for a system. However, commercial developers seem reluctant to employ this type of modelling in their design activity.

This report describes a new modelling tool in which the traditional model generation technique of writing "programming language like" code is replaced with a model generation tool which uses a graphical representation of models. Despite appearing informal, the system retains sufficient formality to permit the models to be executed using the tool, or converted into code for analysis by a traditional model checking tool.

Acknowledgements

I would like to acknowledge the help, support and encouragement which I have received from family, friends, colleagues and others whilst pursuing this work. In particular, I am indebted to my supervisor, Peter Henderson for his support and encouragement over the years.

Of the others, I hesitate to name names in case I should miss someone out. Let me just say this: you know who you are and I appreciate your help and support.

R J Walters.

Contents

List of Figures

Chapter 1 Overview

As computer and other systems continue to become larger and more complex we need to find methods which enable us to manage this complexity. In most theatres where complexity is an issue, a winning technique has been to break the problem into pieces and then create the required solution by combining a collection of these pieces. Electronic and computer hardware is an area where this type of approach has been spectacularly successful. The size of the pieces is a balance. Smaller pieces are easier to understand and handle, but require greater effort to assemble into a useful whole.

This technique has been deployed very successfully in many areas and we are seeing its adoption in software development as the need arises to manage the complexity of systems more effectively. However, because computer systems inhabit a world which is devoid of physical laws, they are not subject to the constraints on variety and complexity which apply to most other systems. As a consequence, making software systems from collections of components is proving to be more difficult than in other disciplines.

1.1 Why build executable models?

There are two issues which need to be addressed where a software system is to be constructed from a collection of components. First there has to be a way to connect the components together. Then we have to get them to do what we want. The matter of making pieces of software which will fit together has been the subject of considerable effort and systems and schemes exist which address these issues (COM, EJB, RMI, MSMQ…) [Object Management Group, Sun Microsystems, JavaSoft 1996, Allen and Garlan 1997, Box 1998, Gray, Hotchkiss et al. 1998, Sessions 1998, Thomas 1998, Platt 1999, Microsoft 2001]. Typically, these arrangements work by managing and controlling the interfaces between components (as well as providing some underlying support). Examples of analogous schemes in hardware include the PCI bus or the standards which we apply to do-

mestic wiring. By forcing components to conform to rules about how they interact with the outside world, these systems ensure that components do not damage each other when they are connected and perform interactions which each should understand. To a large extent, this problem may be addressed by managing component interfaces and ensuring that components are only connected through compatible interfaces. We can see this type of consideration being applied in the physical world, in things like the standardised plugs and sockets we use for domestic electricity installations and other applications.

The other problem is more subtle and difficult. We need to ensure that the assembled system does what is required [Gravell and Henderson 1996, Hoare 1996]. Outside of software, this part of the problem of building from components is often much simpler than matters concerned with a physical interface. For example, the interface between a domestic appliance and the electricity supply actually only does one simple action (supplying power) so all that matters is that the connection is made between compatible interfaces. Although the details are more complicated, the "meaning" of the connection between devices made using a PCI bus is usually obvious too.

Unfortunately, where the components are pieces of software, it is not the case that collecting together all the parts we need to solve our problem and connecting them together respecting the rules associated with their interfaces in a system will ensure that the resultant system will do what we want or anything at all [Garlan, Allen et al. 1995]. Put another way; just because the interfaces between two components permit them to be connected is no assurance that they will interact properly or even that making the connection makes any sense at all. A good example of this is from the physical world is a power extension lead. The plug on one end of the lead is a perfect fit into the socket on the other end, but does fitting them together make any sense? Probably not.

One approach to solving this problem is to enhance a component's interfaces with requirements about their use [Luckham, Vera et al. 1995]. This approach has

shortcomings: even for small components decorating interfaces in this way is an enormous task and would prevent the component from being used for a purpose which was not envisaged when it was designed. And yet, one of the benefits which can accrue when systems are built using component technologies is the innovative use of components (as part of the solution to a problem which was not envisaged when the component was designed). Taking the example of the power extension lead again, if we are not interested in supplying power and just want keep the loose ends tidy, then plugging the two ends together might well be the thing to do even though it makes no sense in the context for which the article and its "interfaces" at either end were originally designed.

The real solution to this problem is to reason about and check the behaviour of a system and the components from which it is built [Hashmi and Bruce 1995, Magee, Dulay et al. 1995, Ip and Dill 1996, Magee and Kramer 1996, Henderson 1997]. Obviously, the definitive answer to the question, "if we put these components together like this, will it work?" can only be obtained by doing it and trying the completed system. But, if we are to avoid wasting time and effort, we want to know the answers before we build when the real system is not yet available. So we have to use something else and this is where models can help [Clarke, Burch et al. 1991, Clarke, Grumberg et al. 1994, Grumberg and Long 1994, Barjaktarovic, Chin et al. 1995, Gravell and Henderson 1996, Beizer, Juristo et al. 1997, Holzmann 1997, Sullivan, Socha et al. 1997, Hartel, Butler et al. 1999, Henderson and Walters 1999]. These models do not need to replicate the behaviour of the whole system. All they need to do is to represent the particular aspects of the behaviour of the system which we are concerned to check. This means that these models can abstract away unimportant detail which in turn reduces their complexity and the amount of time and effort required to construct them [Jackson 2002]. However, we do need to be able to analyse these models to ensure both that they display the required behavioural features and that they are free of undesirable behaviours. This analysis could be simple reasoning based on a diagram but, to be really effective, it needs to be more thorough - and for that we need models which have some element of formality to them. On occasions, it may be deemed neces-

sary to apply proof techniques to the project [Chin, Faust et al. 1995, Butler 1997], but often something less rigorous like execution or model checking is more than acceptable [Coffman, Elphick et al. 1971, Clarke, Grumberg et al. 1994, Henderson and Walters 1999].

1.2 Desirable features of a modelling system

Modern software systems are highly complex, typically built from a collection of components [Hawley 1999]. Some, if not all of these components will be sufficiently complex for a developer to have difficulty understanding their entire behaviour fully. Combining them creates a system which is so complex that the developer needs help with the task of analysing it for correct behaviour. Building models which are sufficiently formal for them to be executed and subjected to examination with model checking tools is one technique which can help. And yet, getting working software developers to build formal models of their systems as part of their design effort is like getting smokers to give up. In discussion with either group, they readily agree that the proposed action is a good idea and that they would benefit from doing it but they don't act.

The reason why developers are so apparently reluctant to add (formal) modelling techniques to their design activities is not easy to identify. However, most existing modelling systems are daunting for the novice to use and this is certainly a factor.

The problems seem to stem from three areas:

- Most model checking tools require the model to be constructed by writing a form of code which is specialised to the tool. This means the potential modeller needs to learn what amounts to a new programming language syntax and semantics before they can start to use the tool.
- The features and style of formal modelling systems appear to be motivated by the creators' desire to maximise the expressiveness or elegance of their language.

- Novice modellers have difficulty interpreting the meaning of textual descriptions of systems and applying this to their existing understanding of a system. The success and popularity of UML would appear to support the assertion that this barrier is lowered considerably when the models are presented as diagrams rather than text.

For a modelling system to appeal to the broader community of software developers, it should have the following features:

- An interface which permits the novice user to build "basic" models quickly and easily without a significant "up-front" investment of time and effort learning a new programming language
- Since the medium is so appealing to experts and novices alike, it should be able to present its models in a diagrammatic form.
- It should have an approach to model construction which has an obvious sympathy with that which will be used in the implementation of the system to be modelled.
- It should permit the modeller to proceed from constructing their model to performing an analysis of it smoothly, at least for some elementary analysis.

There can be no doubt that formal methods have a reputation for being hard to use. If we are to see them break out of specialised applications into widespread use, this perception needs to be addressed and dispelled. Unfortunately for the "mainstream" software developer, this reputation is probably justified for the more mature formal methods and their tool support. A new user approaching one of these methods and its support tools for the first time will have to invest a significant amount of time and effort getting to grips with the concepts which underlie the operation of the tool. Only after this has been completed can they begin to understand how using the technique might fit into their design and development process and what benefits it can bring. Usually they will also have to learn the syntax of the model input language too. This "up-front" investment in time and effort is enough to discourage many potential users from investigating the possi-

bilities. This investment in initial familiarisation may be reasonable and necessary if the full benefits of these tools are to be realised. However, developers don't always need all of this power and they would be more easily tempted to investigate formal methods if some of this initial effort could be avoided, even if this reduced the scope and variety of the analysis which could be applied.

Related to the issue of the high entry cost of using formal methods is the appeal of pictorial representations of models and processes to potential users. Potential model builders seem to find a graphical representation of their model which is generated by the support tool considerably assists them in satisfying themselves that their code accurately describes the system they wish to examine. Also, where features of a model have to be described to people outside the mainstream of the system development activity (such as clients who are unlikely to be familiar with the technical details of the development of their product), this task is greatly eased if the model can be presented as a diagram instead of text.

Another difficulty for the inexperienced user of formal modelling systems is to understand how the communications paradigm of the modelling system maps onto the inter-process communications which is to be used in the implementation of the system. Whilst component systems broadly based around procedure calls continue to be popular, there is a noticeable move away from these synchronous mechanisms towards asynchronous systems such as message passing middleware products [Sun Microsystems, Dickman 1998, IBM 2001, Microsoft 2001] and web based products ["Universal Description Discovery and Integration (UDDI)), Technical White Paper" 2000, Christensen, Curbera et al. 2000, Microsoft 2000]. The potential difficulty which this generates for developers new to formal model building is that they don't see the relationship between the event sharing communications paradigm preferred by the majority of formal modelling languages and their own, channel based, understanding of communications within their system implementations. The shared event communications paradigm feels unnatural to these people when they encounter it for the first time. It is not that the realisation that there is a simple equivalence between event and channel based communica-

tions (for synchronous models), is particularly difficult. Instead it is that, for many, it is not immediately obvious and adds a further conceptual burden for the developer building their first few models. Where communication in the implementation is asynchronous, the situation is more difficult. If it is important for the model to match the proposed implementation by using asynchronous communication, this can be achieved easily by the expert. All that is required is for appropriate buffer processes to be inserted between communicating parties as required. However, with these addition processes included (aside from providing extra opportunities for errors to creep in), the model looks less like the implementation adding to the difficulty of understanding the mapping from the model to the implementation.

The modelling language described in this document attempts to address these issues, though to enjoy the advantages of the language fully, the user needs to work the language together with suitable support tools. Together with the support tools, the language permits the creation, execution and limited further analysis of models without the need for writing code. These models are presented as diagrams which are designed to be understandable even by observers with minimal knowledge.

In addition to supporting the creation, modification and interactive execution of models, the example support tools described also provide the modeller with easy access to checks on their work for features such as deadlock, unreachable code and unsatisfactory end states by providing an automated translation of models into the input language of an industry standard model checking tool. A modeller may apply the more advanced features of the model checker to perform further analysis of their model including discerning if the model satisfies arbitrary properties (defined by the modeller) using the automatically generated code as a starting point for this further investigation. However, even if they do not pursue analysis of the model, building it will have been worthwhile if it helps the modeller/developer to formulate a system design which is free from deadlocks and unreachable code [Kaveh and Emmerich 2001]. The motivation for the language is that building and analysing formal models as part of the design processes is a worthwhile effort,

even if the extent of the analysis is strictly limited and seeing the benefits the application of a lightweight version of this type of technique might tempt developers to look further into the area.

Chapter 2 History and background

2.1 Background

There continues to be some debate about exactly when the computer was invented with competing claims from at least two machines, but regardless of the reality, it is not much more than fifty years since the first machine was built. In that time, computer hardware has progressed at a breathtaking rate. Computers today are so reasonably priced that they have become a commodity item which almost anyone can afford whilst at the same time offering features and performance which would have been unthinkable even just a few years ago. It is a testament to the success of our hardware engineers that we now have machines small enough to carry in a pocket and have many times the capability of a machine which would have occupied a large room in the 1960's. However, although there can be no doubt that the software we use today is better than what we have been accustomed to in the past, it would be hard to claim that software has progressed as fast as the hardware on which it runs. For example, five years ago a 200Mhz computer was considered to be a high specification machine. Today the norm is in excess of 2Ghz and 200Mhz machines are being discarded as too slow to be useful. Yet, whilst there have been improvements, the software most of us use on these machines is not noticeably different.

There is another major difference between the development of hardware and software: reliability. We are used to the idea that hardware works exactly as it should and we expect it to run for months if not years without problems. With the present rate of advancement in hardware, this means that much hardware is discarded as being obsolete long before it develops any faults. At the same time we accept that software will contain errors (bugs). This disparity in our tolerance of errors is well illustrated by the public reaction to the discovery of a minor problem in the floating-point calculations of early Intel Pentium processors [Edelman 1997]. Although this was a relatively minor fault which only affected a particular class of operation – and even then only in limited circumstances, there was a scandal. It

was headline news. Most owners demanded upgraded replacements regardless of whether the problem was likely ever to affect them. At the same time, bugs were being routinely discovered and reported by users of all sorts of software – often the users just found ways to work around them and didn't even bother to report them to the developers.

This apparent disparity in the performance of software and hardware systems may be more apparent than real because, although they are impressively large, there is not the same enormous variety in the function and behaviour of hardware systems as there is in software systems. With few exceptions, modern computers are general purpose machines built according to the "von Neuman architecture" [Aspray 1991]. The adoption of the general purpose computer was made possible by the insights of Turing [Turing 1936] and Church [Church 1936] who (independently) demonstrated that a general purpose computer was able to perform any task which is possible for any computer. This adoption of the general purpose computer has assisted the hardware engineers by enabling them to concentrate their efforts onto a single class of machine.

Hardware engineers also have other advantages over those building software such as the fact that, at least to some extent, the software engineers have to wait for the hardware to be built before they can start their work and typically hardware engineers do not have to face changing demands from the ultimate end users. Certainly it appears that, so far, the hardware engineers have been better able to handle the enormous complexity of modern computer systems than their counterparts in software engineering.

There are many factors which make the task of building large software systems difficult. These include relative immaturity of software engineering and the way that the size and complexity of the systems seems to cause us trouble at many levels. One feature which sets software systems apart from most other types of system is the environment in which they operate. They live in a "virtual world" which imposes few, if any constraints on their activities. This freedom permits us

to construct systems in software which would not be possible in hardware. However the price we pay for this lack of constraints is increased complexity. Even specifying how we expect systems should behave seems to be problematic.

Today's software systems can be very large indeed. Systems which are compiled from millions of lines of source code are by no means exceptional. With software systems this big, it is impossible to perform exhaustive testing on them and the best we can hope to achieve in testing is to check the bulk of the most likely execution patterns. Even this is fraught as there is no simple way to ascertain the environment in which a piece of software will be used. Consequently the actual pattern of use may differ significantly from that expected during development. It is clear that we cannot hope to build and test (or otherwise evaluate) such systems in a single unit.

If we look at the success of the hardware developers, we see that one of the key features of their technique for building big systems is construct useful pieces and then use these components to build progressively larger components until they have their complete system. It seems reasonable to expect that building software systems from components should bring similar benefits. In some senses, we do use components in software systems extensively already. For example, a desktop system comprising an operating system plus several software applications could be considered to be built from components but often only one of these programs is executing at any moment and the interactions between them are mostly limited to simple, well defined interactions between the operating system and one of the applications. We are beginning to see more components being used in the construction of software systems [Szyperski 1998] but this is a style of development which is not yet as widespread as in hardware.

Despite their advantages, components bring problems too: how do we put them together? How do we ensure they do what we expect and want? Much of the difficulty of the first of these questions arises from the "virtual" nature of the world in which our systems operate and has been the subject of considerable effort in

recent years. Amongst other things this has led to the creation of systems and schemes which address these issues. These include network communication protocols (TCP/IP), and message formatting standards (HTML, XML) as well as component frameworks (COM, CORBA & others), message passing middlewares (MSMQ, WebShpereMQ). These frameworks address the issues related to how to put components together. They are very good indeed and it is tempting to suggest that in principle, if not in practice this problem has been solved. It is easily possible to create a re-useable software component which interacts with other components by making appropriate use of these frameworks. They could be considered as analogous to hardware standards like PCI or USB. However, these frameworks deal almost exclusively with matters related to how to connect components together: they attempt to guarantee that components are *able* to communicate without destroying one another in the process. In some form or another, they are concerned with "interface management": the form communications should take, the type and order of parameters in procedure calls, the type of response to expect. They do not address issues related to whether, having been connected, the components actually will interact or what the effects of that interaction might be.

Often, when making physical connections in hardware this is not a problem. Compared with the equivalent situation in software, the purpose of the connection between components is often simple and obvious so that difficulties don't arise. For example the interface between a domestic appliance and the electricity supply only does one simple action (supplying power) so that all that matters is that the connection is made via compatible interfaces (shape of connector, voltage, etc.). Understanding the "meaning" of the connection and ensuring that the appliance and the supply behave appropriately is not an issue. This applies to a greater or lesser extent in most hardware systems. For example, although the details are more complex, the "meaning" of the connection between devices made using a PCI bus is usually obvious.

The question of what the behaviour of a collection of connected software components will be is more subtle and difficult because the behaviour of software com-

ponents is much more diverse. It seems reasonable to suppose that maturing standards and frameworks will help with some of this complexity in the same way that they have in hardware, but the problem needs to be addressed if we are to gain the real benefits of building software systems from pieces (components).

One approach is to press interface technology which has been so successful in the area of providing the connection infrastructure into service in this area too. This might be achieved by enhancing the definitions of the interfaces of a component with additional elements which attend to the behavioural aspects of the interactions offered through the interface as well as the more practical matters such as parameter and return values and types. This "decoration" of a component's interfaces could be an enormous task as it would be necessary to encode into it every possible behavioural pattern into which the component was prepared to take part. A simple example might be for a component to insist that another perform some form of registration and initialisation before permitting access to other functions. Whilst encoding all of these requirements into the interfaces of components is a huge task and brings some benefits, it would certainly bring with it one very serious drawback: by preventing unforeseen use of components with restrictions in their interfaces we would prevent innovative use of components and stifle creativity in their use. This is much more of a problem in the world of software than in other areas where components are in widespread use because of the variety and complexity of the interactions which occur between software components. This approach is being seriously pursued by some researchers.

An alternative which is advocated in this work is to accept the limitations of the interface model used in the component frameworks and adopt a different approach to addressing behavioural issues, modelling. The task of connecting the components of a system built from a collection of parts is divided into two. Matters relating to the actual connection of components, the "physical" connection, are handled by (the constraints imposed by) some form of middleware. Matters concerned with what the interconnections "mean" and the behaviour of the completed system are considered separately. This examination and consideration of system

behaviour is a part of the design process so to make any sense it cannot be performed on the final system which is not yet built.

Modelling (and simulation) is another technique which has enjoyed enormous success in the world of hardware development. There are mature hardware description languages and simulation tools with which hardware engineers can build impressive representations of systems under development [Jebson, Jones et al. 1993, IEEE 1994, Hodgson and Hashmi 1997, Carpenter and Messer 1998, Hashmi 1998, Hashmi 1998, Siegmund, Muller et al. 1998]. Using these systems, hardware engineers are able to experiment with and refine their designs much more rapidly and cheaply than they could hope to do working with prototype hardware. Modelling and simulation is immediately attractive in hardware development as it permits potential designs to be examined and evaluated without the considerable trouble, time and expense of building a prototype implementation. In the software development environment, it is tempting to imagine that, since they are both software, a model of a system and the software product itself are indistinguishable. Although this may be true in a conceptual sense, it is still the case that building and testing a “real” software implementation is more difficult and consumes more resources than evaluating a model. The reason is that models of software systems need not address the complexities and subtleties of the environment in which the “real” software is expected to operate. Like the concept of building from components, the idea of evaluating software systems by the analysis of models is being adopted by software engineers. We now have a number of highly respected and powerful systems which are used to simulate and evaluate software systems [Grumberg and Long 1994, Holzmann 1997, "FDR2 User Manual" 2000]. To use these systems, a developer constructs a model which is then passed over to the system for evaluation. These evaluations can be quite simple, such as a “brute force” search of the system states for deadlock. They can also compare a model’s behaviour with some requirement(s) which may be specified using the same language as that used to define the model, or some other language specifically selected (or designed) for the purpose. Typically, these systems are highly sophisticated and are able to exploit various techniques to enable them to

manage their hunger for memory effectively and complete evaluations of systems with millions of states in reasonable times. Advocates of systems which avoid this type of analysis by following the interface enhancement route could argue that the value of the results obtained from a model is dependent on the accuracy of the abstraction used in its construction, but similar problems accrue from inaccuracies in the application of behavioural features to component interfaces.

Modelling has other potential benefits: even when the real system is too large for exhaustive testing, it should be possible to build a representative model which is a manageable size. Also, because building a model is an exercise which is undertaken as part of the design phase of a project, results from an evaluation of a model can be available long before the product is available for testing.

The purpose of these models is to assist in the task of assembling the constituent parts of a software system into a coherent whole which will satisfy its requirements. For some systems, the function of the model may be simply to document how the various parts of the system are intended to interact and communicate this to those involved in the development. In this case, a simple diagram such as a RAD [Ould 1995] or UML [Fowler, Scott et al. 1997] diagram may well suffice. However, experience tells us that getting these designs right by simple inspection of a diagrams (or written descriptions) is notoriously difficult and designers need help if they are to avoid unexpected and undesirable behaviour in their systems.

For this analysis of models of software to be effective as a tool to help developers understand the behaviour of their systems, it must be thorough and mechanised. In turn this means that these models must be more precise than a simple diagram. The models need to be complete and formal enough to permit them to executed and subjected to mechanised analysis.

However, whilst most would acknowledge the advantages of building models of systems and subjecting these to some form of analysis, commercial software engineers display marked reluctance to use anything approaching formal methods in

their work. The reasons for this are not entirely clear. There can be no doubt that, despite the considerable effort which has be expended trying to make "formal methods" in general more appealing to commercial developers, these techniques are not in widespread use. They seem to have acquired a reputation for being difficult to use which deters engineers from considering them.

2.2 Essential features of effective modelling systems

As already indicated, if it is to be useful there must be some purpose in mind when constructing a model of a system and this purpose will dictate the type and style of the model which is appropriate. Where the objective is no more than to record (document) features of a system or communicate these features to the members of a development team, some kind of informal diagram is likely to suffice. However, even for this purpose, a more formal model may be desirable since the less formality there is to a model, the more opportunity there is for confusion to arise from alternative interpretations. Where the purpose of the model is to assist in the process of designing the interactions between the various parts of a system or to help the designers to satisfy themselves (and others) of the correctness of their work, the model necessarily has to be more formal. Once this more formal model has been constructed, it provides a much more precise description of the way the system is intended to behave than can be achieved in prose. Such a model can be used to verify or validate the design of the system by showing how the system is intended to operate and, with the help of analysis tools, either demonstrate that the system design satisfies various requirements or that the system is free from undesirable behaviours.

However, if we are to persuade commercial software developers to build more formal models, we need to supply them with modelling tools and techniques which they find acceptable. For the most part, it appears that potential users of formal modelling systems are not concerned about power or expressiveness. Instead they are overwhelmed by the awesome power of the systems which they perceive as having been developed with comprehensive applicability or analytical power in mind rather than approachability (for the inexperienced user). If these

systems are to be more widely applied, they need to appear more approachable. Potential model builders need to feel that it is feasible for them to construct modest models without a great initial investment of time and intellectual effort and that the result will be a model which is both sufficiently accessible that they can discuss it with fellow developers and sufficiently formal that analysis of it can produce useful results about the behaviour of the system described.

2.3 Characteristics of modelling systems

2.3.1 Formal-informal

A measure of how precise a model is in its description of a system. A reasonable test of whether a modelling language could be described as formal would be to consider a modelling language to be formal if it contains enough detail for a complete model to be executed. According to this interpretation of formality, a model constructed using RolEnact is formal whilst one described by a RAD or a UML interaction diagram is unlikely to be. For developers to be able to extract the greatest benefits of modelling, they do need to construct models which are sufficiently precise to be executable.

2.3.2 Graphical/textual

Despite the clear preference of most people for a graphical interface, writing some form of text (code) is the dominant method by which the modeller describes their model. Some are then able to convert this description into a diagrammatic representation [Henderson 1999, Magee and Kramer 1999, Henderson and Walters 2001] as a static description of the model and/or during its execution (or animation).

2.3.3 Communications paradigm

The essential feature which underlies all of the modelling techniques described here is that they describe systems which are built from a collection of parts which communicate, though they differ in the way that these processes communicate. The two fundamental differences are whether processes communicate through a

channel of some form or by taking part in the some kind of shared action and whether this communication takes place between just a pair of processes ("point to point") or between all processes interested in the event ("broadcast"). In addition, communication can be "synchronous" or "asynchronous".

Shared events:

In a modelling language which uses this communications paradigm, where an event in one process has the same name as an event in another process, these events must take place at the same time. Normally a process which is ready to perform a shared event is forced to wait until the other process(es) which are going to share the event is (are) ready to perform the event. The event is then permitted to occur, and when it does all of the processes which share the event change state simultaneously.

This communications paradigm is both powerful and permits the construction of most elegant process algebras, but novice users seem to find it difficult to use. It appears that they prefer to address communication between processes explicitly rather than have it occur as the implicit consequence of re-using an event name even if this means losing some of the elegance of the algebra. Some systems partially address this issue by permitting the modeller some control over the scope in which events are identified. Unfortunately, rather than helping, this seems to serve to further confuse the novice. This style of communication is invariably synchronous – all of the processes which take part in a particular communication are required to do so at the same instant.

Channels:

A channel is an explicit connection between two or more processes in a model. Where this paradigm is used, it is usual for the modeller to be required to both arrange for communicating processes to know (or be connected to) the channel to be used and also to cause the communication to occur explicitly. This scheme of communications seems to appeal to the novice modeller as matching the way they think about communications in real systems more naturally. Typically one proc-

ess will perform some kind of “send” operation to a channel and another will perform a complementary “receive” action on the same channel. This style of communication can be synchronous with all parties to a communication performing their actions at the same instant or channels may be able to store communications enabling sending processes to perform their part of a communication (place values in a channel) before the receiving process(es) is ready. This further enhances the appeal of channel based communication for the novice model builder who will be familiar with buffered communications in other theatres.

Synchronous/Asynchronous communication:

Many modelling languages and systems limit the modeller to constructing models which use synchronous communication only and this is seen by the inexperienced as a limitation of these systems. In fact, it is not. Although synchronous and asynchronous communications behaviours are very different, it is perfectly feasible to implement either using the other: Buffer processes interposed between communicating processes is all that is needed to create asynchronous behaviour in a synchronous systems. A scheme of acknowledgements for messages induces synchronous behaviour onto an asynchronous system. It is a matter of debate which of these schemes is better although it would appear that using synchronous communications leads to more elegant process algebras.

Point to point/Broadcast

The question of which processes take part in a particular communication is another characteristic which differentiates various modelling techniques. With a “point to point” communications paradigm, exactly two processes are involved in each communication: typically one sender and one receiver regardless of how many processes may be available to take part in it. With the “broadcast” paradigm, the number of processes which take part in a communication is not restricted to two. Instead it may be that any processes presently able to take part in the communication will do so or it may be mandated that all processes which are ever capable of taking part in the communication must do so each time it occurs.

The natural choice for modelling methods which use (synchronised) communications via shared events seems to be “broadcast” so that when the shared event occurs, it occurs in every process throughout the entire model in which it appears. Conversely, it would appear that the natural choice for a language which uses channel based communications is for communication to take place “point to point”. This rests easily with the notion of one process placing a value into a channel which another is able to retrieve, possibly later.

As with the synchronous/asynchronous question, there is continuing debate about which of these alternatives is the better.

2.3.4 Objective

For a model to be considered useful or effective, it must be judged against some kind of objective. This objective will drive the choice of modelling method to be used in its construction. For example, something like Promela code is what is required if the model is to be checked for deadlock using a model checking tool, but would be highly unsuitable to present an outline of the system to a non-technical audience where a UML sequence diagram might be ideal.

The following is a representative selection of tools and languages and techniques available for building and analysing models of systems:

2.4 Existing systems

We already have a large collection of modelling tools, languages and techniques. Some of these attempt to provide a comprehensive solution to every modelling requirement whilst others are tailored for a particular purpose. These languages can be classified in a number of ways:

2.4.1 UML

The Unified Modelling Language is representative of many diagram based modelling systems and currently is possibly the most popular of them. UML attempts to meld the best of many modelling techniques which have been proposed over a pe-

riod of years into a single coherent system. In doing so, arguably there is some duplication in UML where more than one type of diagram portrays essentially the same information. At the same time the language permits latitude in the way that its various diagrams are drawn and interpreted. Most relevant to the type of modelling under consideration here are activity and sequence diagrams. These diagrams are used to describe how parts of a system interact. Not only are they are easy to understand and construct, they provide an effective mechanism for communicating these ideas to a non-expert audience. A strength of UML is that it is becoming widely accepted in industry and there are heavy weight commercial packages available to support its users. However, in general, models created using UML do not lend themselves to formal analysis. This is not a criticism. The degree of freedom which this gives to the modeller using UML is one of the strengths of the language. This work could have used a sub-set of either activity or sequence diagrams, but it was felt inappropriate to make changes to such well known notations. [Fowler, Scott et al. 1997]

2.4.2 Rapide/Darwin/Wright

Rapide, Darwin and Wright [Luckham, Kenney et al. 1995, Luckham and Vera 1995, Magee and Kramer 1996] are representative of the approach to the problems of building software systems from components which attempt to incorporate behavioural issues into component interfaces.

2.4.3 Process Algebras and Model checking

In a sense, these techniques occupy the middle ground between informal system construction and rigorous formal development [Ip and Dill 1996, Holzmann 1997, Sullivan, Socha et al. 1997]. They do not provide “proof” of the correctness of a system in manner of “traditional” formal development where a system is developed from a specification (which is proven to have the required properties) by successive steps, each of which is proven to preserve the required features of the initial specification (and any other requirements arising from previous steps of the development). Instead, they provide a language with which a representative model of a (proposed) system can be constructed. It is this model which is sub-

jected to analysis, hence the name of the technique. In principle, if not in actual practice, model checking tools operate by performing an exhaustive search of all of the states of a model. This search is directed at establishing some particular property of the model. Typically, model checking software permits the modeller to input the property to be checked in either the same language as is used to describe the model or another language tailored for this particular purpose. In addition, or as an alternative, the modeller may be offered the opportunity to check a model for one of a number of standard properties, such as the model having no unreachable code or being free from deadlock. Generally, where a model check discovers a problem, the system will give the modeller a trace showing how the model is able to reach an unsatisfactory state.

Model checking software is already sophisticated and it is feasible to examine models with many millions of states in reasonable periods of time. Further development of these systems and continuing advances in hardware will enable them to handle even larger systems in the future. However, despite the great power of these systems, there is no realistic hope of them ever being able to handle the full detail of complete commercial systems.

2.4.4 CSP/FDR

Communicating Sequential Processes (CSP) [Hoare 1985] is a formal language for constructing models of interacting systems. It is a highly respected language with a good pedigree. Communication between processes in CSP is achieved by shared events. Each event which can occur in a model has a name. In its standard form, when an event occurs in CSP, it happens in all of the processes in which that event name appears. The event can only occur when it can happen in all of the processes containing the event. Where it is desired that an event is shared between a limited number processes, this is achieved by means of a system of labelling events (and processes). This style of communication is appropriate for some types of model. A variant of CSP uses a channel based style of communication.

A strength of CSP is that there is commercial strength tool support for the language such as the model checker, FDR. FDR is a fully featured and powerful model checking tool able to analyse substantial models written in CSP.

2.4.5 FSP/LTSA

Finite State Processes (FSP) [Magee and Kramer 1999] is a smaller modelling language based on CSP. It lacks some of the power of CSP, being carefully crafted to ensure that its models are finite. It is interesting in this context as evidently part of the reason for the constraints incorporated into FSP is to permit it to be used as the input language to the modelling tool, LTSA. LTSA is an attractive tool. It has easy to use features and will draw diagrams of FSP models in the form of Labelled Transition Systems. It is also able to check FSP models for a variety of fundamental properties. However, in common with CSP/FDR, model input into LTSA still requires the modeller to write code. Consequently, in addition to the concepts of FSP, the modeller needs to learn the syntax of the language.

2.4.6 ALLOY

Alloy is a modest, declarative modelling language developed principally by Daniel Jackson. The philosophy of Alloy has a number of appealing features. In particular, the developers talk about what they describe as “micro-modelling”. They assert that, regardless of the size of the target system, the analysis of models of parts or aspects of they system can provide useful insight even when these models are constructed with just a few lines of code [Jackson 2002].

A further interesting feature of the Alloy modelling language is the mechanism which is employed by the support tool for the analysis of the models. Instead of attempting to provide comprehensive support for the Alloy language and the analysis of models constructed using the language in the support tool, the developers have concentrated on the interface to the user and the features which set Alloy apart from other systems. For analysis, models are recast (mechanically by the tool) into an instance of a recognised problem which is then examined using a

(commercial) third party “solver”. The outcome of the analysis by the “solver” is then re-cast into an appropriate form and presented to the user.

2.4.7 Pi-Calculus

The pi-calculus [Milner 1993, Turner 1995] is a process algebra with a formidable pedigree. Its basic concepts are few in number and powerful. Like CSP, pi-calculus inter-processes communicate synchronously but in contrast communication takes place “point to point” through channels. The pi-calculus builds on the earlier work in “A Calculus of Communicating Systems” (CCS) [Milner 1989]. Despite its pedigree, there is no well known heavy weight implementation of an execution tool or model checker based on the pi-calculus (like FDR for CSP).

The major concept of the pi-calculus is of a “name”. Names are used to identify processes, channels and the values which are passed around between processes which contributes to the elegance of the calculus and enables some of its more powerful features. For example, since a channel is just a “name” and “names” are used for the values passed between processes in communication, processes are able to pass channels around (through channels) which permits a rearrangement of the interconnections between the constituent processes of a pi-calculus model during execution. This facility for dynamic re-configuration sets the pi-calculus apart from most other process algebras.

The essential features of pi-calculus are described in Chapter 6.

2.4.8 SPIN

SPIN is a highly featured, highly respected model checker which is both successful and popular. In common other model checking systems, SPIN models are built by writing code in a language devised for the purpose. The model input language of SPIN is “Promela” which has a syntax reminiscent of the popular programming language “C”.

SPIN has a graphical “front end” (XSPIN) which transforms the appearance of the analysis of a model, but the modeller still has to write code to construct their model for which they need to understand the Promela language.

SPIN and Promela are described further in Chapter 7

2.4.9 RADs

A Role Activity Diagram (RAD) is a style of diagram developed by Martin Ould. These diagrams and their application to business process modelling are described extensively in his book [Ould 1995]. The particularly attractive feature of RADs is the clear uncomplicated way that they describe the behaviour of a process. In Ould’s work he is concerned to describe business processes and change, but the technique is sufficiently general to be applied to any form of process. A complete RAD shows a collection of processes and how they interact. Each process is contained within its own box and interacts with other processes by sharing events. Each process has a starting point at the top of its box and progresses by taking part in events which take it into new states. The state of a process may be named. Named states are shown as circles labelled with the name of the state. Where a process returns to a previously visited state, this may be shown by a loop back to that state, but the rules governing the drawing of a RAD also permit a state to appear in the diagram in more than one location permitting the model builder to show looping behaviour without breaking the convention that the flow of control in a process moves down the page (and some freedom in the arrangement of the diagram).

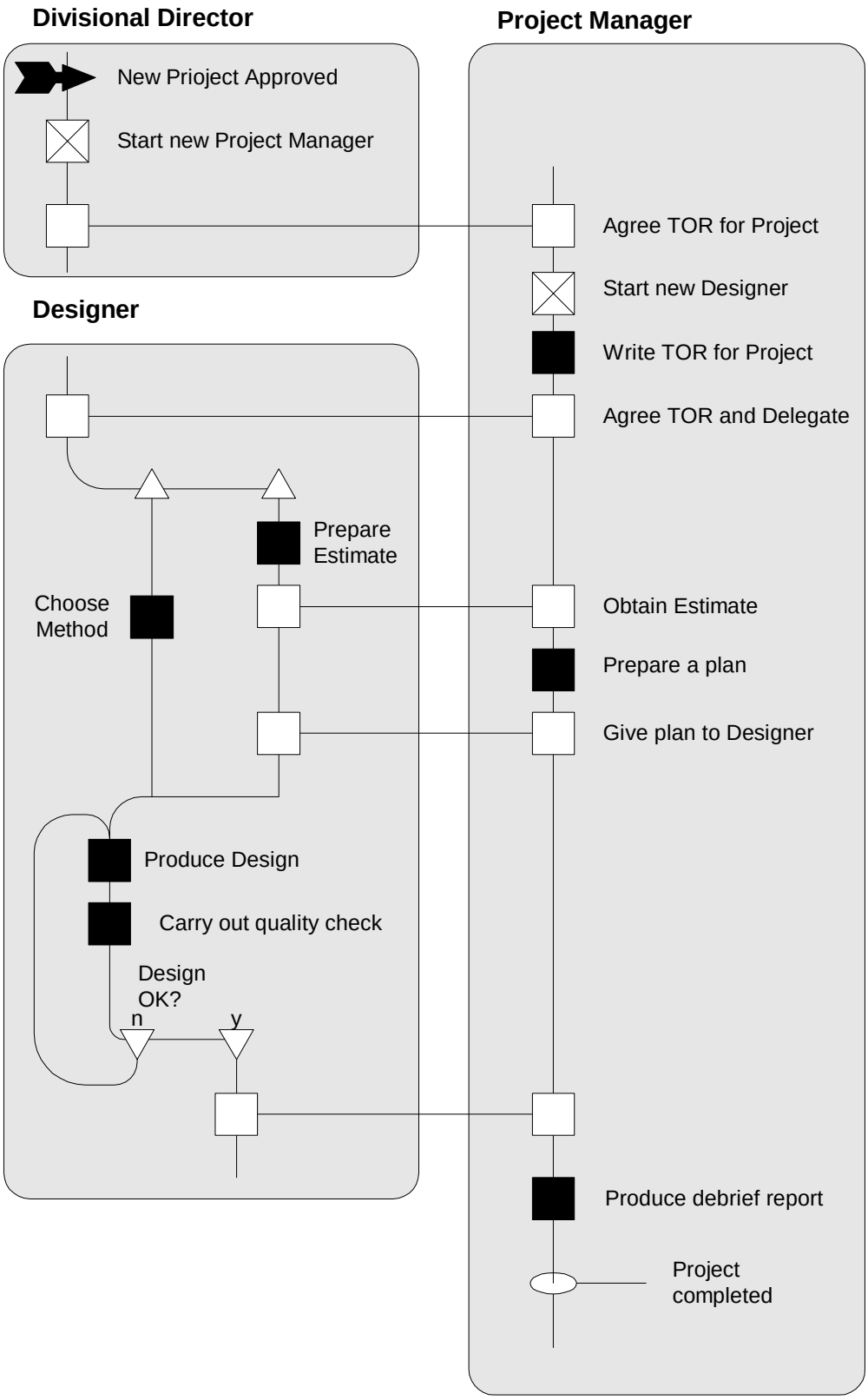


Figure 1: An example of a Role Activity Diagram

Within a RAD, there are several classes of event. The simplest is carried out by a process in isolation (and is shown as a black square). Others involve some form of communication between processes (shown as a white square). Communication within RADs is by means of shared events. However, unlike most modelling techniques which use this paradigm, shared events in a RAD are highly visible to the modeller as they are shown by a horizontal line connecting the events involved. Most shared events involve two Roles, but the notation admits the possibility that three or more may share in an event. There is no requirement to indicate which of a number of Roles sharing an event is responsible for initiating the event, but this may be shown by the box for the event in the "driving" Role being shaded.

A RAD process also has access to a special type of event which causes another process to be brought into existence. Such an event is distinguished by having a cross in its box. The type of Role which is created in the event is indicated in a label attached to the event.

Within the RAD notation, in addition to the various types of event, there are constructs which permit the description of choice and parallel execution within a process (or Role). In addition to the features described above, the full RAD notation has further features which permit the user, amongst other things, to add descriptions to states (in addition to naming them), external events which affect Roles and show partial descriptions of Roles.

2.4.10 RolEnact

RolEnact [Phalp, Henderson et al. 1998, Henderson 1999, Henderson and Walters 1999, Henderson and Walters 2001] was developed by Peter Henderson. It has a special place in this work and it is described further in Chapter 3

Like RADs and in contrast with the other techniques described above, the intitial target of RolEnact was to build a tool and method suitable for Business Process

modelling. This objective meant that it was anticipated that the RolEnact models would be built by business people who, although experts in their own fields, would have limited knowledge and tolerance of computer programming. Consequently, the RolEnact language was kept deliberately small to minimise the familiarisation required before a new user could see their first models running. This overriding desire to make the language suitable for use by the "non-expert" brought with it the need to accept that there would be limitations in the expressiveness of the language.

RolEnact succeeded in some respects. It has an attractive execution tool in which each running instance of a Role has its own window in which is displayed information about the Role and the actions it is presently able to perform. This interface seems to be sufficiently straightforward for the non-specialist modeller to understand what is happening and relate the behaviour of the model to the real situation being modelled (and a RAD where this is available).

However, although the language of RolEnact was particularly small (especially in its revised form), this need to write code in order to produce models still seems to present a barrier to the construction of models by non specialists.

2.5 Conclusion

There already exists a large and varied collection of modelling tools and techniques which might be appropriate for the construction of a model of any kind of proposed system to be built from a collection of parts. These systems vary from informal diagramming techniques to hard formal systems with rigorous mathematical underpinnings.

However, whilst most developers would accept an argument which proposed analysis of models as part of the process of evaluating system designs, few models are actually built. Regardless of the actual reality, modelling systems and their analysis tools are perceived as being inaccessible to non-specialist modellers.

To some extent these systems:

- Attempt to offer features (e.g., Expressive power) at the expense of "useability"

- Provide users with a text (or code) based interface although it is clear that they find this unattractive

- Require potential users to engage in a significant conceptual challenge before they are able to build and use even the simplest of models

- Provide only synchronous inter-process communication (typically using "shared events") when the typical enterprise system being constructed today will use a channel based asynchronous communications of some form.

Working developers are already persuaded of the benefits of using formal and semi-formal methods to assist them in the design and development of today's large systems and whilst not all of the problems are solved, existing systems are more than good enough to be very useful. However, if we are to incite people outside of specialist areas to make extensive use of these systems, we need to improve the way they are perceived by the novice on first introduction.

Chapter 3 RolEnact/RaDraw/ARE

RolEnact and its associated tools are significant to this work as they provided the inspiration for the interface to RDT. In particular, the development of RaDraw exposed numerous issues concerning the practical implementation of such a tool. This section will describe the RolEnact modelling language which was originally conceived for business process modelling, a graphical front end and automated execution tool which were developed later and finally some thoughts on how the language might have been developed.

3.1 What is RolEnact?

RolEnact [Henderson 1999, Henderson and Walters 1999, Henderson and Walters 1999, Henderson and Walters 2001] is a modelling system developed by Peter Henderson primarily targeted at Business Process Re-Engineering. It represents an attempt to create modelling system which is simple enough to be handed to the people involved in the processes to be modelled so that the models would much more nearly approximate to the actual process being considered. The RolEnact language presented below is a refinement and simplification of the original code. Later the system was enhanced by the development of a "graphical front end", RaDraw which presents models as diagrams and eliminates the need for a modeller to write and code and an evaluation tool, ARE [Henderson, Howard et al. 2001] which assists in the analysis of a model by executing it automatically and recording the results.

RolEnact has the concept of a Role which is modelled on the Role of a RAD. To build a model using RolEnact, the modeller first defines the behaviour of the various types of Role in the model. A RolEnact Role has state and is defined by the actions (referred to as events) which it is able to perform. There are four types of action:

3.1.1 Action

An “action” is a unilateral action of a Role causing it to change state. For a Role to be able to perform an action it has to be in a specified “before” state. The effect of an “action” is that the Role concerned unilaterally changes state to a specified “after” state. An “action” is specified in RolEnact code as follows:

```
Action RoleName.ActionName
       Me (BeforeState -> AfterState)
End
```

Where RoleName is the name of the Role for which the action is being defined, ActionName is the name of the action and BeforeState and AfterState are the names of the “before” and “after” states of the action. The use of keyword “Me” as the first word of the second line indicates that the state change specified in the following brackets applies to the Role which is to carry out the action. The keyword “End” shows the end of the definition of the event.

3.1.2 Interaction

An “interaction” is an action (or event) in which a Role and one or more other Roles which are known to a Role change state in a single co-ordinated event. An “interaction” is specified in RolEnact code as follows:

```
Interaction RoleName.InteractionName
       Me(MyBeforeState -> MyAfterState)
       Role2(R2BeforeState -> R2AfterState)
       …
End
```

The meaning of the first two lines is as for an Action. The additional line(s) in the body of the event definition describe the corresponding changes in the other Role(s) which are involved in the Event. In these lines, the keyword “Me” is replaced by the type of the other Role whose state change is described in the following parenthesis. For an Interaction to occur, in addition to being in the required before state itself, for each additional line in the body of the event definition, the Role must know of a Role of the stated type which is in the stated “before” state. When an interaction event takes place, all of the Roles involved change from the

specified “before” states to the specified after states simultaneously. Although, the language permits interactions between any number of Roles (as is permitted in RADs), in practice this feature is rarely used.

3.1.3 Selection

The way a “selection” is defined in the code is very similar to an interaction:

```
Selection RoleName.SelectionName
       Me(MyBeforeState -> MyAfterState)
       Role2(R2BeforeState -> R2AfterState)
       …
End
```

For Roles to interact, they must know of each other. There is a number of ways that components which need or want to interact with each other can acquire knowledge of each other. These methods, in essence, amount to arranging the model so that this essential information is injected as part of the initial definition of the model or by the use of some form of “name service”, or a combination of the two. RolEnact uses a version of the “name service” technique in the form of a special type of “interaction” event, called a “selection”. A “selection” event is like an “interaction” event, except that the Role performing the action does not need to have pre-existing knowledge of the others Role(s) with which it will interact. Instead, the RolEnact system supplies the name(s) of one Role of each of the required type and state if they exist. A side effect of the event is that all Roles involved retain knowledge of each other. Any existing references to Roles of the type involved are overwritten. Where a suitable selection of Roles does not exist, the event cannot occur. If there is a choice of suitable Roles, it is the system which chooses which of the available Roles will be involved in the interaction. The way this selection is made is not defined in the language. Roles which have interacted in a “selection” event are able to co-operate later by means of “interactions”.

3.1.4 Creation/Create

A Create event is another special type of interacting event. It is defined as follows:

```
Create RoleName.CreateName
       Me(MyBeforeState -> MyAfterState)
       New OtherRoleType
       …
End
```

As with all the other types of the event, the Role performing the event must be in the stated "before" state. However, it is distinguished in that it creates new instances of Roles in the model at runtime. After the line which describes the changes of the state of the Role performing the event, each of the remaining lines in the body of the definition of the event starts with the keyword "New" and is followed by the name of a type of Role. When the event occurs, a new instance of each of these types of Role is created (and placed into the distinguished state "initial"). As with the Selection event, a side effect of a Create event is that the Roles concerned retain references to each other so that they may "interact" later.

In addition to the code outlined above describing the Roles in a RolEnact model, all that remains for the modeller to do is to write code to tell the RolEnact execution tool how many of each type of Role to create at the start of execution using a series of lines in the following form:

```
New RoleType
```

Once the code is completed, the modeller can load their model into the RolEnact execution tool and examine its behaviour. The execution tool is a Windows program which displays each of the Roles in the model in a window. These Role windows show the name and the current state of their Role together with a list of events which are presently available to the Role. The user drives the model by selecting one of the available events in a Role and "double-clicking" on its name. The execution tool then performs the Event, changing the states of the Role or Roles affected and re-calculating the list of available events for each of the Roles in the model.

3.2 Analysis of RolEnact

The RolEnact execution tool interface is particularly appealing to and easily understood by the non-expert modeller so that it can be very effective. However, there are problems associated with the generation of the models which make the language less appealing than hoped to new users:

3.2.1 "Style" of RolEnact code

As part of the effort to make the construction of models (code generation) as easy as possible, every effort was made to make the code minimal and eliminate all unnecessary features and duplication. Unfortunately this has had undesirable side effects. The style of the code, which was designed to make it approachable is off-putting to potential new modellers because it is quite unlike anything else which they have encountered. Also, although an experienced modeller is not handicapped by the minimal features of the language, those new to the discipline feel that it lacks features they wish to use and seem to prefer to use a syntax and semantics which feels familiar, even if doing so requires more intricate code.

3.2.2 The communications paradigm

RolEnact uses the "shared-event" communication paradigm whereby Roles communicate by virtue of two of more Roles being involved in the communicating event. However, whilst shared events may be a natural and appropriate model of communications for describing a business process, it is not a such a good fit onto the communications which occur in computer systems built from components. Here inter-component communication often uses some form of channel-like point to point infrastructure.

3.2.3 The semantics of select

The "select" event is a powerful construct which allows Roles which wish to interact with others to find each other without requiring the modeller to implement arrangements for this "discovery" themselves as part of the modelling exercise. However, the power and simplicity of "select" comes at a price. Much of the

power of “select” comes from its ability to create associations between Roles which is an unseen side-effect of this type of event. The drawback of permitting “select” to work in this way is that, in the event that there is choice, the modeller has no control over which Roles will actually interact. This lack of control makes the novice nervous and can be frustrating to the more experienced modeller.

A further potential problem is hidden in the implementation of the RolEnact execution tool. In the event that a model is in a state where a “Selection” event has a choice of Role, the semantics of the RolEnact language do not specify which of those Roles is chosen. The implementation of the execution tool makes an arbitrary choice. However, the implementation of this choice is deterministic so that, for a given state of the model, the result is always the same. The semantics of the language do not state how this selection should be made, so this is not wrong and it means a series of executions of the model are consistent. However, it also opens the possibility that some execution paths which are permissible for the model can never be explored by the tool.

3.2.4 Opaque communications

In most modelling languages which use “shared events” for communications, each of the interacting parties has information about the communication within its own definition. The effect of this is that information about the interaction is scattered about the code. Its parts are located in the definitions of the various parties involved. To see and understand the effect of a communication, the modeller needs to bring these pieces to code together somehow. A further problem for the inexperienced is the possibility of causing synchronisation of events in different locations by inadvertent re-use of an event name.

RolEnact takes an alternate approach. All of the detail of an interacting event is included in a single location: the definition of the event in the “initiating” Role. This solves the problem of not having information about each interaction scattered throughout the code. However, it creates a complementary problem which is that information about the communication is completely absent from the definition of

some (at least one) of the Roles which are involved. Consequently, whilst understanding what happens in any particular communication is simplified (and accidental synchronisation is unlikely), seeing the full behaviour of a Role requires anyone reading RolEnact code to look at the whole file instead of just parts which define the events of the Role under consideration.

3.3 RaDraw

The initial motivation for RaDraw was to provide an interface to RolEnact which did not require the user to read or write RolEnact code since, despite the effort devoted to making it straightforward, potential users seem to find generating and reading RolEnact code difficult. There are two elements to RaDraw: it draws a representation of a RolEnact model as a diagram and it generates RolEnact code through a dialogue box based interface.

As the file containing the model is loaded by the tool, the code is parsed and the details of the model are stored in a data-structure. The model is displayed in the main window of the application. By making the appropriate selection from the "view" menu, the user is able to choose to see the model as a diagram generated from the data-structure or as text. Although it follows the same style, the textual display does not reproduce code read in from the file. Instead it is re-generated as required from the data-structure which describes the model. Should the modeller elect to save the model from within the tool, the same routines are used to create the text which is written into the file. The tool will also display a model as a diagram. This diagram is not drawn by the RaDraw user, instead it is generated as required from the data-structure. This generation of the diagram is entirely automatic and does not require any input from the user. The style of the diagram follows the general principals of a RAD.

RaDraw is definitely better received by the novice than the textual code interface to RolEnact, but there are problems with the tool. Some of these problems could be addressed by further refinement to the code, but others arise from the nature of the task which the tool attempts to perform. Problems identified include:

- Roles which return to a previously visited state. The algorithm which draws the diagram is easily confused by this kind of behaviour. To some extent this is addressed by permitting the modeller to give hints when a state is being revisited (by suffixing "=" to its name). A solution to this problem could be built into the drawing algorithm.
- As good as the diagrams are for showing the interactions between processes, they don't convey the important differences between "Selection" and "Interaction" types of event.
- When generating code for execution in the RolEnact "stepper", the execution tool, RaDraw needs to know how many of each of the types of Role in the model there should be at the start of execution of the model. There is no place on the diagram for this information. The usual assumption made by the tool is to create a single instance of the first defined Role. This in turn has led to the confirmation of an idiom in RolEnact models whereby the models include a Role (often named, "Control") which is not part of the system being modelled. This Role has one "Create" event for each of the other types of Role in the model and is used by the modeller at the start of execution to bring the required Role instances into existence.
- It might be felt that there is an equivalence between a RolEnact model as code and that same model drawn by a modeller as a RAD-like diagram. In fact this turns out to be untrue. The code lacks information that is evident in the diagram such as the placement of Roles and events. An expert's selection of how to arrange the Roles and the events within them is likely to be better than that generated algorithmically by the tool, even when this is not influenced by knowledge of the processes and events being described. The diagram lacks some essential information too, such as the number and types of the (instances of) Roles which exist at the start of execution.
- Since the syntax of RolEnact code was crafted to permit users to write code directly, the tool had to read and write files which used the published syntax. One of the aims of the tool was to generate pictorial representations of existing models. Allowing the tool to use additions to the

RolEnact syntax or a different file format would have made its models incompatible with existing code, and prevented those already familiar with the code from using that knowledge as they would have been unable to reproduce in their work any (undocumented) additional code required by RaDraw.

In order to operate, RaDraw makes various assumptions such as those mentioned above. Often these assumptions are correct or reasonable and have little impact on the diagram or the model when executed, but this is not always the case.

3.4 ARE

ARE is a tool which takes a RolEnact model and runs it automatically. As the model runs the events which occur are recorded into a list. The tool was constructed to permit some analysis of the behaviour of a model. Once a model is loaded into the tool, it repeats a loop in which it selects an event from those available, records which event has been chosen and causes the event to happen. This process is repeated either until the model reaches a state where no events can happen or some number of events set by the user have occurred. Experience with running models led to the incorporation of a system of hints to ARE in its selection of which event to select from those available. This enables ARE to reproduce the behaviour of a system where, for example some events only occur when no others are available, such as a “timeout” and where others will be chosen whenever they become possible. Running a model in ARE could potentially find errors in a model like deadlock and failures to make progress, but this was not its real purpose. Instead it was built to give a modeller the opportunity to see the behaviour of a typical execution of their model and to experiment with the consequences of adding or removing instances of processes or the attitude towards the selection of particular events which they became available.

Provided the execution hints don’t preclude it, random execution of a model by ARE might reasonably be expected, eventually, to exercise a model fully so using it might find deadlocks or other faults in a model, but this would not be efficient,

nor can there ever be any guarantee that the tool has visited all states or explored all of the possible traces of execution.

3.5 Further potential enhancements to RolEnact

Despite any shortcomings of the RolEnact language, the interface of its execution tool is so appealing that it was felt worth the effort to build a successor language which used the same execution tool interface, if not the same tool.

The RaDraw tool represents an improvement on the system yielding two distinct advantages over the original form of the language. It provides an interface which permits users to create RolEnact code without learning any of the syntax of the language (whilst retaining compatibility with hand constructed code) and presents models to users in the form of a diagram.

A critical analysis of RolEnact, with the addition of RaDraw as a language for building models of software system leads to the conclusion that it could be improved in several ways.

A particular problem arises from the solution adopted in RolEnact to the problem of "discovery" – the means by which Roles (or processes) find each other during execution. In RolEnact, there is a side effect to both "Create" and "Selection" events which causes each Role involved in such an event to retain a reference to the others. Any pre-existing reference to Roles of the type(s) concerned are overwritten and lost. In the case of the "Selection" event a Role is able to find a sought for instance of a type of Role by appealing to the system infrastructure, in effect "by magic". The definition of the RolEnact language does not specify which Role is to be Selected, should there be a choice, so there is no constraint on the way the execution should chose the Role to Select, should there be more than one candidate. In the implementation, the operation of the this mechanism is hidden from the modeller who has to accept that when more than one Role is available to be selected, they get whichever the system decides to choose. (In fact the

algorithm used is extremely simple so the wily user might be able discern how it operates and manipulate it to a limited extent by re-arranging their code.)

Several variations and alternatives to the existing discovery mechanism of "Create" and "Selection" were considered, including:

- Permitting a Role encountering a new instance of a Role to add it to a collection of references to instances of that type of Role instead of overwriting and losing an existing reference. On occasions where more than one of the known instances were ready to take part in a particular "Interaction", the user of the execution tool would be offered a choice.
- Adding a new type of event in which a Role would behave like a "Selection" event if either there was no existing reference to the required other Role, (or that a known Role was not in the required state) and as an "Interaction" otherwise.
- Eliminating the "Selection" type event and replacing it with an explicit "name server" type mechanism for the discovery of other Roles or a means by which the modeller could inject information about references to other Roles at design time and eliminating dynamic discovery from the language.

Eventually, it was decided that none of these really amounted to a satisfactory solution to the discovery problem. In addition, it was felt that the shared event paradigm used for communications in RolEnact constrained the usefulness of the system because it does not lend itself to being extended to permit asynchronous communications. Together these factors led to the decision to take the attractive features of RolEnact and apply them in the development of a new modelling language.

Chapter 4 The RDT notation

This section describes the features of the RDT, a graphical language for the description of processes and systems built from communicating systems built from instances of these processes [Walters 2002].

4.1 Rationale of the language

The primary focus for this language was to generate something which would appeal to users who are not familiar with building formal/executable models. For this reason, where choices have had to be made, the language adopts the alternative expected to be more appealing or more easily understood by the novice, even when this constrains the expressiveness of the language.

As observed previously people in general like diagrams and pictures. They also seem to be able to absorb information about the behaviour and structure of systems more easily from them. Consequently, the decision to adopt a pictorial paradigm for the description of RDT models in preference to the more usual text (or code) was obvious.

The next decision was to make a clear distinction between the description of processes in general and the use of instances of these processes in particular systems. RDT does this by separating the two actions of describing the behaviour of (types of) processes from assembling (instances of) them into systems into two separate and distinct activities. By making this distinction in this way, the modeller is explicitly aware that they are dealing with instances of processes during the activity of assembling processes into a completed system. At the same time, since the task looks different, it is not so tempting for them to slip out of thinking that they are describing a type of behaviour when they build a process in the language. However, forcing this thinking onto the user in this way brings with it a constraint: in the typical formal modelling language, the syntax of the language is arranged in such a way as to permit the modeller to use the same constructs to build either

their final system or compound processes which they can then combine (with or without other “simple” processes) into their final system. In adopting the scheme it uses to establish this distinction between whether the modeller is dealing with the behaviour of a process in general or particular instances of a type of process, RDT forgoes the usual mechanism for the modeller to make their models hierarchical. This lack of opportunity to build models in a hierarchical style is considered acceptable for this language for several reasons:

1) It was felt that the novice modeller is unlikely to miss these features.
2) For hierarchical model building features to be really effective, they need to address issues relating to the encapsulation of communications between components as well as placing components within larger components.
3) It is expected that the modeller using RDT will confine their efforts to building relatively modest models. There is potential for even the smallest, most abstract model to be useful [Jackson 2002].

The communications paradigm adopted for RDT is based on the notion of channels and is point to point. This matches communications in the pi-calculus, but this was not the reason for adopting this style. Instead it was adopted as it was felt to most nearly match the typical communications regime which the target RDT audience would be familiar with using. Although the model generation tool and the language itself are silent on the length of channels, the option for channels to be buffered was added to the RDT execution tool for similar reasons.

The form of the diagrams used to describe processes in RDT follows that used by RaDraw which in turn was inspired by the Role Activity Diagram (RAD) but certain features of the RAD are omitted. In particular, where in a RAD, a fork in the line of control may be a choice, or the division of the activity of the Role into two or more parallel threads of operation. A RAD distinguishes between these two cases by the orientation of a triangle placed at the head of each line of control. In RaDraw and RDT, the division of the activity of a Role/process into parallel threads is not permitted so where the line of control branches this must always

represent a choice, consequently the triangular device is not required. The decision not to include parallel threads within processes in RDT is motivated by the desire for the language to be as simple as possible and justified by the two observations that its omission from RolEnact did not present many problems and, should it be essential for a particular model, this behaviour can be simulated by co-ordinating the actions of several processes.

It was felt that the collection of events to be provided in the language should be kept to a minimum in the spirit of making the language small enough for a new modeller to feel able to understand it quickly. The language has just three types of event which are described below.

4.2 The classes/instances problem

In a description of a complete system, there is a distinction between matters which relate to the behaviour of types of process within the system and matters which relate to particular instances of those behaviours. However, this distinction is often blurred or ignored in existing modelling tools so that the modeller is not required to think explicitly about the differences between these two types of information. In some circumstances, such as the case of a process of which there is exactly one instantiation, this distinction is unimportant, but not addressing this distinction usually leads to confusion, if not ambiguity. RADs are a good example of this: proper consideration of a diagram of a Role in a RAD has to be construed as describing the behaviour of that kind of Role as a class. Yet the user, particularly the inexperienced, finds it hard not to slip into thinking of each of the Roles in the diagram in terms of a single individual. This leads to problems when a situation is encountered where there must be many instances of the Role. A similar blurring of these matters is evident in FSP where processes are described and then combined into a complete system for analysis. The initial description of each of the processes describes the behaviour of a process of that type, as does any composition of those processes into a composite process. However, the execution and analysis of a system is carried out upon a single instantiation of the selected final system - and in instantiating that system, each of the constituent components

of that system is resolved into a instantiation of the process concerned. Thus, what is usually the last line in the file, the final description of the system to be created differs from all the rest in that it defines instances of a behaviour instead of classes of behaviour.

In the RDT system, this distinction is made explicit. A complete model is constructed in two distinct parts: first, for each process type there is a description of the behaviour of that type of process: its events, states and the channels it uses, then there is a description of how instances of these behaviours are assembled into a particular model.

The first part of the description is concerned with describing the behaviour of the processes in general, or as a class. It describes how "one of these processes" behaves - the events it takes part in, its states, the channels it uses and the names it uses for them (internally).

The second part is a description of a particular model which is to be executed and analysed. Here each process in the description is an instance of one of the types of process described earlier. This part describes the correspondence (if any) between the names used for channels within each process instance and how this particular collection of instances of the processes (which have been described in general in the first part) is connected together to make a complete system.

4.3 Description of the language

Building a modelling tool which offered all of the features of the pi-calculus would be a significant task in itself, but for this language an objective was to minimise the effort required by an unfamiliar user. Selected features of the pi-calculus are offered by the language, including the notions of processes, channels and communication along channels. Processes are able to communicate synchronously using channels in a manner similar to the pi-calculus, but in addition channels can be set to behave as fixed length buffers giving asynchronous communication between processes. This additional feature is provided based on the

pragmatic observation that most substantial systems under construction today use some form of asynchronous communication. The language does not have the full features of the pi-calculus. In particular, it does not have a feature which corresponds with the pi-calculus "!" operator and whilst it does permit the creation of new channels, this is not exactly equivalent to the pi-calculus (νx)P construct.

RDT differs from a more traditional modelling language in that models constructed using the language are built by drawing diagrams in place of the more normal textual descriptions. The style and arrangement of the elements in these diagrams has been selected to be familiar to a user who has encountered other diagram based techniques, such as UML (or RADs). This was done deliberately to permit a user looking at an RDT model to be able to understand the meaning of its various diagrams based on their intuition (born of learning about other, similar techniques). The rules about the placement of the various elements in the diagrams and their meanings are described below.

A completed model in RDT is divided into two parts: RAD-like diagrams which describe the behaviour of each of the types of the processes in the model and a further diagram looking a bit like a wiring schematic which shows and names the instances of the processes which are included in the model and how they are (initially) interconnected.

4.3.1 Describing Processes

These are described by RAD-like diagrams and form the building blocks from which a complete model is assembled. Processes have a named state. When they are created this internal named state is set to the distinguished value of “initial”. The modeller is free to select names for the remaining states which are appropriate to the process being described. Processes proceed from one state to another by taking part in events.

In the diagram, the states of a process are shown as circles which are labelled with the name of the state. Events are shown as squares which are also labelled with

their name. The flow of control is generally down the diagram and is shown by vertical lines. Each state is joined to the events in which a process in that state might take part by a line from the bottom of the state's circle. Where more than one event may follow a state, the process has a choice of action and the line of control is forked as required. There is the only situation where the line of control is forked in RDT so any fork represents choice and there is no need for these divisions to be decorated as they are in a RAD. Similarly, the state into which a process is moved after each event is connected to the event by a further vertical line from the underside of the event's square and, where more than one event may immediately precede a state, the lines are joined. Where a process returns to a previously visited state, as with the RAD notation, the re-visited state may be drawn again lower in the diagram. In RDT, the fact of such a state being a duplicate is shown by suffixing its name with an additional "=" character each time the state is re-drawn.

An RDT process may take part in three types of event. The collection of events offered in the language is intended to be minimal (to reduce the effort required to learn about them) without being excessively restrictive in the behaviours which they can represent. The need for the "Send" and "Receive" events follows naturally from the purpose of the language which is to model the behaviour of collections of processes which communicate via channels.

After consideration, it was decided not to include an internal event in the style of the RolEnact "Action". The omission of this event encourages the modeller to abstract away internal details of their processes from their model and concentrate on the interactions between the processes, which is where most of the difficulties arise.

In addition to the "Send" and "Receive" event types, RDT also offers an event described as "Create" which permits a modeller to processes to emulate the effect of the "ν" operator of the pi-calculus by permitting the creation of new channels at runtime. Making this event available to the modeller does add complexity to the

language. However, it does assist with the elegant representation of several types of behaviour. For example, the event may be used to bring into existence a new channel when a new interconnection between processes is being forged. By allowing the creation of a new channel rather than requiring the modeller to re-use an existing one, the modeller can guard against unplanned communications arising from past disclosure of the channel name in connection with some previous use. A further use for a newly created channel is to use it not for communications but to represent an item of data (perhaps the outcome of an enquiry to a database or the result of some calculation).

The three types of event, how they are drawn, when they are permitted to occur and their effects are outlined below.

4.3.1.1. Send

Figure 2 shows the way a send event in described in RDT. The event is represented by the uncoloured square which is labelled with the name of the event. A process executing a send event moves from a named "before" state (which will be above it on the diagram and to which it is attached by a vertical line) to a named “after” state (which will be below the event in the diagram and also connected to the event by a vertical line). This internal change of state is the only internal effect of the event on the process. In addition to this change of local state, the event also causes a value to be placed into a channel. This is shown on the diagram on a horizontal line which is labelled with the name of the channel into which the value is placed, together with the local name of the value which is sent. The channel name specified in the definition of the event is the local name this process uses for that channel. Similarly, the value which is written into the channel is that referred to by the local name for the value concerned.

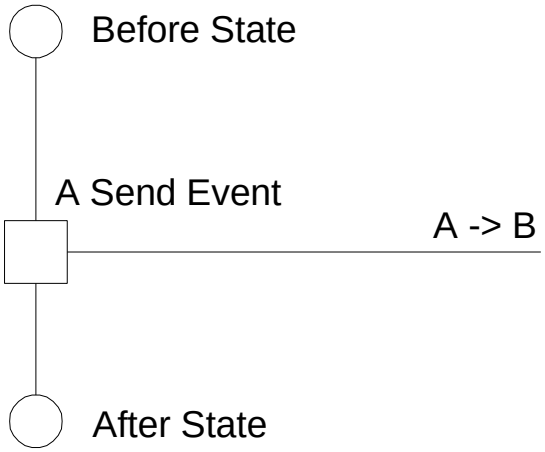


Figure 2: A Send Event

For a process to be able to perform a “send” event, four conditions must be true:

1. The named state of the process must be the “before” state of the event
2. The specified local channel name must be associated with a channel.
3. The specified process value name must be associated with a value.
4. The channel must be prepared to accept an additional value.

4.3.1.2. Receive

An example of a Receive event is shown in Figure 3. This type of event is complementary to a Send event. As with a Send event, a Receive event is shown as a square between two circles representing named states on a vertical line of control. To contrast it with the Send event, the square of Receive event is coloured in.

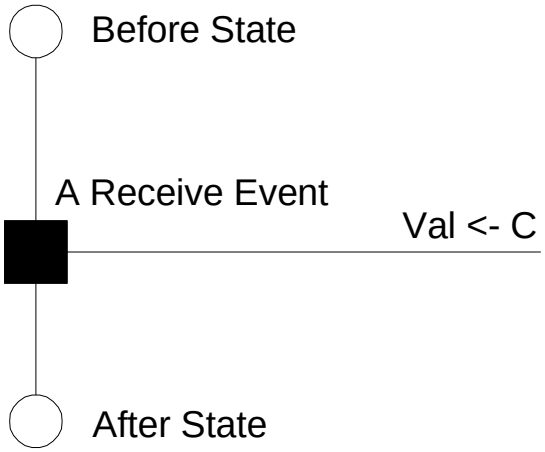


Figure 3: A Receive Event

As with the Send event, should the event occur it has two consequences which occur as a single indivisible action. The process moved from the named "before" state to the named "after" state and the process takes a value from the named channel. This value is then associated with the name specified in on the horizontal line connected to the event. Should a process read from a channel into a value name for which it already knows a value, the new value overwrites the existing one (which is lost).

For a process to be able to perform a "receive" event, three conditions must be true:

1. The named state of the process must be the "before" state of the event.
2. The specified local channel name must be associated with a channel.
3. The specified channel must have a value available for the process to read.

4.3.1.3. Create

A Create event is a special type of Send event. It is represented in a process diagram as shown in Figure 4. The difference concerns the value which is written into the channel. Unlike the Send event, the value to be written into the channel by the event is new and created as part of the event. Additionally, the new channel is associated with the local name specified in the description of the communication associated with the event.

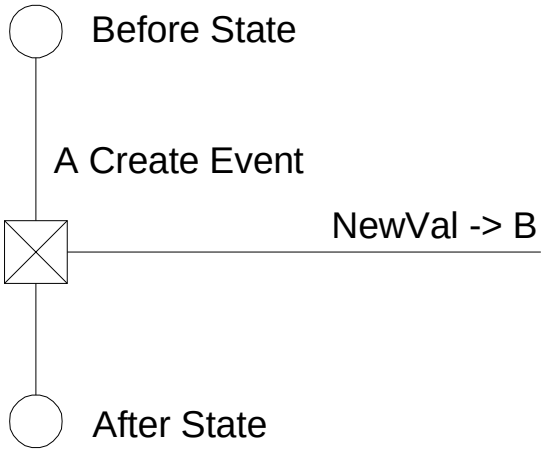


Figure 4: A Create Event

For a Create event to occur, the following conditions must apply:

1. The process must be in the stated “before” state.
2. The specified local channel name must be associated with a channel.
3. The channel must be prepared to accept a new value.

When the event occurs, all of the following effects occur as a single indivisible action:

1. The process moves from the named “before” state to the named “after” state.
2. A new channel is created.
3. The new channel is associated with the local channel name.
4. The new channel is written into the specified channel.

As with a receive, should the local data name already be associated with a value, this is overwritten and the old value is lost.

4.3.2 Constructing Processes

The processes of an RDT model are constructed from a collection of event definitions. Each process starts in a starting state which is given the distinguished name

of “initial”. The circle representing this state is connected to each of the events which can happen when the process is in this state. In turn each of these events is connected to a circle representing the “after” state of the event. These lines of control are branched and re-connected as appropriate. When drawn, a process is labelled with its name and contained within a box. Figure 5, Figure 6 and Figure 7 show three examples of small processes.

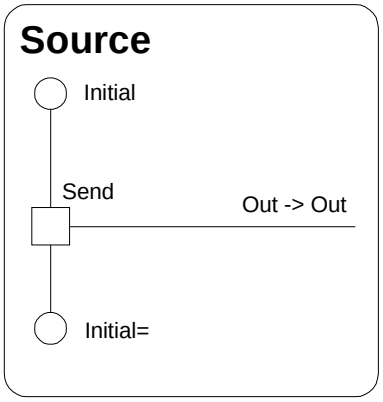


Figure 5: A minimal Source process in RDT

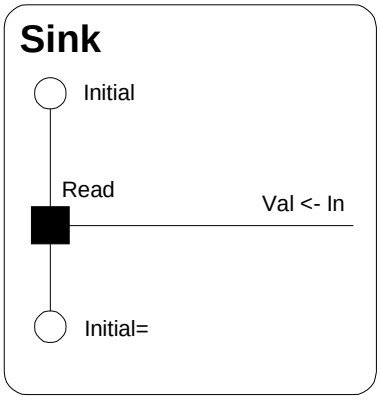


Figure 6: A minimal Sink process in RDT

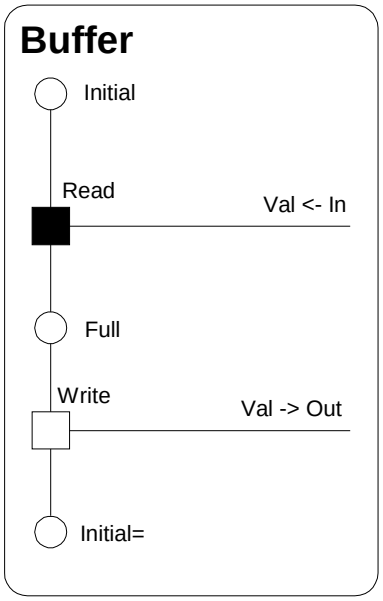


Figure 7: A one place Buffer process in RDT

4.3.3 Describing Models

In addition to the collection of diagrams which describe the behaviour of the processes in a system, the creation of a model in RDT requires a further diagram. This diagram describes (and names) the instances of the processes which will form the complete system and how they are initially connected. As with the style of the process diagrams, the style of this diagram is deliberately reminiscent of other diagrams which the anticipated user may have encountered in other activities. The diagram has two important features:

4.3.3.1. Instances

An "instance" is a single instance of one of the types of processes previously defined. A process instance is drawn as a rectangle from which the names which it knows for channels and are available for connections are listed down the right hand side. Each process instance has its own name and is independent and distinct from other instances of the same type of process. A process instance is uniquely identified by a pair consisting of the name of the type of process of which it is an instance and the name assigned to that particular instance. Each of the “names” known to the process instance potentially represents a channel which the process instance might use to communicate with others. These names are

shown on the diagram as labelled blobs connected to the instance's box by horizontal lines. Figure 8 shows how an instance of the Buffer process (whose behaviour is described in general in Figure 7) is drawn in a Model diagram. Notice that the name of this particular process instance (Buff1), together with the type of behaviour it displays (Buffer) are shown inside the box and the channels available for connection to other process instances drawn down and labelled (with their local names) along the right hand side of the process instance's box. It should also be observed that the channel, "Val" which exists in the definition of the "Buffer" type process behaviour does not appear in this description of the process. It makes little sense to include this name in the list of channels for two reasons:

1. No event in the process description reads from or writes into the channel named "Val", hence no communication can take place along this channel and so there is no point to connecting it to anywhere.

2. In this particular example, the first event of the behaviour of the "Buffer" process causes a new value to be associated with the identifier, "Val" so, even if it were connected somehow, the process cannot use that information as it is lost as soon as the process performs its first event.

Omitting such "names" from the process instance pictures helps to eliminate unnecessary clutter from the representation of process instances in the model diagrams whilst at the same time assisting users to construct correct models by preventing them from making this type of meaningless connections.

Figure 8: An instance of the Buffer process named Buff1

4.3.3.2. Connections

For process instances to communicate, they need to know an appropriate value for a channel which is known to the intended partner in the communication. Unlike RolEnact, in which processes can use a “Selection” type event when they wish to locate a partner with which to communicate, an RDT process can only communicate through channels which it already knows. A process may have this information because it has been received through an earlier "read" event or by knowing the information as part of the initial configuration of the environment in which it operates. Connections are the means by which this initial mutual knowledge of channels is injected into the initial state of a model. These "connections" describe an association between pairs (or more) of local names used within instances of processes - making a connection between two names indicates that they both (all) refer to the same value. They are shown on the diagram by drawing a line connecting the appropriate blobs on the sides of the process instances. Figure 9 shows a model diagram in which three process instances are connected using two “connections”. In the model shown in Figure 9 all of the names in the three processes are shown as connected to another in a different process instance, but this need not be the case. A name may be connected to another in the same instance or not at all. Names known to a process instance which are not connected in the model diagram may become connected during execution as a consequence of a “receive” or “create” event or, in at least some executions, they may never be connected.

As with the process diagrams, a completed model diagram should be enclosed in a box and labelled with its name. Figure 9 shows a model in which one instance each of the Source, Buffer and Sink processes described in Figure 5, Figure 6 and Figure 7 are connected in the expected manner to form a complete model named “Buffer Model 1”.

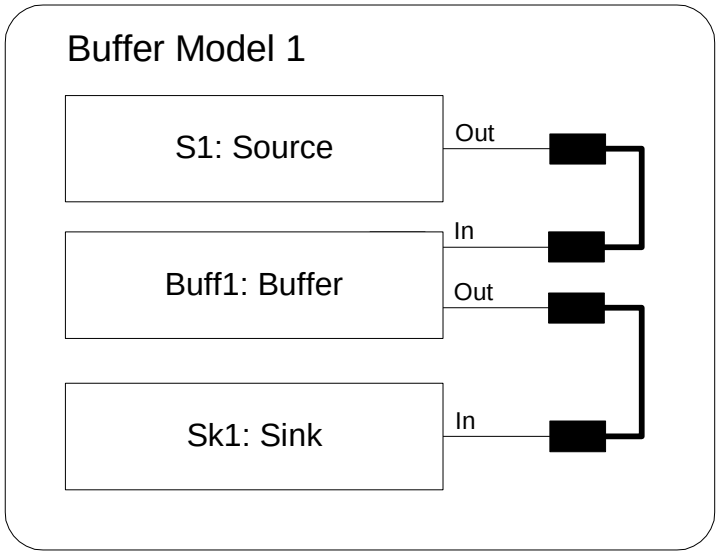


Figure 9: A Model diagram showing two connections

4.4 Using the language

The following diagrams illustrate the use of RDT to construct a model of one of the standard RolEnact examples, the "Barbershop" and the business process described by the RAD in Figure 1.

4.4.1 The Barbershop

The Barbershop is a standard model used in the documentation of RolEnact. It is based on what used to be the typical pattern of activity in a barbershop. In one of these shops, there would be one or more barbers. A customer arriving at the empty shop would be ushered into a chair by (one of) the barbers who would then establish what sort of haircut was desired. The barber would then cut the customer's hair and ask for payment. Once the customer had settled up, they would leave the shop and the barber would then be ready to attend to another customer. A customer arriving at the shop to find that all the barbers were busy would either sit on a bench and wait or leave (presumably either to find another barbershop, or return later). The usual version of the model has the refinement of permitting the barbers to elect to take a break during which they are not available to cut hair.

A representation of the behaviour of the barber is shown in Figure 10. This Barber starts with the choice of events. The first is the create type event called “Get Customer” which represents the Barber selecting a Customer. The action of this event is to create a new channel which the Barber elects to refer to as “MyCh” and send it on the channel which the Barber knows as “Custs”. Now, the Barber waits to receive instructions about the cut to perform. These instructions arrive as a value which the Barber receives along the channel “MyCh” and associates with the name “Work”. The Barber then sets about cutting the Customer’s hair. When the cut is finished, the Barber informs the Customer by sending another newly created value to the Customer along the channel “MyCh” and associating it with the name “Cost”. The Customer completes the interaction by sending a value which the Barber associates with the local name “Cash”. The Barber is then returned to the initial state ready to attend to another Customer. This sequence of events shows two ways in which the modeller may use the “create” event of RDT. The first is the creation of the channel known as “MyCh”. By sending this new channel to the Customer and using it for their subsequent interaction, the Barber is able to establish a private communications channel with the Customer. The other way a modeller may choose to use a “create” event is to disregard the special properties of the channel which is created and use it as if it were an “ordinary” data value. The Barber’s event “RequestFee” illustrates this style of use. The event generates a new channel (named Cost within the Barber process). No communication passes along this channel in the model. Instead it is used as a token to represent the bill which is presented to the Customer by the Barber on completion of the haircut.

Alternatively, from the initial state, the Barber may decide to take a break. In RolEnact, this action of taking a break is modelled by a pair of internal actions which affect only the named state of the Barber. RDT does not have an event of this type, partially to encourage modellers to abstract away detail of the internal operations of processes and concentrate on interactions between them. If this behaviour must appear in an RDT model, it has to be modelled as some form of communication. In this example, the events are modelled by the Barber informing

some third party (along the channel “Info”) when they start a break and when they return.

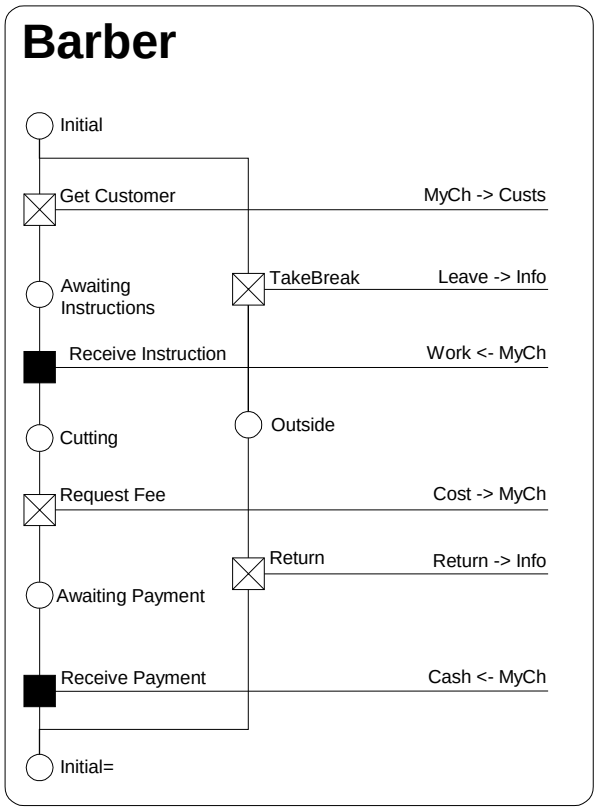


Figure 10: The Barber process in RDT

The complementary behaviour of the Customer is described in Figure 11. As with the Barber, at the start of execution the Customer has a choice of actions. They may either wait for a communication from a Barber or leave the shop. As with the Barber taking a break, the essentially internal action by the Customer of leaving is modelled by informing a third party of the decision (along a channel known to the Customer as “Info”). In contrast with the Barber, the Customer who has either had a hair cut or left the shop does not return to their initial state. Instead they enter a final state. There are no events for which this is the “before” state so they are unable to leave this final state. Having reached this state they can take no further part in the execution of the model.

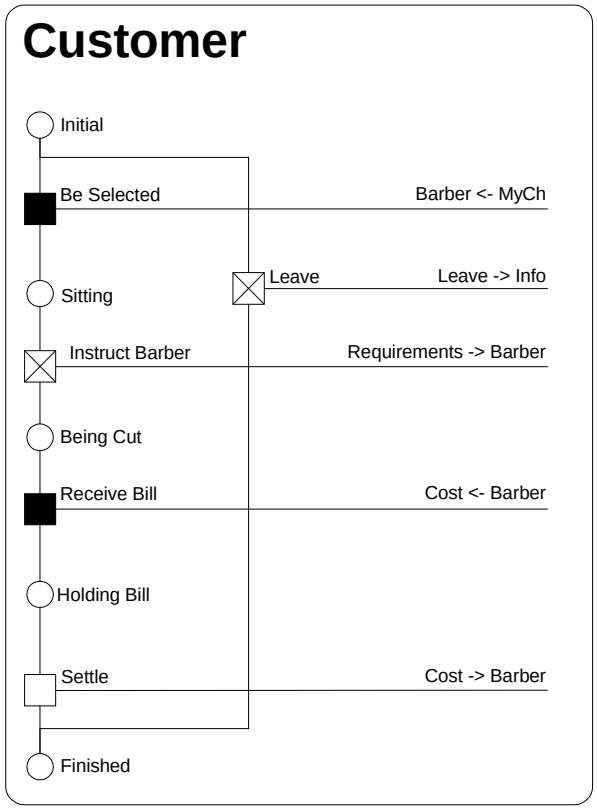


Figure 11: The Customer process in RDT

An example of a completed barbershop model shown in Figure 12. This particular version has two Barbers, two Customers and a Sink process instances. The Sink process instance accepts notifications from Customers who decide to leave and from Barbers taking or returning from a break. The model diagram shows the connection of the "Custs" channels of the Barbers and the "Barber" channels of Customers and also the connection of the "Info" channels of both Barbers and Customers to the "In" channel of the Sink process. None of the "MyCh" channels is connected in the diagram. Instead connections between these channels are created during execution of the model.

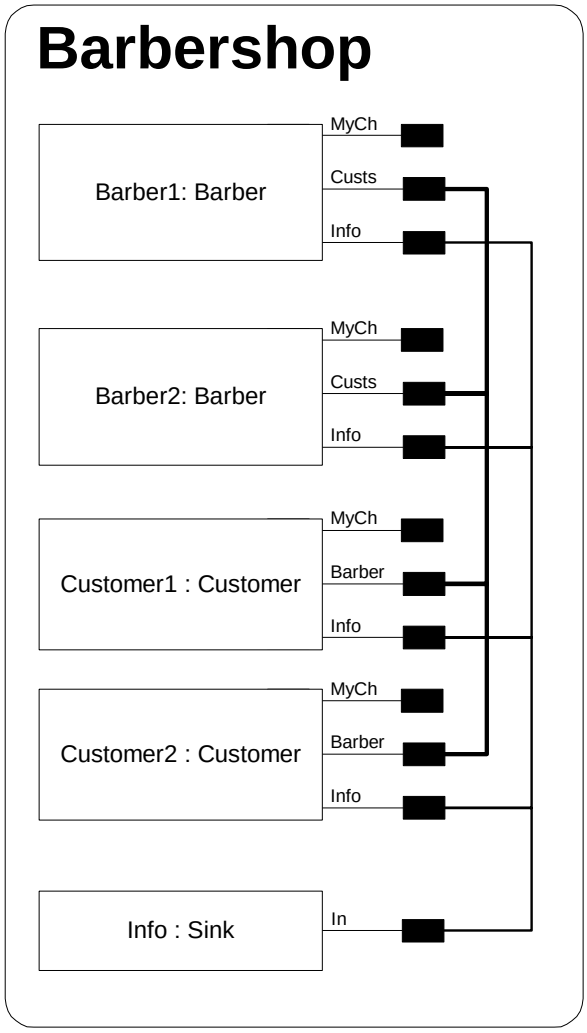


Figure 12: A Barbershop model with two Barbers and two Customers

4.4.2 Project Management

This is a model in RDT of the project management process described in the RAD in Figure 1. Whilst the process diagrams of RDT are inspired by the RAD notation, there are important differences, some of which are illustrated by this model.

The first of the three processes in the model is the Director. Comparing the RDT process with the RAD description of the Divisional Director, differences are immediately apparent. RDT does not have an equivalent for the arrow used in the RAD to indicate an external event which is used here to highlight the approval of a new project and direct the reader to the start of the process. In the RDT model presented here, this external event is abstracted away and assumed to have oc-

curred before the start of execution of the model. In other situations, the modeller may prefer to model such actions explicitly as internal communications, perhaps by making one of the communicating parties represent the "outside world".

The first action of the Divisional Director process in the RAD is to start a new Project Manager process. This is represented in the RAD by the crossed box. RDT does not permit additional instances of processes to be created at runtime. Instead all of the processes required for the execution of a model are created before the start of execution by being shown on the "model" diagram. However, RDT does permit creation of new channels and also the re-arrangement of the interconnections between processes so where the effect of process creation is essential it could be simulated using the creation of links between processes. The RDT model presented here represents a "single run" of the project management process permitting the required processes (one each of the Director, Project Manager and Designer) and interconnections between them to be instantiated before the start of execution.

The remaining action of the Director in the RAD is to "Agree TOR for project". This appears in the RDT Director process as its only event and illustrates another difference between the RDT and RAD notations. In the RAD, this event is shared between the Divisional Director and the (newly created) Project Manager as equals. In RDT such an interaction is modelled as a point to point communication. In such a communication, one party is the sender and the other is the receiver. The sender of a communication is generally considered to have initiated the interaction. In this particular instance, the interaction is modelled as the Director sending a value to the Project Manager since it is felt that the passing of instructions from the Director to the Project Manager will the dominant element of this interaction.

The second process in this model is the Project Manager. As already discussed, in the RAD this process is started by the Divisional Director but for the RDT model, an instance of the process will need to be created and appropriately connected to

an instance of the Director in the model diagram. The first event in the RDT Project manager is to receive instructions from the Director as discussed above. The RAD then shows the Project Manager creating a new instance of the Designer process which has to be handled in similar manner to event in the Divisional Director which creates a Project Manager. The Project Manager then proceeds through a series of interactions with the Designer. The final state of the Project Manager in the RAD shows another of the differences between the two notations. It is labelled "project completed". Labelling of states with descriptions or names is optional in a RAD, but required in RDT.

The final process in the model is the Designer. The designer process as described in the RAD looks quite different from the RDT version. The first reason is that the RAD shows the Designer performing a pair of tasks in parallel which is not included in the RDT notation. In this example, one of the parallel tasks is "Choose Method" and is one of the process' internal actions (which don't exist in RDT either). As with the other internal actions of the processes in the RAD, this one has be abstracted away remaining in the model only as part of the name of the state which would have preceded it ("Choosing Method and Estimating"). Abstracting away the internal actions of the process has also eliminated the looping behaviour near the end of the process.

After generating process descriptions for the processes of the RAD, a further diagram is required to complete the RDT model of the Project Management process. This is the "model" diagram which describes the instances of the various types of processes which are required to build the model and how they are (initially) connected. Not all of this information is available from the RAD, so some has to be inferred or assumed. Figure 14 shows one of the possible instantiations of the Project Management process. There is one instance each of the Director, Project Manager and Designer and these are connected in the obvious way. In this example, the intention of the RAD is easily deduced. However, the picture is less clear when there are many instances of each of the processes. For example, consider the situation after a second occurrence of the external event which starts a new

project. The Divisional Director will respond to each of these by creating a new Project Manager. In turn each of these will create a new Designer. Now we have two instances each of the Manager and Designer processes. Are the Designers bound to interact with the Managers that created them? Or might they be able to interact with any Manager? This kind of information is explicitly stated in the RDT model diagram, but generally has to be inferred from a RAD.

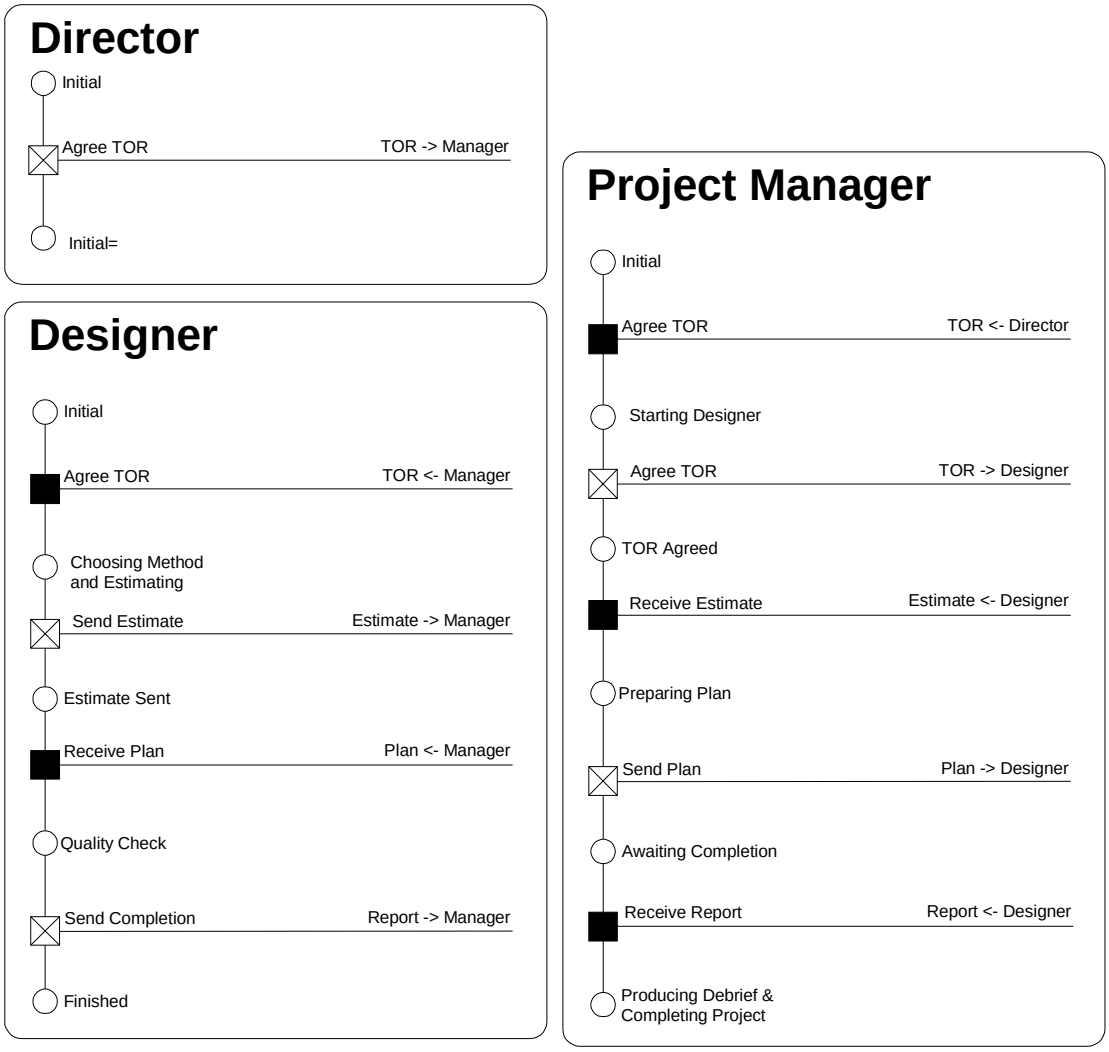


Figure 13: The processes of the Project Management model

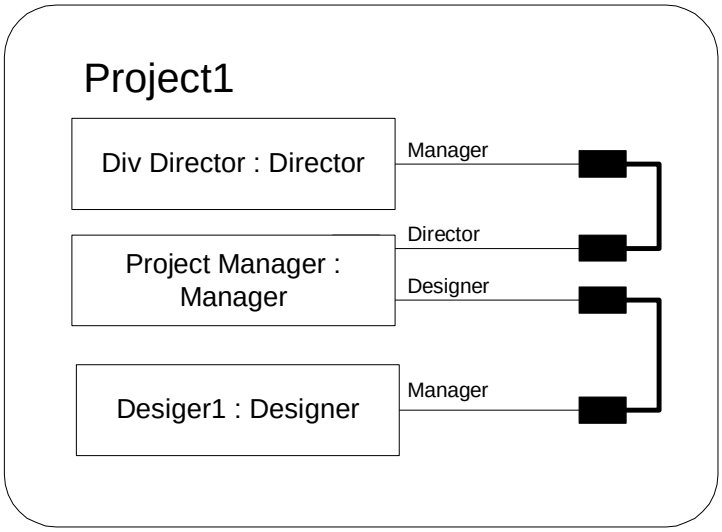


Figure 14: The completed model diagram for the Project Management model

Note:

In the implementation of the RDT tools, the “Send” event and the “Receive” event have been renamed to “Write” and “Read” respectively.

Chapter 5 Practical application of the RDT language

The RDT graphical modelling language as described in Chapter 4 was designed with the objective of making the language appealing whilst at the same time permitting a modeller with limited experience of formal modelling to understand models described using the notation. However, making the reading and understanding of completed diagrams straightforward is only part of the objective of the language. The second major objective is to create a modelling language with which the inexperienced modeller would be able to construct, manipulate and analyse their own models. As described in Chapter 4, constructing the diagrams of an RDT model would be a time consuming task which would require the modeller to learn a collection of rules about what elements should be present in each of the diagrams, how they should be drawn and how they are laid out. Creating the diagrams this way either on paper or using a drawing package is a time consuming task which takes just as much effort as using a text based technique.

In addition to making them approachable for new users viewing completed diagrams, an additional strength of the RDT language is that it has also been designed to be "machine friendly" permitting the construction of tool support which is able to automate many of the time consuming tasks associated with using the language manually. The combination of the user friendly appearance of the diagrams and appropriate tools to support their generation and analysis permits RDT to approach its goal of providing an attractive language and environment in which a "light weight" user might begin to enjoy the benefits of working with formal models after only a small amount of familiarisation.

5.1 Construction of a model

Whilst a modeller might elect to draw an RDT model by hand on paper or with a computer drawing package, this is not how they are expected to work. Instead, they are expected to use a model generation tool. The benefit for the modeller of using such a tool is that it can automate much of the task of drawing the diagrams

releasing the modeller to concentrate on more important aspects of their model. For example, when describing a process what is important to the behaviour of the process is the communication associated with the events in which the process takes part and the way the process proceeds through its various named states (the interconnections between the process' states and events in the diagram). The details of how the events are distributed about the diagram and the order in which choices are presented are relatively unimportant. Relieving the modeller of the burden of the mundane task of arranging elements of diagrams on the page frees them to concentrate on the real task of understanding and describing the system they are studying.

When drawing a diagram of a process by hand, the modeller has to consider how to distribute the various symbols for the states and events of the process as well as making the connections between them, joining or forking lines of control as required. Consideration of the information contained in such a description of a process reveals that, as regards the actual behaviour of the process, the distribution of the event and state symbols in the diagram is irrelevant. What actually matters is the manner in which these items are interconnected and it is not necessary for the modeller to supply this information as it may be inferred from the event descriptions. Consequently, the whole of the behaviour of the process is contained in a full description of its events and, provided the language is amenable, a modeller can construct a description of a process by just supplying details of the events. The task of deciding how to distribute the various elements about the diagram, drawing them, their interconnections and other decorations can be delegated to the tool. The model generation tool can further assist the modeller by managing names such as the state names of a process, or the channels known to a process so that the effort required from the modeller describing an event can be further reduced by providing them with a list of candidate values whenever possible.

Once the process behaviours have been described, instances of these processes are put together in a "model" diagram. Like the description of the processes, this is a task which could be performed by hand, but suitable tool support can assist con-

siderably. Here there is again the requirement to place the various elements onto the diagram, but in this case, where a diagram is to be constructed by hand, the modeller needs to identify the number (and names) of channel connections to be drawn for each process instance. All of this can be handled by a support tool which has access to the process descriptions relieving the modeller of the task and permitting them to concentrate on selecting the correct process instances and making the appropriate connections between them.

A prototype model generation tool, known as the RDT model generation tool is described extensively in Appendix C and Appendix D. This tool provides support to the modeller creating a system model in the RDT language. The tool accepts process descriptions from its user in the form of descriptions of the events. It is able to generate the process diagrams which it keeps up to date as the user adds events to their processes. The tool also supports the generation of model diagrams and is able to save and retrieve these descriptions into file (as XML) in a format which is suitable as input to the other prototype tools, RDX and RDTtoSPIN.

5.2 Analysis of the model

As with the creation of system models using the RDT language, the analysis of the resulting model could also be performed by hand. In reality, some limited informal analysis will be performed manually by the modeller as a part of the model generation process at the times when the modeller is thinking about how they expect their system to operate. However, the whole motivation for building this style of model is to assist the system developer (and modeller) to develop confidence in the behaviour of systems. For this to be convincing, it needs to be more thorough than can be achieved by simple inspection. It is envisaged that a modeller using RDT will perform their analysis in two stages. The first stage will be execution in which the modeller will load their model into an execution tool which will enable them to examine the behaviour of their model interactively. This should permit the modeller to achieve limited confidence the value of their system design. At the least they should be able to satisfy themselves that their

system is capable of behaving as required. The modeller can then proceed to a second level in which the model is subjected to more thorough analysis.

5.2.1 Initial considerations

The definition of the RDT language and its diagrams states that process (instances) communicate by channels. Implicit in this definition is that these channels are "point to point". However, the diagrams are silent on the important matter of whether these channels are buffered. To a large extent, for the purpose of drawing the diagrams and making the connections between the process (instances), this detail of the operation of the channels is unimportant and it is sufficient to the modeller to think in general terms like "A sends X to B along C" without being concerned about the precise details. Indeed, being able to abstract away unnecessary detail is one of the skills which the modeller needs to develop if they are to be build useful models quickly so forcing them to consider low level details of communications during the model generation process is undesirable. However, when the behaviour of a model is to be examined the exact nature of the inter-process communication within the model is a matter which must be addressed.

Most formal modelling paradigms use a form of synchronous communications. This is an easily justifiable approach, particularly as it can be argued that doing so does not restrict the expressiveness of the language in question – buffered communications can be modelled in a synchronous environment by the addition of buffering processes in the appropriate places. However, throughout the development of the RDT language a primary concern has been to make it approachable for the novice and it is felt that these potential users are likely to be working with systems where communications are asynchronous. The need for asynchronous communication in addition to the more normal synchronous model may be a side-effect of the way that fine detail of the real system has been abstracted away or it may be that the system is to be developed and operated in an asynchronous environment (such as some form of message passing middleware). If the language is to appeal to the novice user, they need to be able to establish a simple

mapping from the processes of their model to those of the real system so it was felt that the language should offer asynchronous communications directly.

The selection of synchronous or asynchronous communication has no impact on the way the processes and models are represented in the diagrams, so this decision need not be made at the time that the diagrams are drawn. However, at the point where the model is to be executed (or further analysed), the decision can be delayed no longer and the model execution tool has to elicit from the modeller the communications paradigm to be adopted. An advantage to delaying this decision until execution is that the modeller may experiment with different communications and see how these might affect the behaviour of their system.

5.2.2 By execution

The purpose of executing an RDT model is to permit the modeller to perform some initial analysis of their model. Typically this analysis will consist of running the model through a collection of scenarios designed to show that the proposed system is able to perform as intended. The modeller might also perform a number of "what if" experiments to see how their system responds. This is not an exercise which can be reasonably performed by hand. Tool support in the form of an execution tool is necessary.

An execution tool for the RDT modelling language needs to be able to present the process instances of the model and offer the modeller suitable means to exercise the model. In addition to presenting the processes, since it is a requirement of the language that communications between process instances may be asynchronous, the execution tool needs to provide suitable buffers to hold those values which have been written into channels but which have not yet been read.

The RDX tool which is described extensively in Appendix C and Appendix D provides an execution environment for RDT models. This tool is able to read the XML files written by the RDT model generation tool. The tool uses an interface inspired by that of the RolEnact execution tool. Each process instance in the

model has its own window within the application which shows the named state of the process together with a list of the names of the channels it knows and the events which it is presently able to initiate. Each channel also has a window in which those values written into the channel but not yet read out are listed. The user makes their selection of the length of the channels by a menu choice before starting execution and then drives the model by "double-clicking" on the event they wish to see occur. The tool then actions the effect of the selected event and updates the displays of all the process instances and channels.

If the user elects to set the length of the channels to zero, the communications between the processes instances in the model become synchronous. During execution of the model this appears as the tool forcing the execution of the model to proceed in a series of paired events with each pair of RDT events corresponding to a single communication event in model. At the start of execution, the only events which may be performed will be those which cause a value to be written into a channel subject to the condition that no such event is enabled unless there is another process instance which is in a condition which will permit it to read the value. Once a value has been written into a channel, the model moves to an intermediate state in which the only events which are permitted are those which will remove the freshly written value from the channel, thereby completing the communication. Since each of these pairs of events forms the two halves of a single event (and no process can do two things at once), the process instance which reads the value from the channel cannot be the one that wrote the value in.

5.2.3 More comprehensive analysis

Whilst a modeller may enjoy benefits from executing their model, or even from the exercise of constructing the model itself, the real benefits of building a formal models accrue from subjecting them to more rigorous analysis. There is already a considerable body of work directed at building systems for the automated analysis of formal models so it was felt that it would be inappropriate for this work to attempt to construct such a tool. Instead, the RDT language is designed so that models constructed using the language are sufficiently precise to permit them to

be transformed into the input languages of existing “model checking” software tools. It is anticipated that this will be the manner in which a modeller will perform the final analysis of their models.

Since the objective of creating the RDT language is to provide an easy introduction to the task of building formal models and obtaining useful results from analysing them, it has to be expected that the target users will have at most, minimal experience of using model checking software so the transformation of a model from the diagrams of RDT to the input language of the chosen model checking tool needs to be automated.

Chapter 7 discusses the selection of a model checking software and describes one feasible transformation from RDT models an input language for a model checker. The RDTtoSPIN tool (which is described in Appendix C and Appendix D) performs this transformation automatically using the XML generated by the RDT model creation tool as its input.

5.3 Presentation of a model

The main focus of attention of the work developing the RDT language has been to create a language with which a novice modeller can construct and analyse a model with a minimum of initial knowledge. However, formal analysis is not the only motivation for developer to create a model of system. Models, since they describe the essential essence of a system and how it is intended to operate make an ideal medium for recording and communicating this information.

The RDT language was designed with the primary objective in mind of providing a modelling language which would be attractive to the inexperienced modeller. A side effect of the effort directed at making the diagrams accessible for these potential users is that they are also accessible to others making a model built using a collection of RDT diagrams immediately suitable for presentation to a non-technical audience.

Chapter 6 A translation into the pi-calculus

This chapter demonstrates that the semantics of communication in RDT are sound and closely related to those of the pi-calculus by describing a translation from RDT to the pi-calculus. The pi-calculus was taken as the pattern for the RDT language and tools because it has a number of features which make it attractive for our purpose:

1. Its simplicity. Despite its power, the essential core of the calculus is quite small.
2. It uses a channel based scheme of communication based upon the notion of names. As discussed elsewhere in this document, it was felt desirable for the RDT language to use a channel based communications paradigm because this fits more comfortably with the way that the target users think about communications.
3. Since names are not distinguished from other data, channels (names) may be passed about between processes. At the core of the motivation for RDT was the desire to construct a language which is powerful enough to be useful whilst remaining small enough for a novice to learn its features in no more than a few hours. However, it was felt that users would want to be able to investigate the behaviour of systems which were able to perform run-time reconfiguration. In order to do this, the language needs to have suitable mechanism such as the ability to pass channels between processes of the pi-calculus.
4. It has a strong formal foundation.

6.1 An outline of the pi-calculus

The pi-calculus is described fully elsewhere [Milner 1993] so here we only give a short summary of its features. The pi-calculus models systems as collections of communicating processes. Communication between these processes is synchronous and point to point. The calculus makes no distinction between channels or

other data consequently channels may be passed along channels. In the "monadic" pi-calculus described here, the values passed between processes in communications are simple single values. These values are referred to as "names".

The most elementary process description in the pi-calculus is the empty process which does nothing and is written 0. Where the context permits, it is often simply omitted. This process may be combined with one of two forms of communication to create slightly more useful processes:

$A = p(r).0$

$B = \overline{p}s.0$

Process A above reads a value (or name) from the channel it knows as p, associates it with the "name" r, and then behaves as 0 (does nothing). Similarly process B writes the value which it knows as s onto the channel it knows as p and then behaves as 0. These would normally be abbreviated to:

$A = p(r)$

$B = \overline{p}s$

Processes may be further elaborated by the addition of more communication events and choices of action:

$C = p(v).\overline{v}n + \overline{p}n$

The process C has a choice of paths of execution shown as a sum. C has just two options. The notation permits any number of terms is such a summation, and hence any number of alternative execution choices. Here the choice for C is between reading a value on channel p (which is associates with name v) and then writing n to channel v, or writing value n onto channel p. Once the choice is made to follow one of the possible paths of execution all others immediately and irretrieveably disappear.

The three processes A, B, C may be combined into a complete system comprising one instance each of A, B and C operating in parallel as follows:

$$S = A \mid B \mid C$$

which is equivalent to:

$$S = p(r) \mid \overline{p}s \mid p(v).\overline{v}n + \overline{p}n$$

Examining process S we see that it has a choice of actions:

- Process B writes s into channel p which is read by Process A which calls the value received r.
- Process C (following its second alternative) writes n into channel p which is read by Process A which calls the value received r.
- Process B writes s onto channel p, which is read by process C taking the first of its alternative paths.

In the event of the first described communication happening on channel p, the result is:

$$0 \mid 0 \mid p(v).\overline{v}n + \overline{p}n$$

Processes A and B are reduced to “0”. Process C did nothing and is unchanged.

The occurance of the second alternative leads to the following:

$$0 \mid \overline{p}s \mid 0$$

Process A is reduced to 0. Process B is unchanged (it did nothing). Process C has taken its second alternative and also reduces to 0, since the events of the first alternative are lost irretrievably as soon as the second alternative is chosen.

Should S indulge in the third of the possible communications on channel p, the result would be:

$$p(r) \mid 0 \mid \bar{s}n$$

Here, process A did nothing so is unchanged. Process B is reduced to 0. Process C has performed the first action of its first alternative which caused v to become associated with the value read from channel p. The result is that the remaining action available to C becomes to write the value n onto the channel s (the newly acquired value for v).

There is one further device which we need for the following representation of RDT in the pi-calculus, (vi)P ("new i in P"). This is the way that fresh, unique names are created in the pi-calculus. Consider the following process:

$$S' = A \mid (\nu p)(B \mid C), \text{ or}$$
$$S' = p(r) \mid (\nu p)(\bar{p}s \mid p(v).\bar{v}n + \bar{p}n)$$

S' is looks similar to S apart from the inclusion of the (νp) operator and the following parenthesis which indicate the extent of its scope. The effect is to declare that the p referred to within the parenthesis is unique and distinct from all other p. The consequence for S' is that the only communication which can occur is that in which process B writes to channel p and process C takes its first alternative. The reason being that p referred to in process A is not the same as the p referred to in the remainder of S'.

More formally, processes and systems in the pi-calculus are built using the following syntax:

$$P ::= \sum_{i \in I} \pi_i.P_i \Big| P \mid Q \Big| !P \Big| (\nu x)P$$

Each of the parts of the syntax has the following meaning:

I is a finite indexing set. Where I is empty, the process is unable to perform any actions. Such a process may be written as 0 although, where the context allows, it is often simply omitted. An element $\pi.P$ in the summation describes a process which performs the action π followed by becoming the process P. The summation describes a process which may perform just one of the possible actions π_i followed by becoming the process P_i. There are just two types of prefix: $x(y)$ which means, "input some value on x, call it y" and $\bar{x}z$ which means "output the name z on x". These prefixes are the means by which processes communicate. The nature of the communication deserves some further explanation:

Communication is synchronous and "point to point" so prefixes happen as simultaneous pairs of the opposite types (read and write) acting on the same "name" or channel.

P|Q processes P and Q run in parallel.

!P means P|P|P|P|P|P… for as many copies of P as required/desired. This is not used in this work.

$(\nu x)P$ declares a new, local name x which is available for use within P.

There are other features in addition to those described here, including polyadic version of the calculus in which a message to be passed along a channel may (and likely does) comprise an ordered list of names instead of the single value available of the monadic version described. This could be implemented using the monadic calculus.

6.2 A mapping from RDT to Pi-calculus

The language of RDT uses a pictorial representation of processes and models in place of the more usual code and is divided into a collection of diagrams. There is one diagram which describes the behaviour of each type of process in the system and a further diagram which describes the instances of the types of process in a particular model and the interconnections between them.

RDT makes a sharp distinction between the description of the behaviour of a type of process in general from the particular collection of instances of processes (and connections between them) which form a particular system under consideration. As with other calculi and modelling methods, the pi-calculus does not make this same sharp distinction. This mapping between RDT and the pi-calculus will be in two parts first showing how a description of an RDT process could be represented in the pi-calculus and then how the particular description of a model built from a collection of instances of these processes might be represented in the pi-calculus.

6.2.1 RDT processes to pi-calculus:

A process in RDT is primarily described by its diagram (and stored as XML). Unlike pi-calculus processes, RDT processes have explicit names for their state. However, since the behaviour of the process is dependent on its state, each process could be considered as a collection of pi-calculus processes each of which carries out one action and then behaves as another process from the collection. The first of these processes would be named after the RDT process and would offer the choice of events available RDT in the "initial" state. To ensure the names are unique, the other processes are named after the RDT process of which they are part, subscripted with the name of the state from which they derive. A final RDT state (one which is not the "before" state of any of the processes events) is equivalent to the pi-calculus event "0". This is the process which does nothing. Where the context allows, “0” is often simply omitted from pi-calculus descriptions.

Converting from an RDT representation of a process to pi-calculus, comprises writing a collection of processes containing one process for each state of the RDT process as follows:

For a process P in state a which, after writing “z” onto “x” moves to state b:

$$P_a = \overline{x}z.P_b$$

A process P in state c which is only able to read a value from channel “x” to be called “y” and moves to state d could be written as:

$$P_c = x(y).P_d$$

A process P in state e which is only able to perform a "create" (generating a new name, “k” and writing it onto channel “x”) and then moves to state f would be:

$$P_e = (\nu k)\overline{x}k.P_f$$

A process P in state g which has no events which leave state g would be:

$$P_g = 0$$

Where there is more than one event leaving a state, the required process is a summation of each of the processes corresponding to each of the events:

For example; a process P in state i from which it is able to either write a value “z” on channel “x” followed by moving to state j or can read a value which it calls “q” on channel “x” followed by moving to state k would be written as:

$$P_i = \overline{x}z.P_j + x(q).P_k$$

6.2.2 RDT "models" to pi-calculus:

An RDT "model" describes a particular collection of instances of processes and an initial scheme of connections between them. There is no direct equivalent in the pi-calculus. Instead, the equivalent of the model is described as a composite process built from others. The modeller then elects to create an instance of this process which is otherwise undistinguished from any other. So, to create a representation of an RDT model in the pi-calculus, we have to construct a single process within which are the processes listed in the model arranged so that their initial sharing of names matches with that described in the model.

This is more involved and involves using a mechanism for creating the required sharing of names.

It is easiest to describe how this is achieved with an example. Consider a model consisting of a pair of process as described in Figure 15, Figure 16 and Figure 17. This shows a simple model in which there is just one instance each of two equally simple processes.

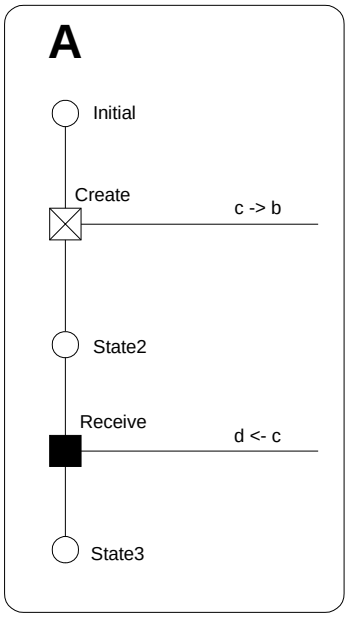


Figure 15: Process A

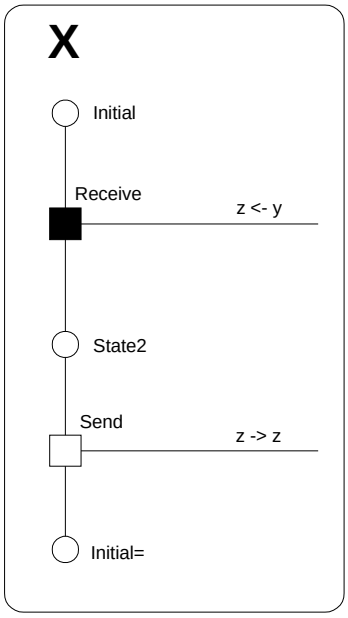


Figure 16: Process X

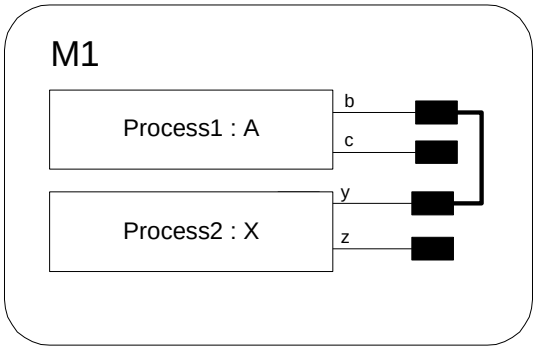


Figure 17: Model M1

Process A creates a new channel which it calls “c” and writes into the channel it knows as “b”. It then reads a value on its channel “c” which it calls “d” and stops.

Process X performs actions complementary to process A. It first reads a value, which it calls “z”, on the channel it knows as “y”. It then sends back the channel it now knows as “z” on “z” and returns to its initial state, ready to repeat its actions.

The model M1 shows one instance of process A, called Process1 and one instance of process X called Process2. It also shows that, at the start of execution, the

channel known to Process1 as “b” is connected to (or shared with) the channel which process Process2 knows as “y”.

So, assuming synchronous communications, execution of this model will consist of event Create of Process1 and event Receive of Process2 occurring as a pair followed by event Receive of Process1 and event Send of Process2. The system then stops with Process1 in a final state and Process2 ready to repeat the sequence.

Following the scheme outlined above, the two processes A and X could be described as follows:

$$A = (\nu c)\bar{b}c.A_{state2}$$
$$A_{state2} = c(d).A_{state3}$$
$$A_{state3} = 0$$

$$X = y(z).X_{state2}$$
$$X_{state2} = \bar{z}z.X$$

The remaining task is to create a composite process which has the behaviour of M1. For this we need to create a process which combines one instance each of process A and process X in such a way that the channel known to the instance of process A as b is the same channel as that known to the instance of process X as channel y. We could achieve this, for example by means of a re-naming “y” to “b” in process X, but this is hard to generalise. An alternative is to parameterise processes A and X and then arrange the required "connections" by selecting suitable parameters:

$$A(b) = (\nu c)\bar{b}c.A_{state2}(b,c)$$
$$A_{state2}(b,c) = c(d).A_{state3}$$
$$A_{state3} = 0$$

$$X(y) = y(z).X_{state2}(y,z)$$
$$X_{state2}(y,z) = \bar{z}z.X(y)$$

$$M1 = (\nu g)(A(g) \parallel X(g))$$

In the parameterised version of process A just the channel “b” is supplied in the first line of the definition. The channel which process A comes to know as “c” is created by “νc” in the first event. For clarity, “c” is then passed on to process A_{state2} as a parameter along with “b”. In this particular example, “b” could have been omitted since it is not used again. A_{state3} needs no parameters as it doesn’t do anything. We also know that the name "z" which appears in the definition of process X acquires its value as an effect of the process reading another channel, so there is no need for it to be supplied as a parameter to process X but, having acquired a value it is passed on to process X_{state2}.

In a more general conversion, all of the these various names known to each process would appear as a parameters to each definition, despite the fact that the values of some would be overwritten before they are used and others would never be used at all. A technique similar to this is used in the translation of RDT models to Promela described in Chapter 7.

Chapter 7 Transforming RDT models for analysis

7.1 Selecting a target model checker

The first task in finding a transformation from RDT to the input language of a model checker is to select the target model checker. In principle, since the models described by RDT in its diagrams are finite state machines, it would be possible to use any one of many model checking tools to analyse its models. Two model checking tools stand out as potential candidates for this work, FDR ["FDR2 User Manual" 2000] and SPIN [Holzmann 1997]. 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 examined is specified.

FDR uses a variant of CSP [Hoare 1985] as its input language. The language is powerful and fully featured but would not look familiar to a programmer. Communication is via channels which are typed as to the type of value which they are permitted to carry though the communication is synchronous. By contrast, SPIN uses its own input language and whilst its meaning is different, it has syntax reminiscent of the “C” programming language. This language is outlined later in this chapter.

FDR and SPIN also vary significantly in the way that the properties to be examined are specified to the tool. Both provide useful analysis by default. For example, given a model either tool will examine it for deadlock. For more specific analysis however, their approach to specifying the problem is different. SPIN takes a model and some additional code which describes the negation of the property to be checked (though it does help with the construction of this code). It then demonstrates the truth (or otherwise) of the desired property by attempting to establish that, in the context of the model this negation can never become true. FDR takes two models, the one under inspection and another which has the desired

property (perhaps trivially) and establishes if the one is a “refinement” of the other and hence has the properties sought.

After consideration the SPIN model checker was selected as the target for an automated transformation of RDT models in this work for the following reasons:

- At some point the modeller is likely to have to relate the code generated for the model checker to their own model and it is felt that whilst our target users are likely to be familiar with programming languages, they are unlikely to have encountered process algebras. Consequently they are likely to feel more comfortable with programming language like Promela code than the CSP-like 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 who are also unlikely to be familiar with the notion of “refinement” used in FDR.

- In Promela, when channels are defined their length is specified and the buffers required if the length is non-zero are implemented directly by SPIN providing a natural relationship between the channels of RDT and the channels in Promela.

- The SPIN model checker is available free of charge in versions for use on several platforms, including Windows, the preferred platform of many commercial software developers.

7.2 An outline of SPIN and Promela

Using SPIN is a multipart operation involving the following steps:

1) Construct a model by writing code in Promela, the input language of SPIN.

2) Work with this code and SPIN/XSPIN to check the syntax of the model, execute the model interactively.
3) Add an appropriate “never” clause to the code if a particular property is to be verified.
4) Generate a customised checking program for the model and run it.
5) Analyse the results from the verifier as required. This operation may include using SPIN to assist in following traces of counter-examples generated by the verifier either directly or via XSPIN.

The Promela language has a syntax which is reminiscent of the “C” programming language, though there are differences in the syntax and the semantics. Promela is a powerful language with many features. Here we describe just the most fundamental elements.

In Promela, the modeller is required to declare variables in a similar style to that used in a “C” program. In contrast with RDT which has just the single type of data item which serves as data, a channel or both according to the context, Promela has a collection of numeric types varying in size from a single bit to a (“C” style) “int”. In addition, the modeller can declare “channel” type variables which are the means by which Promela processes are able to communicate. In the declaration of a channel the modeller is required to state the length of the channel and the type of variable which is to be transferred along the channel. One particular feature of channels is that it is permissible to declare a channel which carries items of type channel. The modeller wishing to do so can also declare their own data types built from these elements.

A model written in Promela consists of a collection of process descriptions. A process description in Promela looks a bit like a function in “C” preceded by the keyword, “proctype” (in place of the return value in a function):

```
proctype process_name(  parameter list )
{ statements }
```

The parameters to a process are named and typed in a fashion similar to a “C” function definition.

Within the body of the process description a series of statements describe the behaviour of the process. Statements are separated equivalently either by a semi-colon, ; (like “C”) or by an arrow “->”.

Where Promela departs significantly from the “C” programming language is in control constructs. We consider just two of those available:

7.2.1 IF

The syntax for “if” in Promela is most easily described by giving an example:

```
if
:: statements1
:: statements2
…
fi
```

The example above enables the process to select between executing either the collection of statements “statements1” or the alternative collection “statements2”. In contrast with the evaluation of an “if” statement in an imperative programming language, which of these collections of statements offered as alternatives within this statement will be executed depends on the first statement in each collection. At the time the “if” statement is evaluated, the first statement of each of the branches is checked to see if the action it represents is possible. Any branch for which the first statement in its list of statements is possible may be chosen, not just the first. Also, when the SPIN is used to perform a full check of the model, when it encounters such a statement where more than one subsequent path of execution is possible, as part of its work SPIN will explore all of them.

7.2.2 DO

The syntax of a “do” statement in SPIN is as follows:

```
do
:: statements1
:: statements2
…
od
```

The meaning of the “do” statement is similar to the “if” statement described above with the exception that SPIN continues to execute the do statement until control is transferred outside explicitly by a break.

The syntax of Promela also permits the modeller to place labels in the code and execute a jump to one of those labels using a “goto” statement.

7.2.3 Communications in Promela

Since variables may be declared with global scope, processes in Promela may communicate by sharing such variables. However, Promela also provides communications channels. As indicated above, these channels are typed according to the type of data which is passed along them. The types of data which channels may carry includes user defined types and channel types. For this work, we use just one type: channels which carry channels.

A process which wishes to write a value “v” into a channel “c” does so as follows:

```
c!v;
```

A process wishing to read a value “v” from a channel “c” does so as follows:

```
c?v;
```

Promela channels are of a fixed length which is determined at the time that they are declared. Zero length channels are permitted.

7.2.4 Initialising a model

There is a distinguished process name of “init”. When SPIN starts to run a model, if it finds a process type called “init”, it will create and run a single instance of this process. Although it is not precluded from any of the actions of other processes, the typical “init” process is used by the modeller to start the required instances of the other processes. (There are other mechanisms for the creation of the initial processes in a model.)

7.3 Considerations in mapping from RDT to Promela

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 choice would be "guarded" by a conditional dependent on the current "state" of the process and the availability of the required communication. However, this scheme is unsatisfactory for two reasons in particular:

SPIN regards a "do" loop as a single statement. Consequently SPIN regards a process created in the manner outlined above as having a single statement and a process 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 in some way which is likely to make interpretation difficult for the human reader looking at analysis results generated by SPIN and trying to relate these to the original model and the generated Promela code. (Promela does have "symbolic constants" which could be used, but just one declaration of this type is per-

mitted in each file, so all of the states of all of the processes would have to be declared as a single collection.)

The solution eventually adopted was to use of labels and explicit "goto" statements in the Promela code. Each of the labels in a process description corresponds to a state of the process and the labels are constructed from the process state names making it easy to relate lines in the Promela code to the state of the process at the time the line is executed.

A further problem which needed to be solved was how arrange the (Promela) channels through which processes will communicate. Each process has a number of names for channels. Each of these may be associated with an actual channel when execution starts as a consequence of a "connection" in the "model", but there is no requirement for this to be the case and usually, at least some of the channel names known by a process do not refer to a channel at the start of execution. This in not an error as they may become associated with channels as the consequences of a read or create event during execution. However, Promela does not permit names of channels to be used (written into or read from) in the description of a process unless they are suitably declared. This declaration may be global, within the process or the channel may be passed to the process as a parameter. Declaring variables for channels which are not connected at the commencement of a process as local variables within the process can be 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 make coping with a "model" in which different combinations of channels are the subject of connections in different instances of the process. So the solution adopted was to supply all of the names used by a process as parameters, thereby eliminating the need to identify which need to be parameters and which could be declared within the process. Where a process has names which are not initially connected, because Promela does not permit the passing of "null" parameters, the process is supplied with a placeholder channel name.

7.4 How RDTtoSPIN performs its conversion

A description of a system in Promela can be viewed in two parts. There is the description of the constituent processes and the description of the "init" process. In Promela, init is a distinguished process. When SPIN loads a system description, it creates a single instance of the init process. This is one way SPIN starts a system and the "init" process approximately corresponds to the creation of a "model" in RDT.

The RDTtoSPIN tool takes an XML file describing a model and creates a Promela process description for each process description in the model. The names of the states of the process are used to create labels in the code. After each label, there is an "if" statement. Within this "if" statement is a branch for each of the events which takes this process from this state. The statements in the branch correspond to the actions associated with the event. The final statement of each branch is a "goto" statement which moves the point of execution to the label corresponding to the "after" state of the event which has occurred. Since each state in the RDT model maps to a label and some associated code in the Promela, when the code is analysed using SPIN it is easily able to identify unvisited states of the RDT model. Also, as the labels used are closely related to the state names in the original model (usually these are the state name with a ":" suffix), the task of relating problems identified by SPIN to the RDT model is simplified.

The init process created by RDTtoSPIN is used to create the initial configuration of the model. It is in three parts:

1) It declares channels for each of the channels required to make the connections between the processes.
2) It declares zero length channels to be used as placeholders where they are required. A separate channel is needed for each name which is not part of a connection to ensure there is no possibility of accidentally permitting a communication to occur though one of these channels.

3) A sequence of "run" statements which start the required instances of the processes described earlier in the file, supplying them with their names of channels as parameters. Promela does not allow "null" type parameters to be supplied to a process so, where a process has name which is not initially associated with a connection, one of the placeholder channels is used.

All of the statements in the "init" process are enclosed in an "atomic" statement to indicate to the model checker that they should all be performed as if they were a single indivisible action. This it to ensure that, when performing checks on our model system, the whole system (all of its processes) are created before any part of the system starts to operate.

7.5 The problem of "Create"

RDT includes a type of event, called "Create" in which a new channel is created. Should an event of this type occur inside a looping behaviour then, in an indefinite execution of the model, an unlimited number of channels will be created. However, the version of SPIN to which this translation is targeted does not permit the dynamic creation of channels. Consequently, all of the channels which are to be used in the model during its execution need to be declared (created) before execution of the model commences.

The immediate solution adopted in the RDTtoSPIN tool is to give any process which contains a create-type event 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 size of the supply of channels is sufficiently large in the context of the model, this solution should not impact on the behaviour of the model. A more sophisticated solution to the problem which is not yet implemented would enable the translation tool to construct a Promela model in which it could be assured that

the behaviour of the model would not be affected by the construct adopted in the translation of the model from RDT.

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 – and if the name to which the newly received value is assigned already refers to a channel, the existing value is overwritten. A consequence of overwriting existing channel names with new ones 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 useable channels than there are local names for them in the process instances of the model. Consequently, it is feasible for the translation tool to implement 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.

7.6 One further minor matter

RDT permits a process to read a value on a channel and assign the name received to the name used, but SPIN will not allow this directly:

The event description shown in Figure 18 is legal in an RDT description. It causes a value to be read on the channel known to a process of type Proc1 by the name X. The value which has just been received is then associated with the name X.

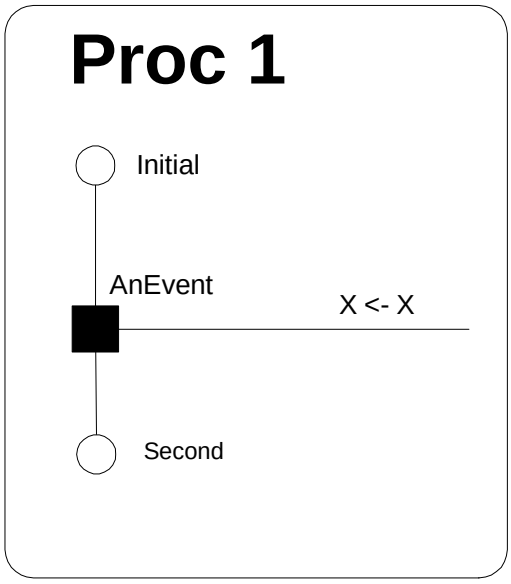


Figure 18: Process Proc1

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. To eliminate this problem, the RDTtoSPIN tool substitutes the following code:

```
chan tmp;
…

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

The tool is able to identify which processes exhibit this behaviour and only declares the channel tmp if it is needed. Tmp is re-used if a process exhibits this behaviour in more than one place. The two statements in the amended version of the code are enclosed in an "atomic" statement to indicate to the model checker that these actions are to be performed as if they were a single indivisible action.

Chapter 8 Communications in models

Communications between processes in RDT are made by passing values (like "names" in the pi-calculus) which can be used as channels for further communication although where appropriate, they may be regarded as simple values such as string or an integer. In RDT, these channels can be buffered and the modeller is permitted to select the length of the channels in a model at runtime (or translation time if the model is being converted to Promela). As discussed elsewhere in this document, this flexibility in the style of communications is offered to help modellers to find what feels like a natural mapping from a model to the real application based on the observation that many systems are being built using infrastructures which provide asynchronous communications. However, although asynchronous communications are provided by the RDT execution tool, the same behaviour could be described using synchronous communications (zero length buffers) by adding the asynchrony to the model explicitly in the form of one (or more) buffer processes interposed between communication processes.

This chapter describes a simple communication between two processes in RDT where they communicate either through a buffered channel or via buffer processes using zero length channels. The results of analysis of these models by SPIN after automated conversion to Promela by the RDTtoSPIN tool are compared with equivalent results for models constructed in FSP and analysed using the LTSA tool.

8.1 A model demonstrating communications in RDT

The communication chosen for this example is the simplest possible. A source process sends the same value many times to a sink process which does nothing but receive the value as often as it is sent. Figure 19 and Figure 20 show these two processes.

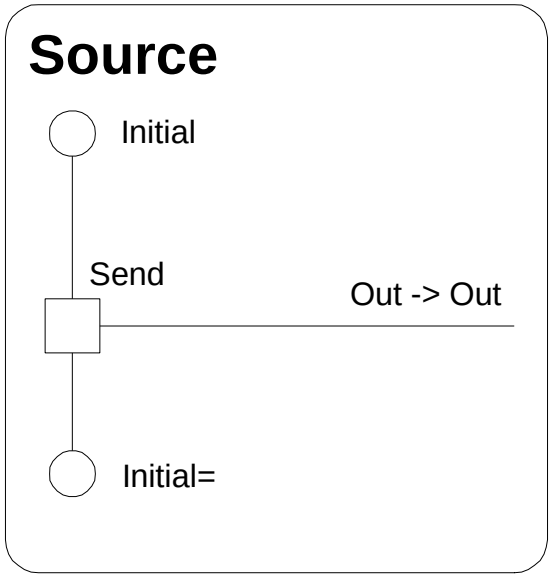


Figure 19: The “Source” process in RDT

These two processes are combined directly in the simplest model as shown in Figure 21.

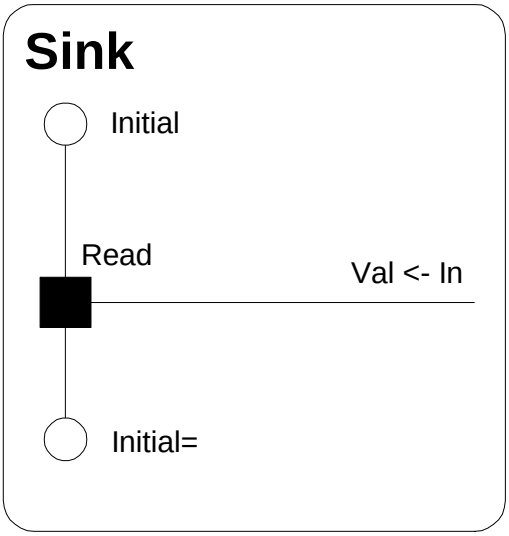


Figure 20: The "Sink" process in RDT

In addition to the “Source” and “Sink” processes, there is a “Buffer” process which is shown in Figure 22. One or more of these buffer processes can be inserted between the sink and source processes to simulate the presence of buffer

spaces in the channel between the processes. Figure 23 shows a model with two of these “Buffers” between the source process and the sink process.

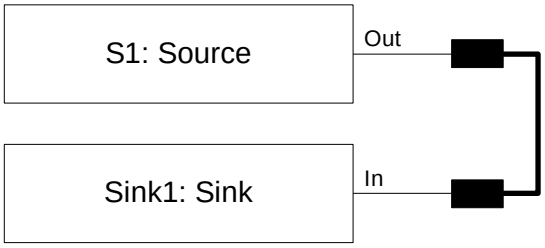


Figure 21: Source and Sink connected directly

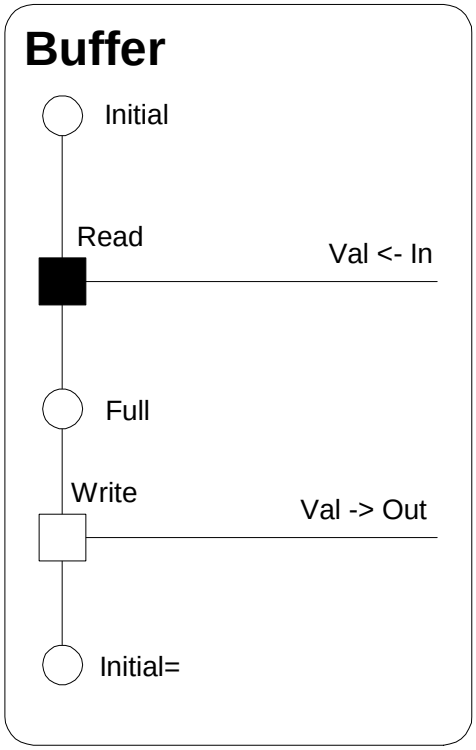


Figure 22: The "Buffer" process in RDT

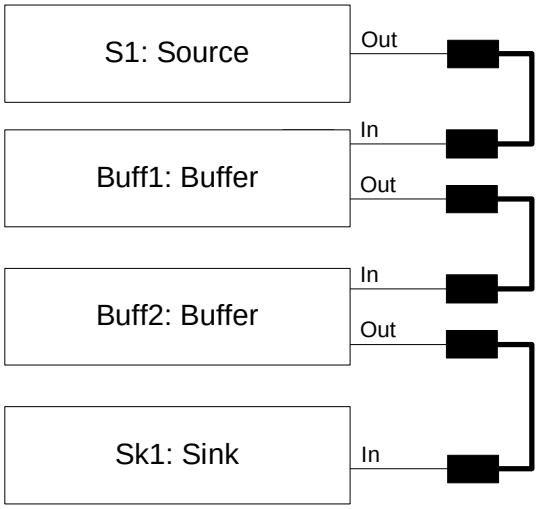


Figure 23: A model with two buffers inserted between the Source and the Sink

Taking the model in which the source and sink process are connected directly, and the using SPIN to analyse the Promela produced from these models by the RDT conversion tool for various lengths of channel gives the numbers of states and transitions shown in Figure 24. Clearly, these figures include some states and transitions which relate to the establishment of the model, but the pattern is evident: for each additional buffer place which is added to the channel, the model gains two states and four transitions.

Channel Length	Number of States	Number of Transitions
0	3	4
1	5	6
2	7	10
3	9	14
4	11	18

Figure 24: Table showing number of states and transitions reported by SPIN for the simple model according to the length of the channel

Repeating the experiment using zero length channels and interposing 0,1,2,3,4 buffer processes between the source and sink processes produces the results shown in Figure 25. The first two lines of this table are the same as Figure 24, but

thereafter, it is clear that these models have more states and transitions than those in which the buffering between the processes is implemented by a buffered channel. The number of states increases by a power of 2 as each additional buffer is added. The pattern for the number of additional transitions is not immediately apparent. However, it is obvious that the additional states and transitions of the models in which the buffered channel is implemented as processes arise from the cascading of values through the buffer processes on their way from the source process to the sink process.

Number of Buffer Processes	Number of States	Number of Transitions
0	3	4
1	5	6
2	9	12
3	17	26
4	33	58

Figure 25: Table showing number of states and transitions reported by SPIN for the example using buffer processes and zero length channels

8.2 Building an equivalent model in FSP

Figure 26 shows a simple interpretation of our model as FSP code. In this code, the required interactions between the processes which are combined to form the compound processes are obtained by suitable re-naming of events. The compound processes are then further refined by "hiding" those interactions which are concerned only with the passing of a value between buffer elements. When LTSA generates composite processes, the names of any events which have been identified as "hidden" are still shown on the diagram, but they are renamed "tau". When instructed to "minimise" the process, LTSA eliminates these "silent" actions and any unnecessary states from the process to give a new process which has the same observable behaviour, but with fewer states and transitions.

Figure 27 shows the diagrams which LTSA generates for the source, sink and buffer processes of the code in Figure 26, together with the diagram for the compound process “SYSTWO” in which two buffer processes are interposed between the source and sink processes. The diagram created by LTSA for the “minimised” SYSTWO process is shown in Figure 28. With the events corresponding to the movement of values between the various buffer processes hidden, it might be expected that the number of states and transitions of the un-minimised version of the model would correspond with those reported by SPIN for the RDT model in which the Source and Sink process communicate through zero length channels with buffering in their communication introduced by the insertion of buffer processes. Similarly, those of the minimised process would be expected to be related to the numbers of states and transitions generated by SPIN from the versions of the RDT model in which the source and sink process communicate through a single channel.

```
SOURCE = (send -> SOURCE).

BUFF = (read -> write -> BUFF).

SINK = (receive -> SINK).

||SYSNONE = (SOURCE || SINK)
            /{receive/send}.

||SYSONE = ( SOURCE || BUFF || SINK )
            /{read/send, write/receive}.

||SYSTWO = ( SOURCE || a:BUFF || b:BUFF || SINK )
            /{send/a.read, a.write/b.read, receive/b.write}
            \{a.write}.

||SYSTHREE = ( SOURCE || a:BUFF || b:BUFF || c:BUFF || SINK )
```

/{send/a.read, a.write/b.read, b.write/c.read, receive/c.write}
\{a.write, b.write}.

||SYSFOUR = (SOURCE || a:BUFF || b:BUFF || c:BUFF ||d:BUFF || SINK)
/{send/a.read, a.write/b.read, b.write/c.read, c.write/d.read, receive/d.write}
\{a.write, b.write, c.write}.

Figure 26: FSP code for a simple interpretation of RDT model

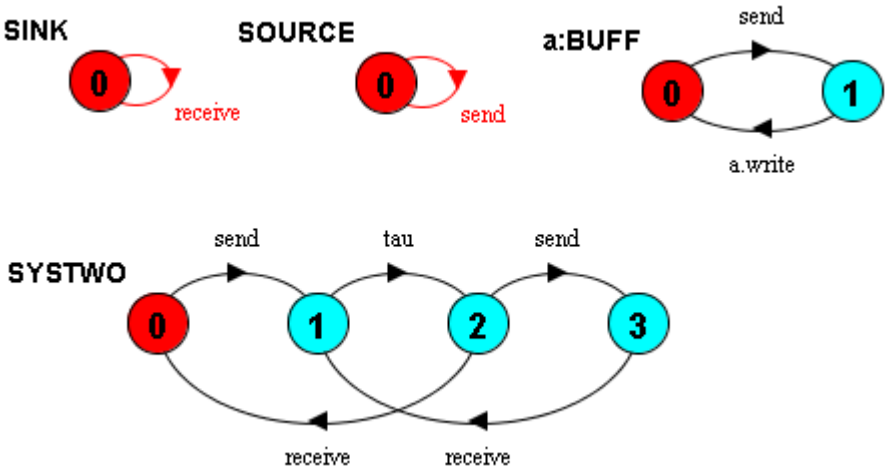


Figure 27: LTSA generated diagrams of simple interpretation of model.

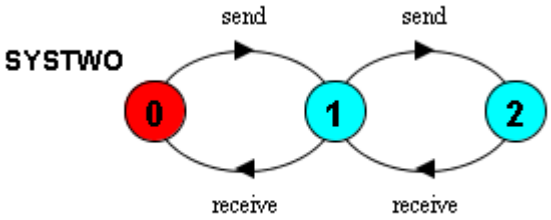


Figure 28: LTSA generated diagram for "SYSTWO" after minimisation

	Without Minimisation		Minimised	
No. Buffers	No. States	No. Transitions	No. States	No. Transitions
0	1	1	1	1
1	2	2	2	2
2	4	5	3	4
3	8	12	4	6
4	16	28	5	8

Figure 29: State and transition counts for simple LTSA model

8.3 An improved model in FSP

When analysed using LTSA, the five compound processes produce numbers of states and transitions shown in Figure 29. These do not match those in Figure 24 and Figure 25. Further inspection reveals that the buffer and sink processes used in this first LTSA model are too simple. They do not match the actual behaviour of the RDT model. Taking the "Sink" process as an example. This process starts execution in a state which is named, "initial". It then reads a value from the channel which it knows and returns to the state named, "initial". However, at this point in execution, although the process state is once again named, "initial", the process is not the same as it was at the start of execution: label "Val" has acquired a value (read from the channel "In") which it did not have last time the process was in this named state. As execution continues, the process proceeds to read many times from the channel "In", but because the source process in the model always writes the same value, the sink process always reads the same value and so never proceeds to any further new states. This distinction between is recognised by SPIN when it analyses the Promela code generated from the RDT model. A similar argument applies to the buffer process. The code for a revised model is shown in Figure 30 and the diagrams for the revised buffer and sink processes in Figure 31. The names of events (read, write, receive) in which the Sink and Buffer processes take part after their initial change of state are "decorated" with a "1" to distinguish them from the first occurrence. Making this distinction between

these events explicit in the name of the event assists in the subsequent composition of the processes into the revised compound processes.

```
SOURCE = (send -> SOURCE).

BUFF = (read -> write -> BUFF1),
BUFF1 = (read1 -> write1 -> BUFF1).

SINK = (receive -> SINK1),
SINK1 = (receive1 -> SINK1).

||SYSNONE = (SOURCE || SINK)
             /{receive/send, receive1/send}.

||SYSONE = ( SOURCE || BUFF || SINK )
             /{read/send, read1/send, write/receive, write/receive1}.

||SYSTWO = ( SOURCE || a:BUFF || b:BUFF || SINK )
             /{send/a.read, send/a.read1, a.write/b.read,
a.write1/b.read1, receive/b.write, receive1/b.write1}
             \{a.write, a.write1}.

||SYSTHREE = ( SOURCE || a:BUFF || b:BUFF || c:BUFF || SINK )
             /{send/a.read, send/a.read1, a.write/b.read,
a.write1/b.read1, b.write/c.read, b.write1/c.read1, receive/c.write,
receive1/c.write1 }
             \{a.write, a.write1, b.write, b.write1}.

||SYSFOUR = ( SOURCE || a:BUFF || b:BUFF || c:BUFF ||d:BUFF || SINK )

             /{send/a.read, send/a.read1, a.write/b.read,
a.write1/b.read1, b.write/c.read, b.write1/c.read1, c.write/d.read,
c.write1/d.read1, receive/d.write, receive1/d.write1 }
             \{a.write, a.write1, b.write, b.write1, c.write, c.write1}.
```

Figure 30: FSP code for the revised model

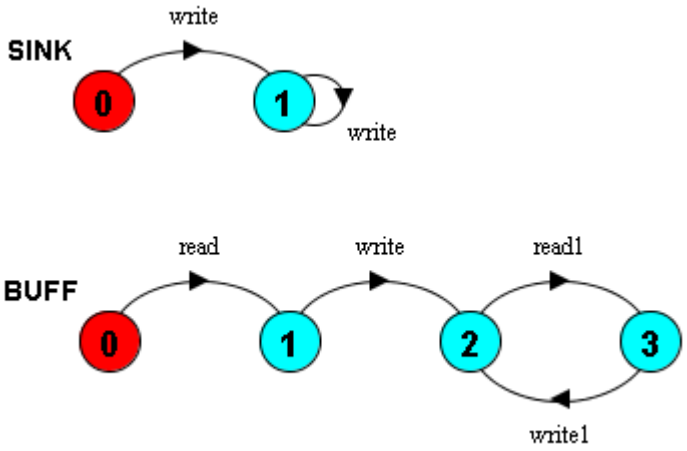


Figure 31: Diagrams of revised buffer and source processes

	Without Minimisation		Minimised	
No. Buffers	No. States	No. Transitions	No. States	No. Transitions
0	2	2	2	2
1	4	4	4	4
2	8	10	6	8
3	16	24	8	12
4	32	56	10	16

Figure 32: State and transition counts for revised LTSA model

8.4 Summary

The table in Figure 32 shows the numbers of states and transitions for each version of the model both for the un-minimised and the minimised versions of each of the compound processes. A comparison of the numbers in this table with those in Figure 24 and Figure 25 reveals a clear relationship which supports the assertion that asynchronous communication via buffered channels in RDT (as interpreted by SPIN after automated translation into Promela) is equivalent to asynchronous communication implemented using buffer processes connected by zero length channels, after taking into account the effect of the events concerned with the passing of values along a multi-place buffer built from several processes.

Chapter 9 Experiments and examples[1]

The following examples illustrate RDT in action. The first selection of examples show the basic features of the language and system. These are followed by three larger examples. Two of these larger examples show models of test systems built as part of the RICES [Henderson, Walters et al. 2002] project. The third is based on the mobile phones example in Milner's tutorial paper on the pi-calculus [Milner 1993].

The first of the larger examples is a model of a system built to represent the behaviour of a collection of platforms(ships). These platforms are able to move around, see other platforms using "sensors" and communicate. This example shows how a complex arrangement of connections can develop from a minimally connected initial state and how these connections can be reconfigured at runtime.

The second is an elementary "internet" banking system. This system illustrates the "inbox" architecture being proposed within the RICES project. This model illustrates taking a realistically sized model in RDT and converting it into Promela for analysis with SPIN. With this model, we were able to identify a feature of the "inbox" architecture which renders it vulnerable to deadlock.

9.1 The simplest model

The simplest process possible in RDT is one which carries out a single event and ends. Since there are three types of process there are three types of single event process. The first is shown as "firstproc" in Figure 33. This process comes into being, knowing of a channel name "p". It writes the value of "p" onto the channel it knows as "p" and moves to a state called, "two". A "write" event is shown in an RDT diagram as a clear box (with a black outline). It proceeds no further. Notice

[1] In the implementation of the RDT tools, the "Send" and "Receive" events have been renamed to "Write" and "Read" respectively.

that the process needs a value to write into the channel. In this example, the process has to write the value “p” since “p” is the only value it knows.

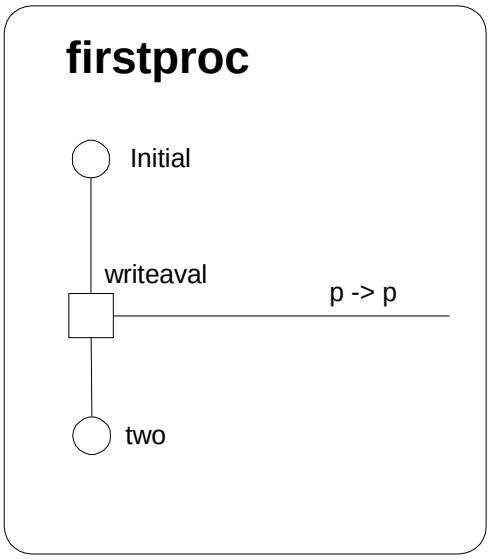


Figure 33: A simple process

The second of these minimal processes is illustrated in Figure 34. This process is called “sink” and differs from the one in Figure 33 in two ways. First, after coming into existence, it reads a value from the channel it knows as “public” which it is then able to refer to as “x”. A “read” type event is shown in that diagram as a black box. Second, this process exhibits a cyclic behaviour: after the “read” event, unlike the first process, it does not move to a new state. It returns to the state “initial”. The only rules about RDT process states is that all processes start in a state known as “initial”. Beyond that, there are no further rules, so a process may exhibit cyclic behaviour by returning to a (previously visited) state – so, where this is desired, an event’s before and after states may be the same. However, although RDT does not prevent the definition of such events, there should be a path (via other events as appropriate) from “initial” to the before state of each event since otherwise the event can never take place.

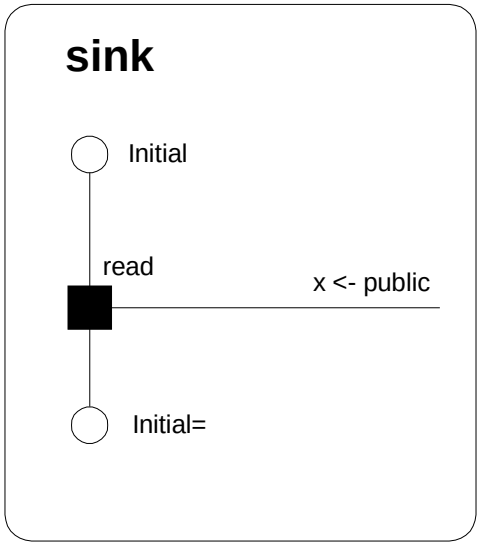


Figure 34: A simple "sink" process

With these two processes, we are now able to create an executable model by creating one instance each of “firstproc” and “sink” and arranging for the channel known by “firstproc” as “p” to be the same as the channel known as “public” by the process “sink”. This is the second part of the RDT system generation process, the creation of a model. Each of the instances in the model is shown as a box labelled with the name of the process followed by a colon and its type. Each of the channel names is shown at the right hand side of the box as a blob attached to the box by a line and labelled with its name. Figure 35 shows our minimal model with the required connection in place.

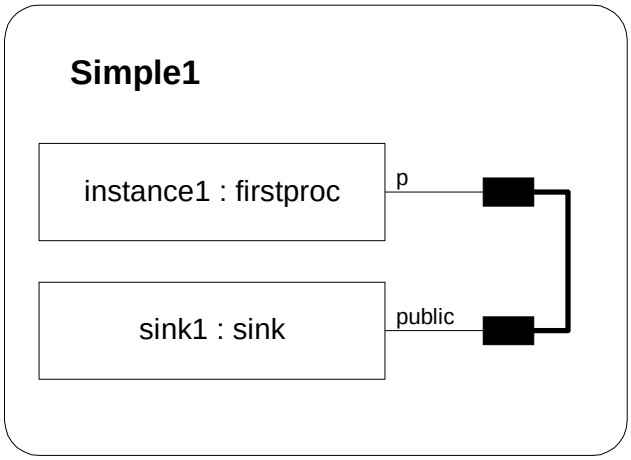


Figure 35: A simple model showing one instance each of firstproc and sink with a single connection between them

Once the model is created using the RDT model creation tool and the file saved (Figure 39 shows the XML generated for this model), the completed model can be loaded into the RDX execution tool. Figure 36 shows this model immediately after being loaded into the execution tool. Notice that the tool shows one window each for the instance of firstproc called, “instance1” and for the instance of the sink process called, “sink1”. There is also a window for the one channel required to make the connection between the processes. In this initial state, “instance1” shows the “writeaval” event in its list of available events. The window to the right of the event list shows the effect of the event: “p -> p” meaning that the event causes the value “p” to be written into the channel known by this process as “p”. “Double-clicking” on the name of the event, causes RDX to execute the event and the appearance of the application window to change to that shown in Figure 37.

Now the list of available events in the window of “instance1” is now empty, the value associated with the name “p” in that process (channel0) now appears in the window of Channel0, and the window of the process, “sink1” now shows the read event, “read” (with its explanation of x <- public) as being available. Double-clicking on this second event causes the “read” event to occur and application window to change to the situation shown in Figure 38.

Now channel0 is empty again, neither process is able to perform any events and the model can proceed no further.

Notice that the list of channel names known to the process “sink1” now shows the value “channel0” associated with the name, “x”. Also, the reason why process “instance1” is unable to perform any events is that it is now in state “two” and it has no events which have that state for a “before” state, whilst the process “sink1” cannot perform any actions for a different reason. It has returned to the state “initial”, and would be able to perform the “read” event again. However, this event reads and removes a value from the channel known locally by the process as “public” (channel0) and that channel is now empty.

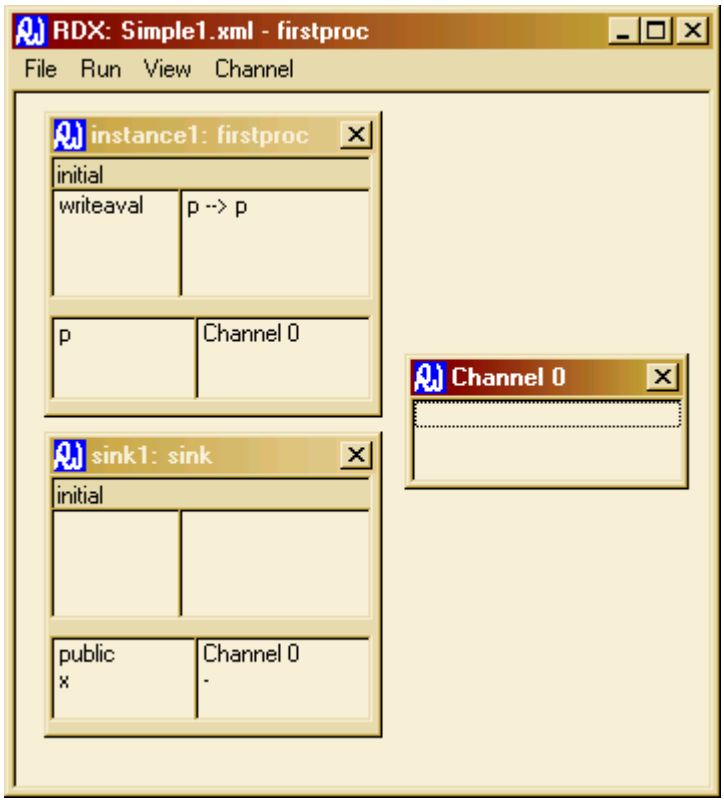


Figure 36: The Simple model immediately after loading into the RDX execution tool

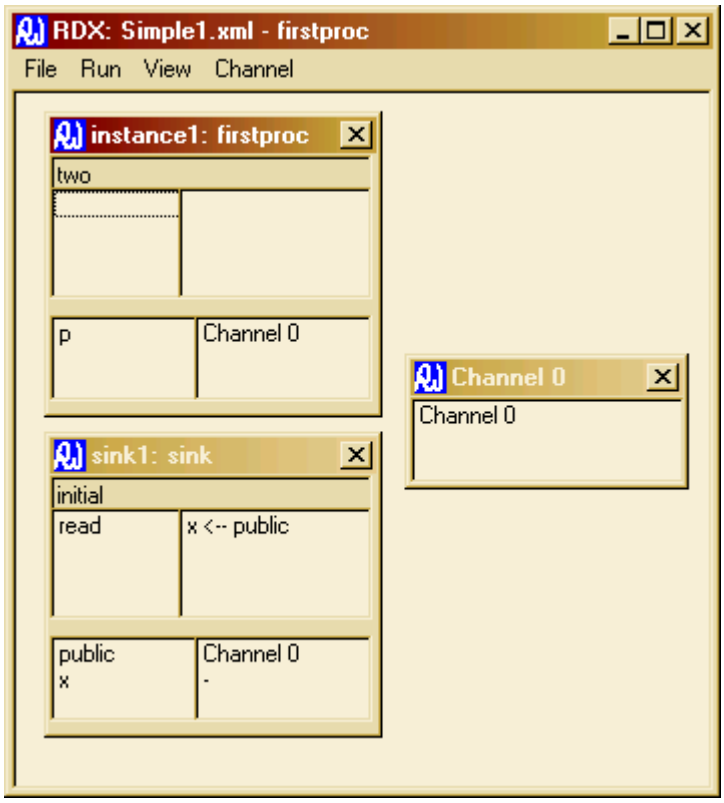


Figure 37: The simple model after the instance of "firstproc" carried out its "write" event showing the value in the channel

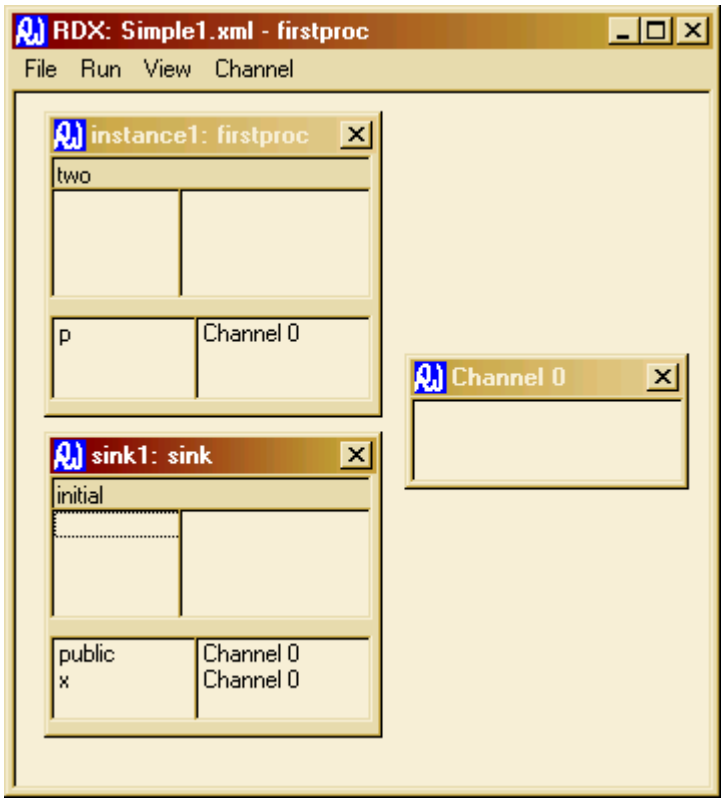


Figure 38: The simple model after the sink process has carried out its read event

```
<Model>
<Process Name="firstproc">
<Event Name="writeaval" Type="Write" Before="initial" After="two"
Channel="p" Value="p"/>
</Process>

<Process Name="sink">
<Event Name="read" Type="Read" Before="initial" After="initial="
Channel="public" Value="x"/>
</Process>

<Instance Name="Model1">
<ProcInstance Name="instance1" Type="firstproc"/>
<ProcInstance Name="sink1" Type="sink"/>

<Connection>
<End ProcInstance="instance1" Channel="p"/>
<End ProcInstance="sink1" Channel="public"/>

</Connection>
</Instance>

</Model>
```

Figure 39: The XML generated by the RDT tool for the simple model.

9.2 A second example model

This example takes the simplest model of the previous example and adds a further process. This process is like the “firstproc” process in that is writes a value to a channel. Unlike the “firstproc” process, it performs an event of type create. The process is shown in Figure 40. The event is distinguished from a normal “write” type event by the addition the diagonal cross in its box. As with the other types of event, its before and after states are shown as circles labelled with the state name and the effect of the event is summarised on the horizontal line leading from the event itself. The effect of the this event, when it occurs is that a new channel (which will be referred to locally by the process as “r”) is created and its value is written onto the channel known to this process as “q”. For this example, the model of the previous example is extended by the addition of two instances of the new process, “secondproc”. Each of these has a channel which it knows as q and these are both connected to the “public” channel of the “sink” process. This model is created using the same techniques as the first example. The completed model is shown in Figure 41. Figure 42 shows this second model loaded into RDX ready to start execution.

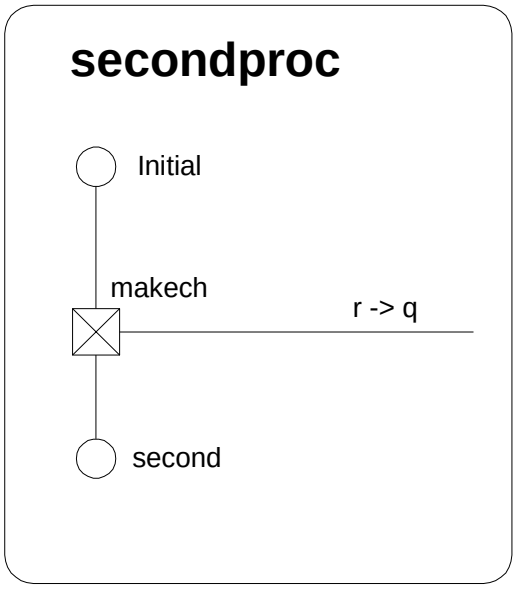


Figure 40: A simple process which creates a channel

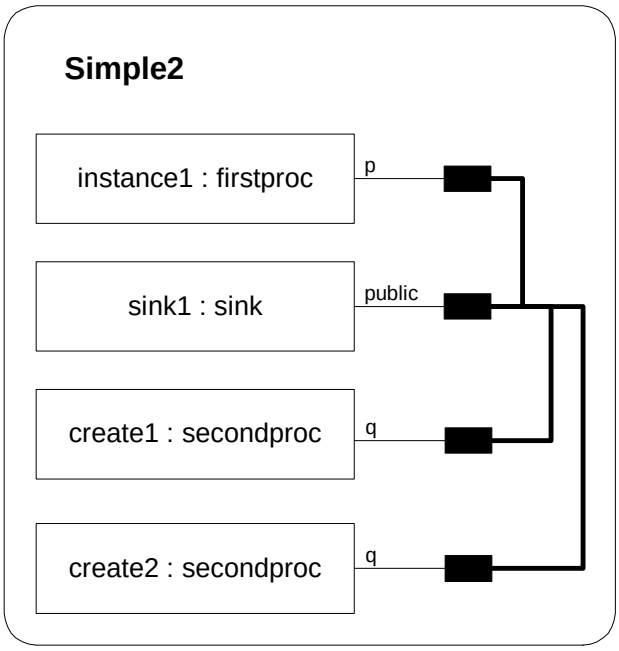


Figure 41: The example model with two instances of "secondproc" added and connection to "sink,public"

As with the previous model, each process has a window showing its state. This time there are four. This model has more connections than the first, but the three connections in this model are from the “public” port of the “sink1” process, so for an execution of the model, in common with the first model, there is just one channel: channel0. The name of this channel is associated with the channel known as “public” in the “sink1” process, the channel known as “p” in the “instance1” process. It is also associated with the channel known as “q” in each of the instances of the processes of type “secondproc”.

In its initial state, this model has three processes able to carry out an event: “instance1” (as before) and the two new instances of “secondproc” called “create1” and “create2”.

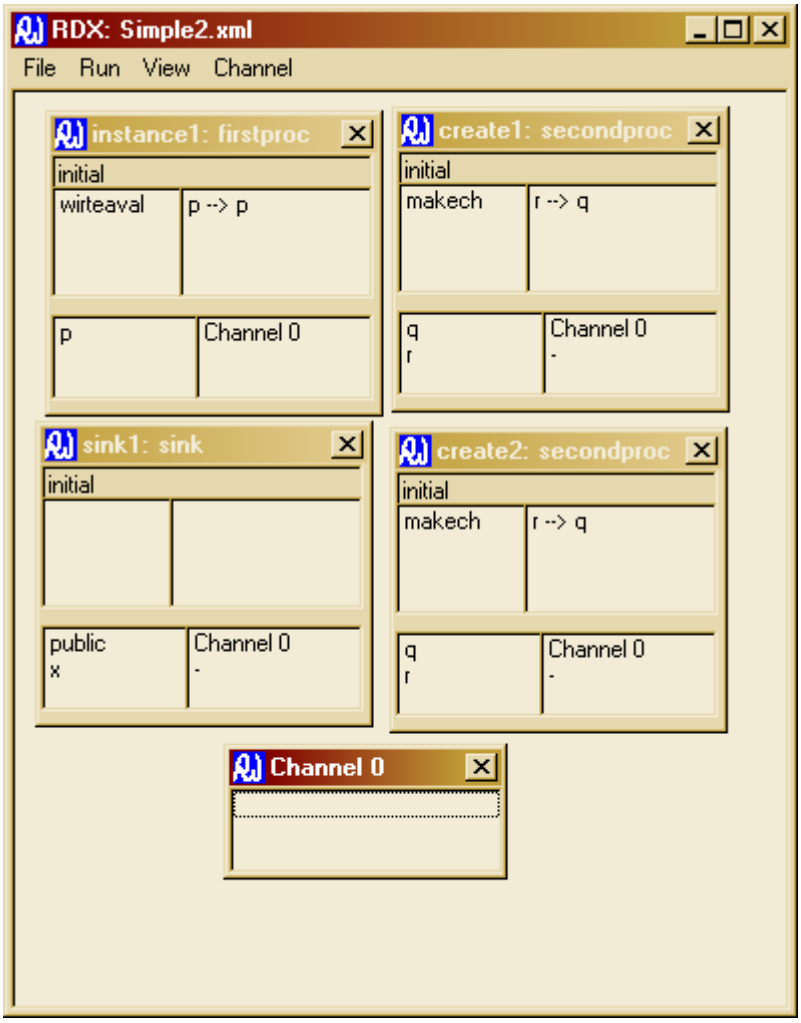


Figure 42: The second model loaded into RDX

For this example, the length of the channels in the model has been set to 0 by using the command available from the RDX menu. Figure 43 shows the state of the model in the execution tool after the process named "create1" has carried out its event, "makech". With the channel length set to zero, the communication within the model becomes synchronous. However, at the point in execution shown in Figure 43, only one half of the communication has occurred. "Channel0" is holding a value and no process has "read" a value. This is an interim state which cannot be allowed to persist. RDX forces the modeller executing the model to bring the model back to an acceptable state by disabling all events except those "read" events which will remove the freshly written value from "channel0". It also draws attention to the process which caused the interim state to come about by colouring the background in the "create1" process. In executing to this interim state, the

"create1" process has already caused a new channel to be brought into existence. This new channel has a window like that of "channel0" and, apart from its name is indistinguishable from "channel0".

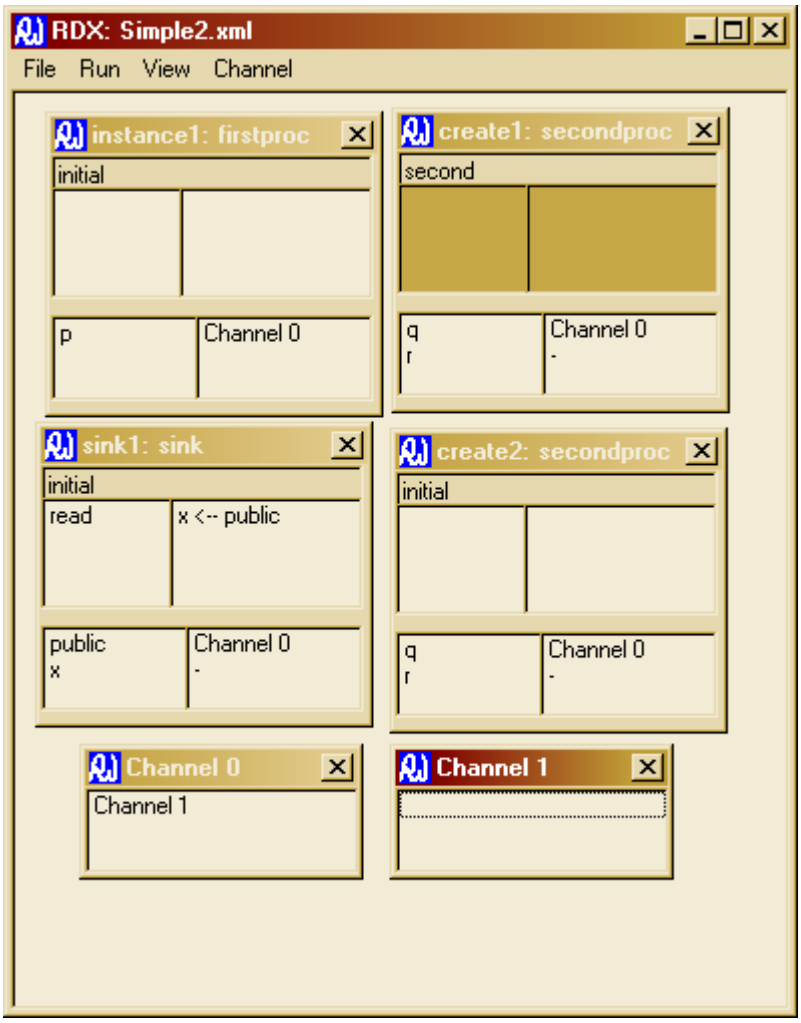


Figure 43: The second model after a create event has occurred, but before the corresponding "read" event.

Once the modeller actions the "read" event in the "sink1" process, the model is returned to an acceptable state and the evaluation of the lists of available events for the processes in the model returns to normal as does the background of the "create1" process as shown in Figure 44. Notice that the new channel, "channel1"

is now associated with the name, “r” in process “create1” and with the name “x” in process “sink1”.

Execution may now proceed until all three of the possible events has occurred and the model stops with no available events.

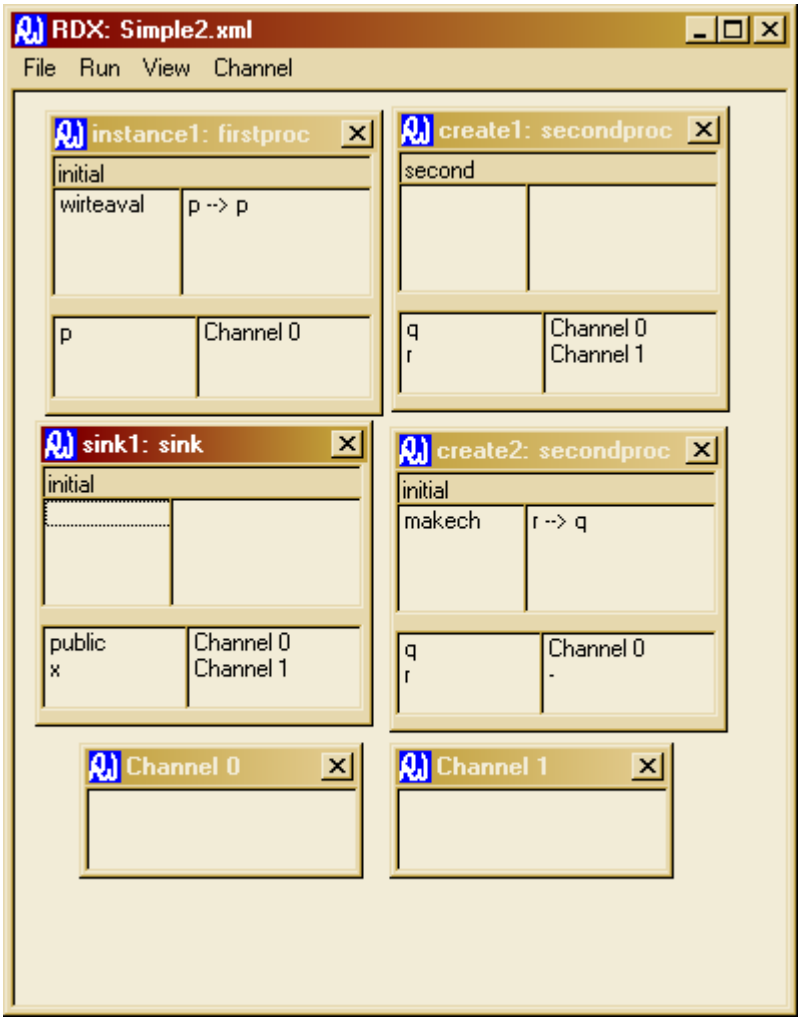


Figure 44: The second model after the "read" type event has occurred

9.3 Defence

9.3.1 Outline of the "real" system

The "MQDefence" system is one of the experimental systems which have been constructed as part of the RICES [Henderson, Walters et al. 2002] project. The scenario is intended to be representative of the type of situation encountered by the military. Here, amongst other problems, they are faced with trying to keep track of the other players in the battle as well as where they are themselves. The complexity of the real situation is enormous. For the RICES project, a simplified system has been constructed with which to explore the problems encountered by the various components in the system as a result of their imperfect data.

The test system has a notion of a small world divided into squares on a grid. Each "platform" (a ship, say) in the system occupies a single square. There should never be more than one platform in any square. Each platform has a number of possible actions.

The simplest action is for the platform to move to another square. The system permits platforms to move to any square adjacent to its present square.

Platforms may also create and use "sensors". These behave like a radar system. A platform with a sensor may "read" that sensor. By doing so, the platform receives the name and position of any platforms within the range of the sensor. The platform itself is included in this data, since the reading platform is at the centre of the area visible to any of its sensors.

Where a platform knows of the existence of another, it may establish a "channel" to the other platform. Once a channel is established between two platforms, either of them may initiate an enquiry on the channel. In the system's present form, a platform responding to an enquiry does so with a complete list of all the platforms in its view of the "world". This channel mechanism permits platforms to discover the existence of others in two ways:

- A platform which establishes a channel to another platform, may find that the data it receives on reading that channel includes the names and locations of platforms previously unknown to it.

- After a channel is established between two platforms, they inevitably know of each other's existence. To create a channel, the creating platform must know the name of the platform to be at the other end, but the converse is not necessarily true. Hence a side effect of a platform establishing a channel my be to disclose its existence to the other platform.

The system has a web-based interface which permits users to take control of a platform, see its "view of the world" and pass instructions to it. Instructions include telling the platform to move as well as to create sensors and channels and to read sensors and channels. Via the interface, the user is also able to decide whether to adopt data received from a sensor or channel read.

In a real battle, the participants have a physical presence which might be considered to represent the true situation. It is fragments of this "real world" which the sensors (radar, etc.) of real combatants discover. In our simulation, the participants do not have a physical existence so, in its place, we have an addition process which provides this service. Platforms are required to report their movements to this process so that this process always has complete and accurate data about the location of the various platforms regardless of what the platforms themselves might believe. When a platform reads a sensor, the sensor consults with this process to discover what it "sees" and reports. This additional process is referred to as the, "world view". As part of the initialisation of a platform, it is required to notify its existence and position to the "world view".

9.3.2 MQDefence Models in RDT without Communications

Even with the limited features of the MQDefence simulation, the system is already too complex to build a model which is complete in every detail. Instead, we

have built two models which are representative of the behaviour of MQDefence. Additionally, RDT models are concerned almost exclusively with behaviour so its processes have a limited ability to hold and manipulate data although processes can hold data in the form of their named state and the names of the channels which are known to them.

The state of an RDT process is not wholly encapsulated in its named state. Some aspects of its behaviour are determined by the values it holds for the various names it knows. This can affect the behaviour of a process since, where a name is not associated with a value in the initial set-up of the model has not yet acquired a value the process will be unable to execute any event which uses that name for its channel. The other is less direct - the other processes in the model with which a process communicates depend on the way in which the associations between channels and the names used within processes. The behaviour of those other processes which also have access to these channels can determine whether or not this process is able to perform various actions by influencing the availability of values for reading from channels or the ability of those channels to accept an additional value to be written.

In the first MQDefence model, each platform initially “registers” with the “world view” process and then is able to make enquiries of the “world view” about the presence of other platforms as a platform in the system would read a sensor. The process receives a response indicating which platforms are visible. Which responses are possible depends on how many platforms are registered with the “world view” process. Where more than one response is possible, the “world view” decides which to send. Matters relating directly to the position of the platforms are not addressed directly by this model. As with the real system, which of a number of possible responses is given is decided by the “world view”. In the real system, this decision is determined by the physical location of the various platforms and the range of the sensor. For the model, the “world view” is released from this constraint and is permitted to respond to any enquiry with any of the possible responses.

Although some data is encoded into the associations between channels and the names used within processes, a process is not able to examine this information. So, for a process receiving a communication to associate a meaning with that communication, it must do so by virtue of its knowledge of which of the names it uses for the channel on which the message arrives. This is the reason for the elaborate registration procedure at the start of each of the platform processes. This sequence of communications enables the "world view" and the platform process to establish enough exclusive channels of communication between them for the platform to make an enquiry of the "world view" and identify which of several possible responses it receives according to the channel on which the response arrives. The platform process is shown in Figure 45.

At the start of execution, the platform process has no choice of action. It has to perform the registration procedure. This consists of creating a new channel which it names "me" to the "world view" process. This channel then forms a private line of communication between this platform and the "world view". The platform then listens on this new channel for three values from the "world view", which it names "one", "two" and "onetwo". It then sends a further freshly created channel which it calls "s" to the "world view" in its first enquiry. The "world view" responds in one of four ways:

1. A message on the channel known as s indicating that the sensor can see no other platforms.

2. A message on the channel known to this process as "one" indicating that the sensor can see one of the other platforms.

3. A message on the channel known as "two" indicating that the sensor is able to just the other platform.

4. Two messages in sequence on the channel known as “onetwo” indicating that the sensor has seen both of the other platforms.

In this model, the values supplied by the “world view” process are unused.

The “world view” process is more complex. It is performs the registration sequence of events with the platforms in any order. Each time a platform registers, channels are created, more of the names of the “world view” become associated with channels and the “world view” process acquires an additional set of actions corresponding to handing an enquiry from the new platform and the enlargement in the number of possible responses to enquiries from any platforms already registered. Whenever the “world view” receives an enquiry, its response is to reply the enquiring platform on a channel which indicates to that platform which of the other platforms in the model are visible within its sensor range.

When a platform process performs a “sensor read” (a SRread event), the “world view” process replies on the appropriate channel to indicate to the enquiring platform that none, one or both of the other platforms is within range of the sensor. When just one platform has registered, the only possible response is “none”. When two platforms are registered, the response could still be “none”, or the other platform, when all three platforms are registered, the response can still be “none”, either of the other platforms, or both. (In the real system, the response would be governed by the positions of the platforms and the range of the sensor. In the model, any of the possible responses is permitted.)

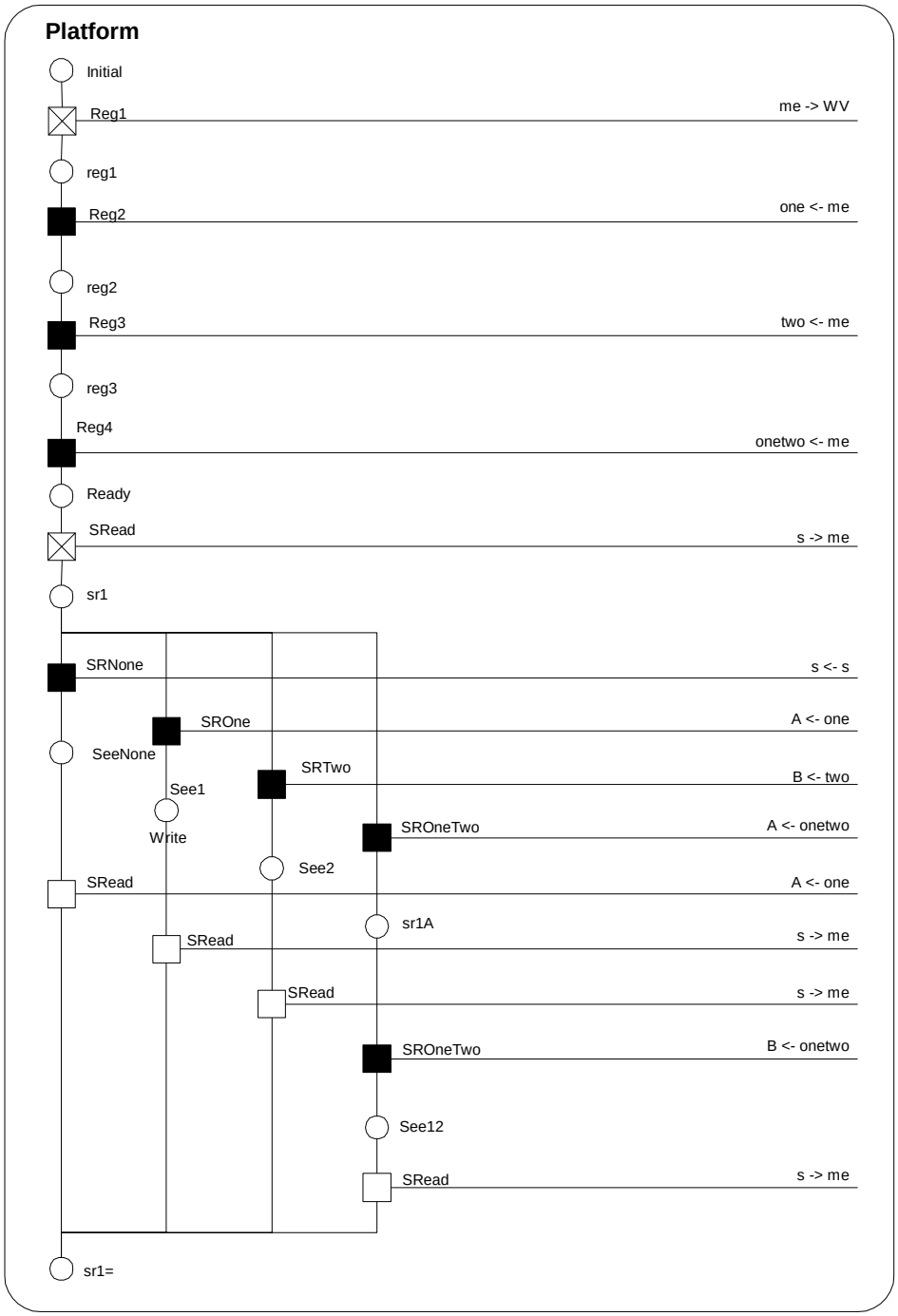


Figure 45: The first version of the platform process

Figure 46 shows the view of an initial "model" of MQDefence with the "world view" process at the top followed by three "platform" processes. This diagram shows the initial connections between the process instances in the model. There

are just three connections associating the “WV” channel name of each of the process instances with the “P” channel name on the “world view” process instance. When the model is loaded into RDX for execution, these three connections resolve into a single channel (channel0) which is know to all four processes. However, as the platforms register and begin to perform enquiries, more channels are created and the situation becomes much more complex. Figure 47 shows the model during execution. Here all three processes are registered, and two of them have made an enquiry. The model which started with just one channel now has 15 associated with the various ports of the “world view” process instance and the platform instances. The exact details of how these channels are shared depends on the order of execution. For example, the “world view” process always refers to the first platform to register as “P1”, but the platform process instance to which this refers is determined by which of the platform instances is first to register.

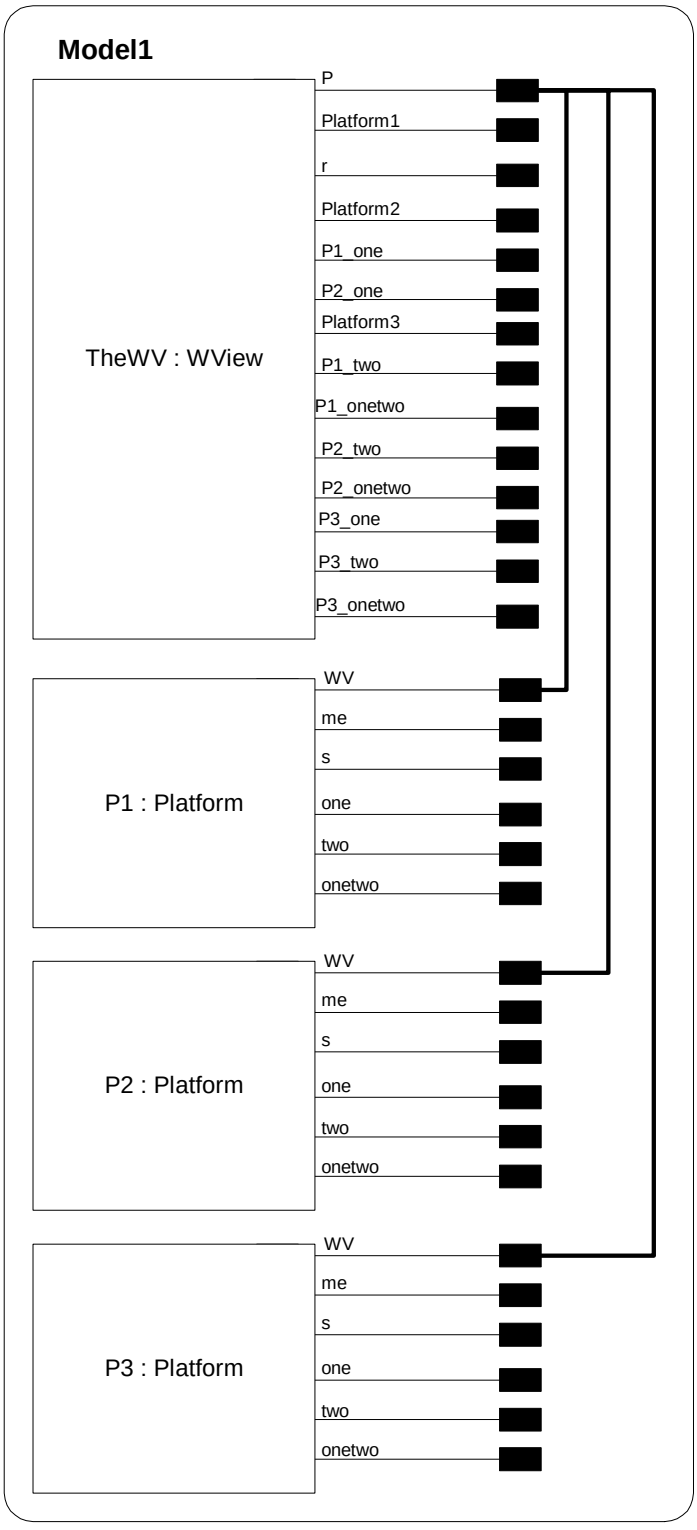


Figure 46: Model view of the first version of MQDefence

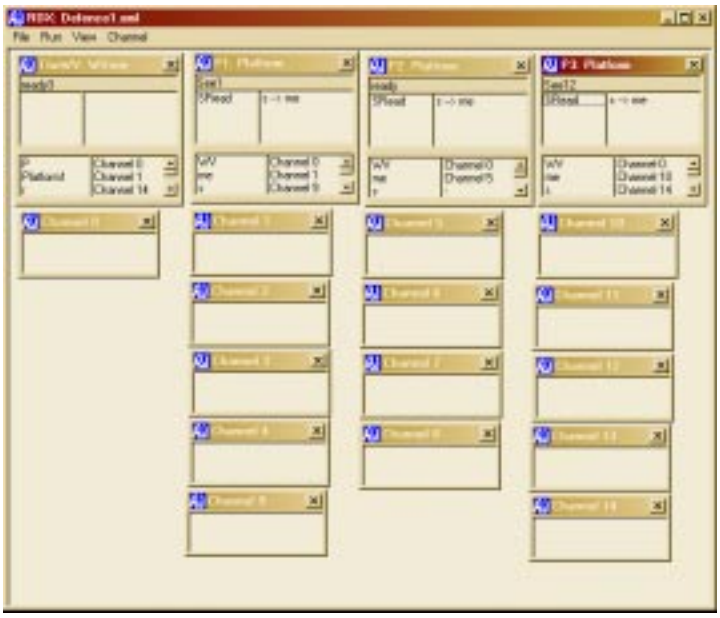


Figure 47: The first MQDefence model during execution

9.3.3 Adding communications to MQDefence Models in RDT

In the second MQDefence model, the behaviour of the "world view" process is unchanged.

The elements of the behaviour of the MQDefence platform modelled in the first model and described above are also unchanged but the platform process is enlarged to include a capability to communicate with others and this is where the values supplied by the "world view" in response to sensor reads and the channel name which is sent to the "world view" in the first sensor read are used. When the "world view" responds to a sensor enquiry, the values which it sends are the names of the second channel supplied to it by each of the registered platforms. These values are stored by the "viewing" platform as "A" and "B" and uses these channels to communication with the other two platforms that may exist in the model. Where a platform "knows" of just "A", say it is possible that the result of communication with that platform may be the discovery of platform "B". This model replicates the behaviour of the actual system where a platform is able to

discover another platform using its sensors and then discover a further platform by interrogating the platform already located.

Including this additional behaviour adds considerably to the complexity of the platform process leading to a process with over 50 events and a similar number of named states. Building and manipulating such a model in the RDT model creation tool confirms that the language and the tool is feasible even though the diagram is too large to fit readily onto a single A4 page. Once constructed, this model was executed in the RDX execution tool and subsequently converted to Promela and subjected to analysis by SPIN which was able to identify a trace to deadlock in the revised model which was not present in the version without communications.

9.4 Banking

The second demonstration system built for the RICES project is of an internet banking system which has acquired the name "WebATM". This system has been the subject of considerable discussion which has led to a proposed architecture for the type of large, distributed, loosely coupled and potentially unstructured system which is the focus of the project. The architecture is commonly referred to as, the "inbox architecture" [Henderson, Walters et al. 2001] and, along with the system itself is outlined below.

9.4.1 "WebATM"

As with the MQDefence system, WebATM has an interface which enables users to operate the system remotely using a web browser and again, for the purpose of modelling the system, most of this aspect is disregarded.

The remainder of the system comprises of a number of "banks" which offer a strictly limited accounting service: they hold balances on accounts and are able to execute transfers between accounts. It is possible to transfer monies between banks using the services of a third party known as the "clearing". The complete system also offers other facilities including the maintenance of "mirror" accounts

and the provision of “agent” services which are not modelled here. (Mirror accounts are a service offered by our banks by which users can maintain their own version of their accounts and periodically reconcile the two versions. Agent services include the maintenance of a specified balance in an account and notification to the user when an awaited entry is applied to an account.)

Aside from the mirror account and agent special features the system operates just as would be expected from any online banking system.

9.4.2 WebATM model

The WebATM modes are built from three types of process, Clients, Banks and Clearing(s).

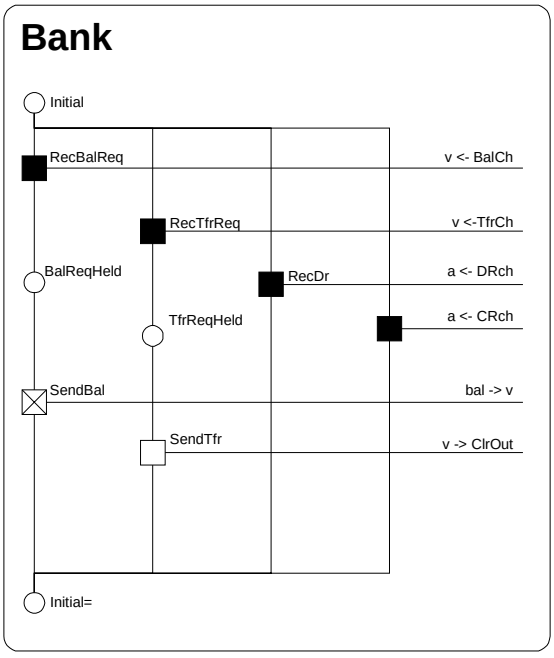


Figure 48: The Bank process

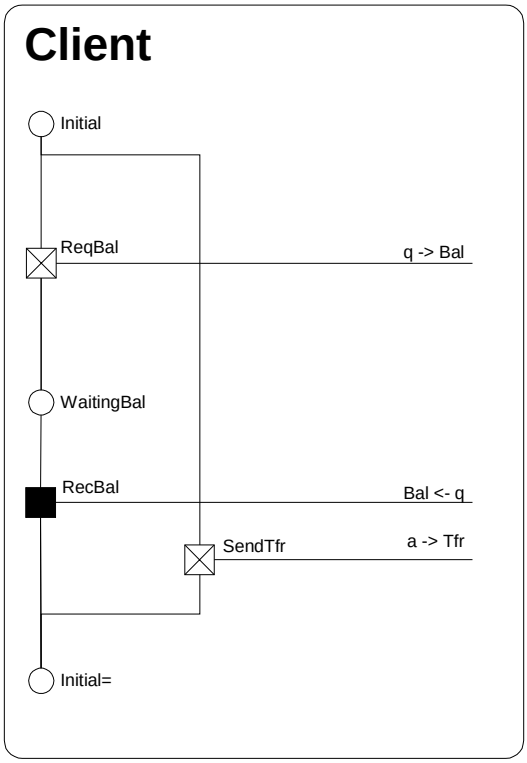


Figure 49: The client process

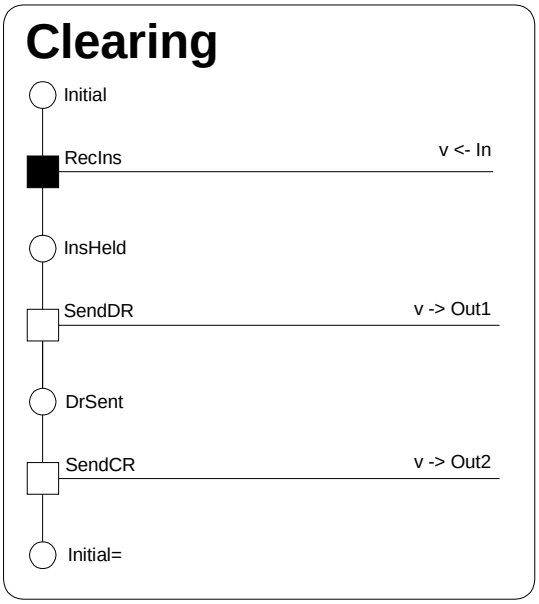


Figure 50: The clearing process

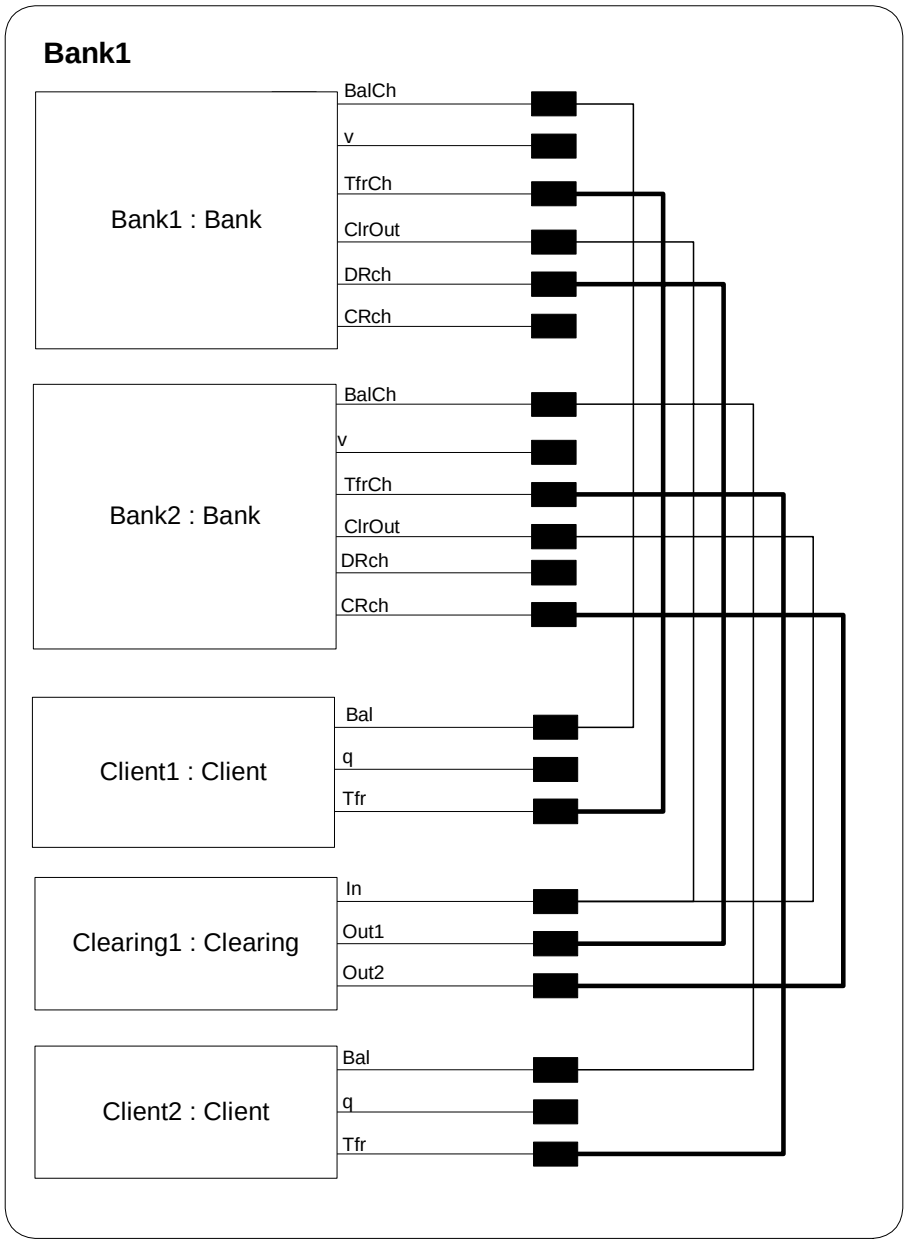


Figure 51: The WebATM model

For the purpose the modelling the system, we have removed most of the complexity of the system by building a model of a system in which each client deals with one bank only, and each bank has just one account (and hence just one client). The behaviour of the system is further simplified by not giving the bank and client any choice about what kind of transfer to make: a client's request is for a transfer is passed on to the clearing process which processes all requests in the same man-

ner. It sends a debit instruction to one bank and a credit instruction to the other. Even this minimal functionality is sufficient to uncover unexpected behaviour.

A client process starting in its initial state has a choice of two actions. It can ask its bank for a balance. It then waits for a reply and returns to its initial state once the reply arrives. The other action the client can do is to request a transfer after which it returns to its initial state immediately.

A bank process in its initial state has a choice of four events. The first is to receive a balance request from its client to which it replies by sending a response before returning to the initial state. The second is to receive a transfer request from its client which it passes on to the clearing application before returning to its initial state. The remaining two options for the bank are to receive one of two instructions from the clearing application – either to place a debit onto an account or to place a credit onto an account.

The greatly simplified clearing process in its initial state is able to receive a instruction which it discharges by sending a debit instruction to one bank and a credit instruction to the other.

The processes of the model are shown in Figure 48 to Figure 50 and the completed model is shown in Figure 51.

9.4.3 The “inbox” architecture

The WebATM system follows the pattern of the “inbox” architecture for a distributed system. The pattern dictates that, whatever else they may do, all processes in the system have an “inbox” to which communications for the process may be addressed, and that a process must read and “deal with” items which arrive in its inbox. How these messages are dealt with is not mandated, so potentially a process could satisfy the requirements of the architecture by regularly collecting the con-

tents of its “inbox” and discarding them, though this is unlikely to be a useful behaviour.

The architecture is being proposed as a pattern for the generation of processes which, when placed together in a distributed environment will be able to, co-exist without damaging each other (though they may be unable to achieve their goals).

For the WebATM, this means that all processes have a queue from which they undertake to read and handle messages as they arrive and this is the way in which the full system is implemented.

9.4.4 Deadlock in WebATM and the “inbox” architecture

Since so much of the functionality of the WebATM system is missing from the model, running the model in RDX is not in itself especially interesting beyond confirming that the system is indeed able to operate as intended with the clients able to obtain responses from their bank to balance enquiries and transfer requests from clients being forwarded to the clearing process which actions them by sending out debits and credits to the banks. However, working with this model, the RDTtoSPIN translation tool and SPIN itself, it is straightforward to identify that this model is vulnerable to deadlock.

The deadlock found by SPIN is a consequence of the way that the banks and the clearing applications rely on each other to achieve their aims – a bank receiving a transfer instruction from its client gets this processed by passing an instruction to the clearing application, whilst the clearing application (upon receiving a transfer request) issues instructions to the banks:

Consider a system in which the channels are of zero length. Communications between processes will be synchronous. Now consider the situation where a client of one bank decides to issue two transfer instructions in rapid succession. On receiving the first instruction, the bank will pass it on to the clearing system and be

immediately ready to accept the next transfer instruction, suppose the next event to occur is the bank accepting the second transfer instruction.

Now;

- The clearing process is in a state where it has to issue instructions to the banks to make debit and credit entries before if will accept another transfer instruction from a bank, but

- One of the banks will not accept a debit (or credit) instruction because it has already accepted a new transfer instruction from its client and won't deal with anything else until it has passed that instruction on.

- Deadlock.

Looking at this situation, it is easy to suppose that, this scenario cannot occur in WebATM as the communications between the processes in the system are asynchronous. Certainly, the deadlock outlined above would be broken if the communications style were to be changed to one where the channels between the processes had a single place buffer. However, further analysis using SPIN and versions of the model in which the channels are of various lengths reveals that deadlock still occurs. All that is required is for a client process to generate sufficient transfer requests to fill up the buffers in the channels between the bank and the clearing processes. Given a path to deadlock, increasing the length of the buffers in the channels in the system is enough to break that particular deadlock, but there will always be another.

On reconsidering the problem, it is possible to identify that systems built using the "inbox" architecture will be vulnerable to deadlocks of this type since all of the required factors for deadlock are present. Unfortunately, the environment to which the "inbox" architecture is targeted is one where controlling these factors is unlikely to be practical. For example, in the type of loosely coupled, asynchro-

nous system we envisage, it is unlikely to be possible to eliminate cyclic dependencies between processes because, although we may know which processes our own process uses and relies upon directly, we probably cannot know which other processes we need and use indirectly. Worse still, even if we were able to establish freedom from cyclic dependencies when we add our component into a system, we cannot know in advance what dependencies may emerge as the system evolves and other processes are added and removed.

However, there is hope and the absence of deadlock in the practical demonstration system suggests that although the problem discovered in the model with SPIN is one of which we should be aware, it need not be a significant danger in a real system. In particular, it appears that the ability of MSMQ (the message passing middleware used in WebATM) to manage the data in queues and extend the available space as required gives a real system the ability to escape a deadlock by extending the length of channels (queues) when need arises.

9.5 “Mobile Telephones”

This model reproduces the car-phone model in Robin Milner’s tutorial paper on the pi-calculus [Milner 1993]. The system being modelled is outlined in Figure 52. The centre is in permanent contact with two base stations. A car with a mobile phone moves around (but is assumed to always be within range of at least one of the base stations). As the car moves, from time to time it needs to change base stations in order to remain in contact with the centre and a hand-over sequence is initiated in which the car relinquishes contact with its present base station and resumes communication with the centre via the other.

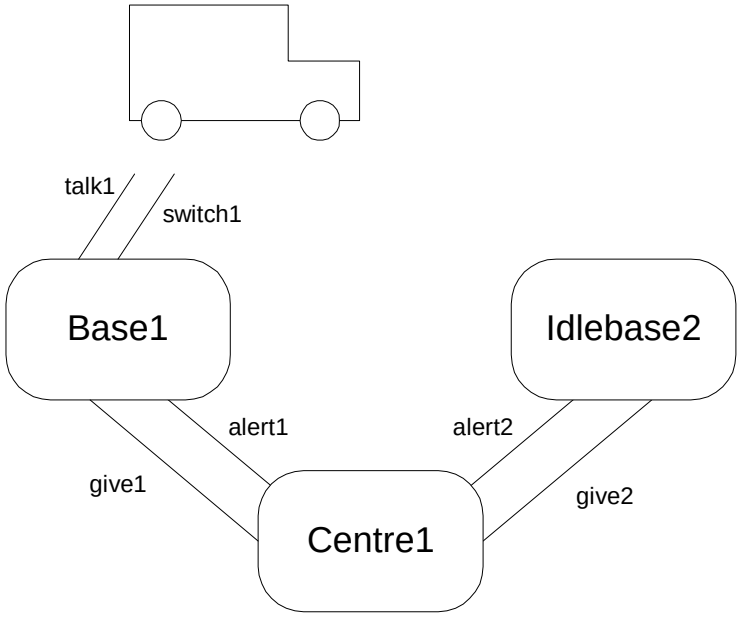


Figure 52: "flowgraph" of the car phone application

The system may be defined in pi-calculus as follows:

$$\begin{aligned}SYSTEM = (\nu talk_i, switch_i, give_i, alert_i : i = 1,2)\\ (CAR(talk_1, switch_1)\\ | BASE(talk_1, switch_1, give_1, alert_1)\\ | IDLEBASE(talk_2, switch_2, give_2, alert_2)\\ | CENTRE_1)\end{aligned}$$

Where:

$$CAR(talk, switch) = talk.CAR(talk, switch)$$
$$+switch(talk's').CAR(talk', switch')$$

$$BASE(t, s, g, a) = t.BASE(t, s, g, a)$$
$$+g(t's').\bar{s}t's'.IDLEBASE(t, s, g, a)$$

$$IDLEBASE(t, g, s, a) = a.BASE(t, s, g, a)$$

and

$$\mathrm{CENTRE}_1 = \overline{\mathrm{give}_1}\,\mathrm{talk}_2\mathrm{switch}_2.\mathrm{alert}_2.\mathrm{CENTRE}_2$$
$$\mathrm{CENTRE}_2 = \overline{\mathrm{give}_2}\,\mathrm{talk}_1\mathrm{switch}_1.\mathrm{alert}_1.\mathrm{CENTRE}_1$$

This description uses an extension to the syntax of the pi-calculus described in Chapter 6 in which a communication along a channel is permitted to be a tuple containing an arbitrary number of values (though the number of values in the tuple sent must match the number in the receiving tuple). Using this syntax greatly assists in the construction of this succinct description of the system. RDT does not have this feature. Where multiple values are to be sent this has to be performed by a sequence of communications (with appropriate care to ensure that no unexpected interleaving can occur). This is easily achieved, if not always elegant.

Informally, the Car knows two channels, talk and switch. It may either communicate with the base on the channel it knows as talk or receive a replacement pair of channels on the channel it knows as switch. After receiving a new pair of channels, the Car may now communicate (via a different Base) on the channel which it now knows as talk, or receive a further pair of channels on which to operate on the (freshly arrived) channel it knows as switch.

Each base process knows four channels which it refers to as t,s,g,a. Their behaviour is described above as two processes. Considering first the base which is in contact with the car, this base is described by the process Base. It may either communicate with the Car on the channel it knows as t or it may receive a new pair of channels from the Centre along the channel it knows as g. The Base then passes these new channels on to the Car and behaves as an Idlebase. An Idlebase waits for a communication along the channel it knows as a and then behaves as Base. (In fact the two bases of the system behave in the same manner, differing only in their initial state.)

The Centre alternately sends a pair of channels to the active Base followed by an activation message to the inactive Base, causing the active Base to pass the channels on to the Car and become inactive and the inactive Base to become active.

Reproducing this model in RDT highlights some differences between RDT and the pi-calculus. Constructing the Car process in RDT is straightforward. This is shown in Figure 53. Building the Base processes illustrates one of the differences between the systems. In the pi-calculus description, the two elements of the base behaviour are described in terms of each other permitting the creation in the final system of two instances of the same behaviour (subject to parameterisation) which start in two different states. This feature is not available in RDT so the modeller has to find an alternative representation. The two obvious possibilities are to describe a single process and then use some initialisation events to cause two instances to move to the required (differing) states for the start of the pi-calculus model or to described the processes separately which is the solution chosen here. The two bases are described by the processes BaseA for a base which is initially active and BaseI for a base which is initially inactive. These behaviours are described in Figure 54 and Figure 55. It is interesting to observe that, despite the apparent difference between the diagrams that these two behaviours are closely related.

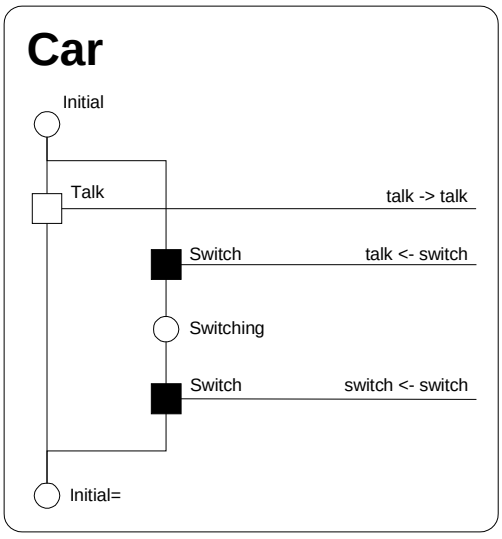


Figure 53: The Car process

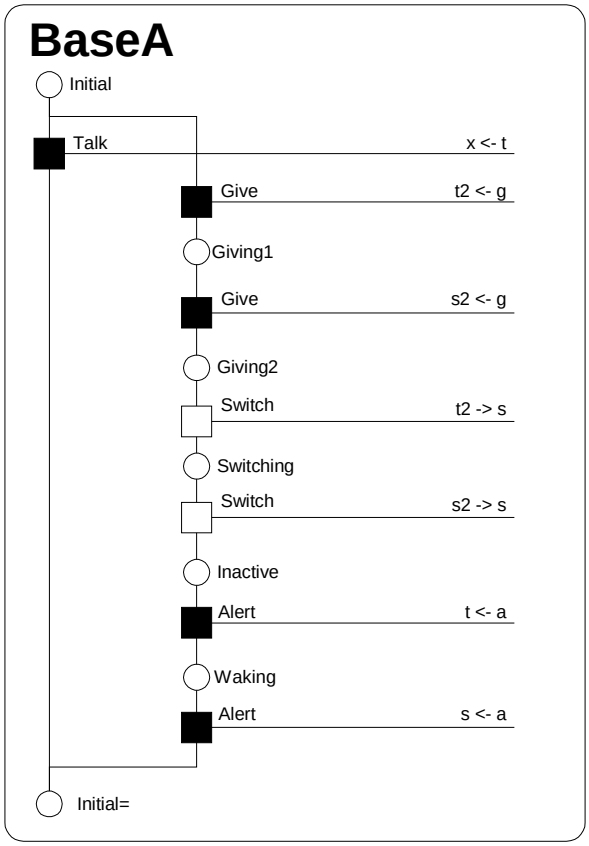


Figure 54: The BaseA process

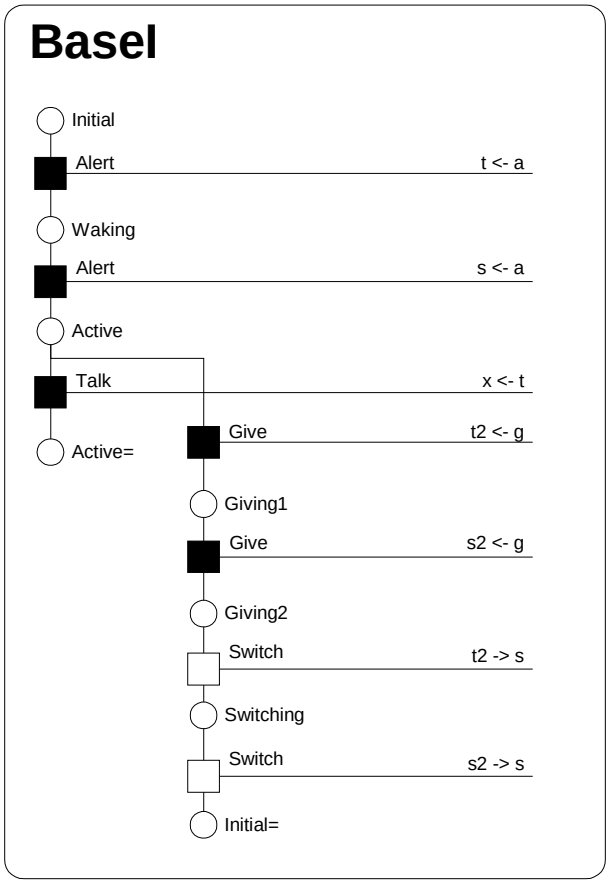


Figure 55: The BaseI process

Building the Centre process and assembling a Car, the two Bases and the Centre into a completed RDT model illustrates another difference between RDT and the pi-calculus concerning the effect of the ν operator of the pi-calculus compared with connections and the create event of RDT.

In RDT, channel names are local to the each process. Channels may be created at design time or at run time. Channels are created at design time by making connections between process instances in the model diagram. The effect of making a connection between two names (within process instances) is for the processes to find at runtime that the two names concerned both refer to the same channel thus enabling communication along the channel. So far as the RDT diagramming language is concerned, the actual channel is anonymous. The effect of a create event at runtime is similar. A channel is created which the creating process instance is able to refer to by the given name. The corresponding receive action (when this

new value is retrieved from the channel into which it was written) causes the new channel to be associated with the given local name in the other process instance.

Both of these behaviours may be replicated in a pi-calculus description of an RDT model, but if the exact meaning is to be reproduced, the scope of the ν operator in each case has to be limited to the particular pair of processes being connected. Thus it is not possible in RDT to reproduce the use of ν in the definition of the system shown in the system definition above. The problem being that the pair of channels providing the initial connection between the car and the active base is not known by the Centre process. They are known only to the car and the active base, but in the system as described in the pi-calculus, the centre sends this pair of channels to the (now active) base when the centre decides to instruct the car to change base a second time and resume communication with the first base. Neither can the centre know the alternate pair of channels which the car will use after being told to change bases since this connection is not in place at the start of execution.

As with the Base station issue, there are two ways to work around this problem. The first would be to manipulate the processes descriptions and the model diagrams to ensure that the centre is aware of all four of the channels which the car may use for communication with the bases. This is not an attractive solution in this example since it would involve creating no less than four connections between the centre and the car just for this purpose where there should be none.

The alternative solution is the one adopted in the RDT version of the system shown here. The initial connections between the Car and the BaseA (the initially active base) are shown on the model diagram. The Centre process operates in two phases. For the first two switching operations, it creates two new channels each time which are passed on to the car via the currently active base. After these two initial switching operations, the Centre now has two pairs of channels associated with local names for channels which it is able to send alternately to the car via the bases in the manner of the original pi-calculus model. The two channels used in

the initial connections between the Car and the BaseA are lost when the car first switches to the other base. The final version of the Centre process and the model diagram of the completed system are shown in Figure 56 and Figure 57.

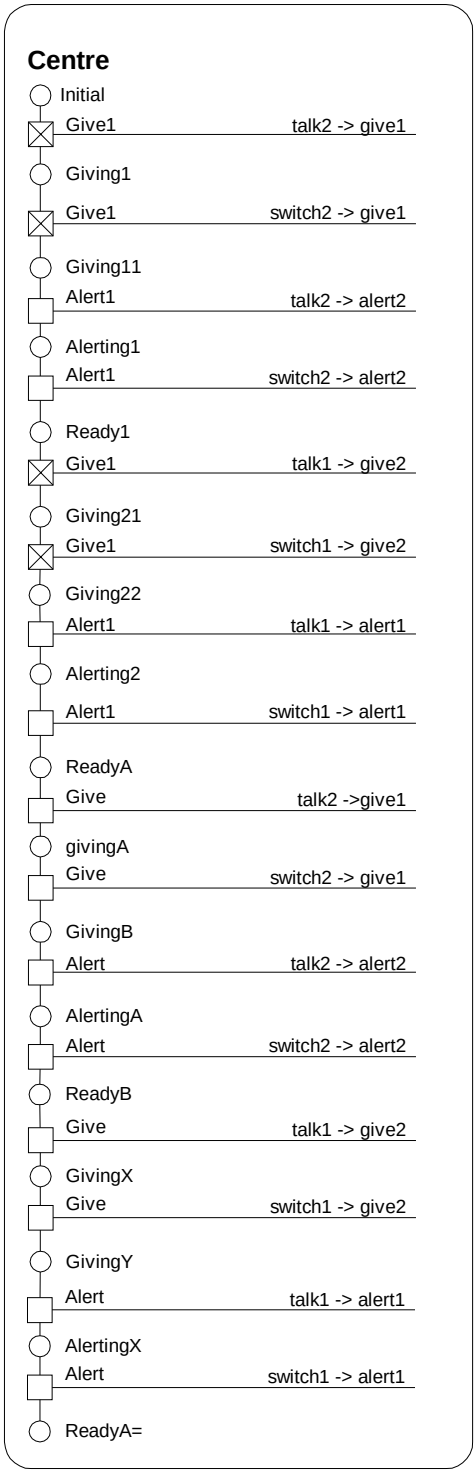


Figure 56: The Centre process

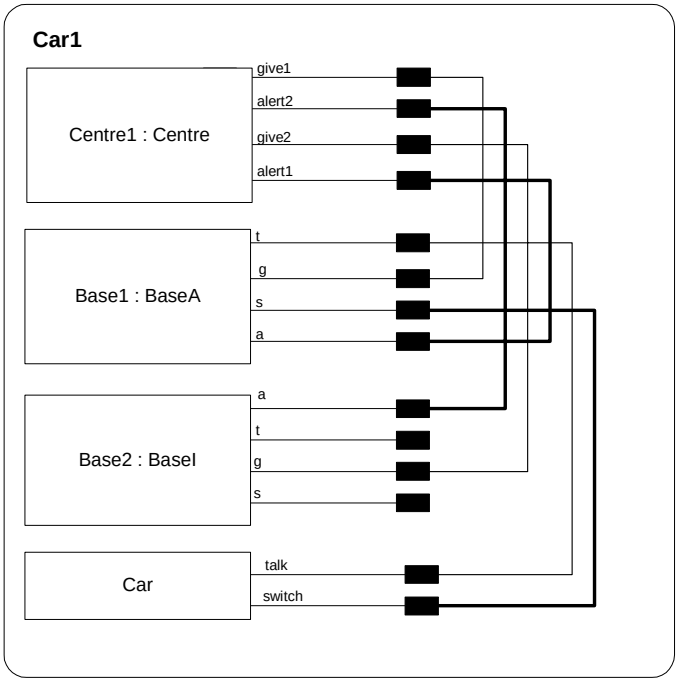


Figure 57: The System model

Building this model in the RDT model creation tool and executing it using RDX reveals the expected behaviour. The car is able to talk to the base which it is currently connected to and the centre is able to cause the car to switch to and fro between base stations.

It might seem reasonable to suppose that, possibly subject completing a transfer between base stations, the car in this model is always able to communicate. Demonstrating this to be true is just the type of task for which analysis with SPIN might be applied to this model. However, after converting this model to Promela using the RDTtoSPIN tool, the model needs some further work before the analysis may be performed.

The first task concerns the identifiers used in the model. “Active” which is used as a state name in the model is a reserved word in Promela so this has to be changed. Likewise, “switch” is a reserved word in the “C” programming language. SPIN operates by constructing a custom verifier according to the model and analysis task it is asked to perform. This verifier is written using “C” so the

“switch” identifier also needs to be altered. With these changes made to the automatically generated code, the SPIN is happy to perform its default analysis confirming the model is free from deadlock, etc..

SPIN along with other software model checkers is also able to check models for arbitrary properties relating to a particular model. Naturally to do this, the particular property to be checked needs to be communicated to SPIN. The manner in which this is done with SPIN is to specify the negation of the desired property as a “never” claim. In its analysis SPIN then establishes whether the “never” claim can ever become true and hence whether the original property is true. Accurate construction of these never claims in Promela is non-trivial, so SPIN provides an automated translation from a property expressed in LTL (linear temporal logic) to an appropriate “never” claim in Promela.

In the case of this model, the claim that the car may always communicate may be expressed in LTL, using the syntax understood by SPIN as, “[]<> t”, (“always, eventually t”) where t represents the occurrence of the car talking. The output from SPIN given this formula is shown in Figure 58.

```
       /*
        * Formula As Typed: []  <>  t
        * The Never Claim Below Corresponds
        * To The Negated Formula !([]  <>  t)
        * (formalizing violations of the original)
        */

never {    /*  !([]  <>  t)  */
T0_init:
       if
       :: (! ((t))) -> goto accept_S4
       :: (1) -> goto T0_init
       fi;
accept_S4:
       if
       :: (! ((t))) -> goto accept_S4
       fi;
}
```

Figure 58: SPIN generated "never" claim

Analysis of the model against this claim reveals that it is not true that the car will always eventually communicate. The explanation for this is that it is possible for

the Centre to endlessly frustrate the Car's attempts to communicate by constantly issuing switch instructions. However, if the system is constrained to ensure that, repeatedly given a particular choice, eventually the system executes all of the options (fairness) the property becomes true.

9.6 Using RDT

The aim of building RDT was to show that formal methods need not be daunting. The language seeks to achieve this by

1. Being approachable

2. Being small and easy to learn

3. Being powerful enough to be build useful models

4. Providing an easy path into exploiting the power of existing tools

Throughout the development of RDT the objective has been to encourage users who are "strangers to formal methods" and find them unattractive to use them by giving them a modelling language and associated support tools which would appeal to them in the way that Visual Basic appeals to new programmers.

9.6.1 Approachability

The first real contact between a potential modeller and a modelling tool is unlikely to be creating their own model. Instead they are going to see someone else's, either a colleague's work (in progress) or an example in the documentation. This initial contact needs to be as straightforward as possible. RolEnact was very successful here with its interface to the execution tool which potential users found appealing and easy to understand. However, the CORE project work revealed that this alone was not enough. Despite the appeal of the graphical interface to completed models, users enthusiasm waned as soon as they realised that building their own models required them to write code. They never took the time to learn how

to do it. RDT picks up on the success of the RolEnact execution tool by offering its own with a similar look and feel, but in addition has been able to carry the graphical, "point and click" style of operating into the model generation activity making it much more accessible than before. Using RDT with the support tools is quick and easy and it successfully insulates the modeller from the need to write code.

9.6.2 Size and ease of learning

Working with a tool or language for modelling or any other purpose for the first time requires some initial familiarisation.

The RDT language is very small, smaller even than RolEnact. It has just three types of event. Two of these, "Send" and "Receive", are so fundamental to communications that users will have prior knowledge of them and it is perfectly feasible build models using just these two.

Also, RDT uses just two types of diagram which appeal to the intuition of the user. The users' interactions with the diagrams is eased considerably because they only need to read them. The diagrams are drawn automatically by the support tools so the user never needs to learn the details of how it is done.

Together these features enable the new user to acquire a the knowledge they need to use RDT quickly and easily.

9.6.3 Capability

A concern with a language like RDT is that its usefulness is compromised by the necessary constraints associated making the language small and easy to assimilate. In the case of RDT this is not the case. The Barbershop and Mobile Phone examples above demonstrate that RDT is able to reproduce the behaviours of these two standard models. The Mobile Phones example in particular shows RDT being used to model a class of behaviour (passing channels along channels) which is not possible in many formal modelling languages. The Defence and Banking models

demonstrate that despite being such a small language and the simple interface of its support tools, it is feasible to use RDT to model quite substantial systems.

9.6.4 Providing a smooth path to using “heavy-weight” formal methods.

Arguably, software model checking using one of the established tools represents the acceptable face of formal methods as we know them today and SPIN is a good as any. We should be encouraging system designers and developers to use this type of technique. However, even using the XSPIN graphical interface, building and analysing models using SPIN is not easy. It requires the modeller to learn enough about Promela to be able to write syntactically correct code before they can even start.

Although ultimately less powerful, learning how to build a model with RDT is much simpler than learning to write code in Promela. However, once a modeller has built a model using RDT and its interactive model generation tool, they can then generate a version of their model which can be analysed by SPIN without learning about Promela at all. This enables them to learn about and benefit from the using an industrial-strength model checking tool without first having to learn its input language.

Chapter 10 Conclusion and further work

This work is concerned with the process of constructing software systems from collections of communicating components and, although the focus has been on large and widely distributed systems such as e-commerce systems, the ideas, problems and solutions apply equally to any size of system constructed from a collection of interacting parts.

The structure of software systems continues to move away from single, monolithic programs towards systems which are built from collections of interacting components. Today's systems are not only large and complex, often they also have aspects which overlap. Together these features make adopting a component based architecture attractive since it permits the task of building a system to be divided into more manageable pieces and the re-use of existing solutions to parts of the problem. In other engineering disciplines, building from components has proved to be a very powerful technique and its adoption has brought enormous benefits. It seems reasonable to expect that it will also be highly beneficial to software engineering. Additionally, with computer networks becoming ever more widespread, we are seeing more and more software solutions being built in which the component pieces of the solutions are working on different machines and communicating over a network. Altogether this means that we cannot avoid dealing with systems comprised of various interacting parts and we need to develop mature techniques for building this type of software systems.

Given that we are to construct a system from components, we first have to gather together a reasonable collection of components to address the requirements. Now we are now faced with two distinct problems. The first is how to assemble the pieces we have into a system in such a way that the various parts can co-operate without damaging each other. The second is to satisfy ourselves that the resulting system will actually work as it should.

The first of these problems, how to connect software components so that they are able to inter-operate has received considerable attention for a number of years and mature, robust solutions to the problem are readily available. Typically these solutions have a form of interface management at their core. Arguably, this aspect of building software systems from components has been solved.

Finding satisfactory solutions to the second problem is proving to be more difficult. One approach which is being pursued is to press interface management techniques into service in this arena too by adding behavioural elements to component interfaces. Aside from the enormous task which adding these elements to component interfaces represents, it brings with it a distinct drawback: it prevents innovative use of components because this type of interface regime would only allow components to be used in the contexts envisaged during their development. There is also the problem of "emergent" behaviour where a collection of components put together in a particular configuration is found to have properties which are quite unexpected.

The alternative is to abandon trying to get the overall behaviour of the assembled system "right" by considering its constituent parts in isolation and examine the whole system for appropriate behaviour. However, this immediately presents another problem. In order to examine the system and get the definitive answers about how it behaves, we need access to the system. But, at the time when this kind of insight is most crucial, during the design, the system has yet to be built. Applying the analysis to the completed system is not the answer, since the whole point of the exercise is to eliminate the wasted effort associated with building systems which turn out to be faulty by identifying and eliminating errant behaviour early in the development process. In fact, even where a complete system is available for analysis, performing convincing tests to establish properties such as the absence of deadlock in substantial distributed systems is difficult. This is where judicious use of models can help. Compared with building real systems, building models should be quick and easy, yet if they are carefully considered they can still represent a proposed system sufficiently well for analysis of the models to provide

useful insights into how the completed system will perform. Using appropriate abstractions in their model building, developers can construct models which, with the assistance of a suitable model checking tool, can be shown conclusively to satisfy various requirements whilst at the same time be free from problematic behaviour. These models can then form a sound basis from which to proceed with constructing the system itself. Because the models are quick and easy to build, a developer can afford to use them to experiment with and speculate about a variety of possible configurations for their system.

In some respects, the whole of the design of a system could be considered to be a modelling exercise in which the developer creates progressively more refined "models" of their proposed system until they reach a point where they feel they are ready to start the actual system development, but here we are only concerned with models describing communications (or interactions) between components within a system or between the system and the outside world. Experience shows that developers find reasoning about this aspect of the behaviour of their systems difficult. In addition, encumbered as they are with notions of how they intend the system to operate, they often overlook potentially problematic behaviour. This is the reason why, in order to get the best from these models they need to be exercised more rigorously than is feasible by simple inspection. In turn this requires the models to have sufficient formality to them to enable them to be executed and subjected to examination using model checking tools. However, notwithstanding the requirement of formality, building useful models need not be a major task. Their purpose is not to replicate the entire behaviour of a particular system. Instead, they should be built using appropriate abstractions to represent just the elements of the behaviour of the system relevant to the analysis to be performed. So, for example, the internal operations of the various component in the system can often be abstracted away. Even the smallest and most abstract of models can be useful.

Mature tools and techniques already exist for the construction and analysis of this kind of model in the form of various model checking tools and their respective

input languages. In contrast with systems testing, these tools perform automated, exhaustive searches of the entire state-space of their input systems (models). Consequently, their conclusions about whether a given model has a particular property are definitive. These tools are already powerful enough to analyse quite substantial systems. Few people would argue with the assertion that using them (perhaps together with other formal methods) to "debug" the design of systems would lead to an improvement in the quality and reliability of systems.

However, despite their strengths, model checking techniques are not in widespread use. They have acquired a reputation for being hard to understand and operate. The consequence is that developers who could benefit from using these techniques, but are generally under pressure to produce systems quickly, are reluctant to invest time into learning how to use them. It is unfortunate that, in general, any method to which the label "formal" (or "semi-formal") might be applied has come to be associated with "difficult" or "complicated" when for this type of work, what this really means is "precise" or "unambiguous". This perception of model checking systems (and formal methods in general) must be partially a consequence of the emphasis which their developers place on the ultimate capability of their tools and systems. They strive to give their systems ever more features and the capability to analyse ever larger models more quickly. Unfortunately, as the systems grow they become ever more daunting for the novice looking at them for the first time.

10.1 Reflection

The language described in this work sets out to provide the means for non-specialist software developers building distributed systems to create and analyse formal models of their systems as an aid to the design process. The language presented is formal to the extent that models generated using it may be executed without further elaboration. It also replaces the usual text based input and presentation of models with a system of diagrams. This adoption of diagrams means models constructed with the language appeal to the non-technical audience, permitting them to be used for presentation as well as analysis. Communication be-

tween processes in its models is via “point to point” channels. The modeller is permitted to select the length of channels.

Section 1.2 identifies a number of desirable features for a modelling system:

- An interface which permits the novice user to build “basic” models quickly and easily.

  RDT addresses this issue by using a graphical interface supported with a model generation tool which relieves the modeller of the burden of drawing the diagrams. Using the tool, the modeller’s task is reduced to first describing the events within their processes, followed by how to connect instances of their processes into complete systems. Throughout the construction task, the modeller works with dialogue boxes summoned from the menu system of the tool in a manner which would be familiar to a user of modern window based applications. Wherever practical, they select the desired value from a list of permissible values. This is both easier for the novice user and less error prone. The execution tool provides an interface to a completed model which is both attractive and easy to use.

  Whilst the RDT language is very small and the interface to the RDT model generation tool is appealing and easy to use, these on their own cannot make the generation of models effortless for the novice. Inevitably, however good (or easy) the modelling language and its support tools may be, the modeller still needs to grasp some essential features. There also remain elements of the model creation process such as understanding the real system and finding appropriate abstractions which the modeller has to address regardless of how they will build their models.

- It should be able to present its models in a diagrammatic form.

RDT is successful in this respect and goes further. It isn't just able to present its models in a diagrammatic form, the models are defined as diagrams. It also offers an appealing representation of a model for its initial analysis by execution. A RAD-like notation is used to describe processes and a separate diagram style is used to describe how collections of process instances are connected together into complete systems for analysis. Together these diagrams describe models completely. The style of the diagrams appeals to the intuition of potential users who are likely to be familiar with the general style of the diagrams from previous experience using UML.

For the execution of models, RDT provides an interface which although not "graphical" resembles the successful interface of the RolEnact stepper.

- The language should have an obvious sympathy with the system being built.

In common with most other modelling schemes, RDT models are built from communicating processes making it easy for the modeller to identify particular parts of the real system with individual processes in the model. In contrast with most other modelling systems, the selection of the communications paradigm in RDT was specifically motivated to permit the modeller to make a simple correspondence between the communications in their model and the real system. Hence, since the communication in most real systems is connection orientated, communications in RDT are by point to point channels. The communication along these channels may be synchronous. However we are seeing an increasing proportion of systems being constructed in which communications are asynchronous (such as those built using message passing middlewares) so RDT permits the modeller choose to make channels buffered. This enables them to build models of synchronous and asynchronous systems directly without the need for constructions in the models to match the communication pattern of the model to the real system.

- The modeller should be able to proceed from model generation to analysis smoothly.

Once constructed, an RDT model may be analysed by execution or conversion to Promela code for analysis with SPIN. Based on the popular RolEnact stepper, the RDT execution tool permits the modeller to take a completed model and execute it with an absolute minimum of effort. However, although the RDTtoSPIN tool performs an automatic conversion to Promela code, the current route to the analysis of RDT models using SPIN still requires the modeller to have a working knowledge of SPIN. In particular the modeller needs to know how to present a model to SPIN for checking and interpret the output. Also, in their current form, the RDT tools don't provide support to the modeller wishing to specify properties of their model for SPIN to check.

In its present form, the RDT language is limited. This is intentional and an inevitable consequence of its minimal nature. The language might benefit from some additional features but these will need to be added with care. There is a risk that adding extra features indiscriminately would enlarge the language to the point that it no longer satisfied the original objective of being simple to understand and small enough for a working programmer to learn to use it in a few hours.

When considered together with its support tools, RDT demonstrates that it is feasible for inexperienced modellers using a simple, graphical interface to build models which can stand up to serious formal analysis.

We need to tempt "ordinary" working system developers into using formal or "semi-formal" techniques, but if we are to do this we need to provide them with a route into using tools like software model checkers which offers them the opportunity to enjoy some of the benefits in return for the most modest of initial effort at the outset. This is the purpose of the RDT language. The language is graphical and designed to be easily understood by any developer, particularly if they are familiar with the common elements of something like UML and small enough for

all of the important concepts to be assimilated in no more than a few hours. The support tools for the language relieve the modeller of the need to draw the diagrams or learn the syntax of an input language and smooth the path to the analysis of models using a commercial-strength model checker. Despite having the formal basis that they need to permit them to be executable, RDT models have the appearance of belonging to an informal diagramming technique.

10.2 Potential enhancements to RDT

The following is a list of potential additions to RDT:

- Hierarchical features.

  It is easy to see the attraction of such a feature. However, adding a simple facility which permits some collection of events (or processes) into a box does not really address the problem since, although some communications which take place entirely within the box could be hidden, many will still remain. The problem is that those communications which do remain belong to the interactions between the components at the lowest level of the model. What is really needed is a second mechanism by which lower levels details of these communications may be abstracted away as part of the same operation which is far from straightforward.

- Introduction of an explicit conditional construct.

  Processes in RDT are able to make choices. Implicitly, they are also able to act conditionally according to some aspects of their state. For example, a process which has a local name for a channel which has yet to acquire a value is unable to write such a channel. However, it is felt that users from the target audience for this type of tool will be very familiar with the “if” type of construct found in most programming languages and it may make them feel more comfortable with RDT if it permitted them to employ this type of construct.

Doing this would require the enhancement of the process diagrams to show an explicit test followed by branches for subsequent execution following the test.

- Addition of an event equivalent to the RolEnact "action".
  RolEnact included a event called an "action" which was a unilateral change of state by an individual process. This has not been included in RDT because it was felt that there is no essential difference between a process which, say, has to perform some "action" before being able to take part in some other event and a process which simply elects not to perform that event (for a period of time) and that consequently it adds little to the expressiveness of the language. However, there may be some merit in adding such an event to RDT simply because users feel that they would like to have it available.

10.3 Improvements to the RDT tools

Potential improvements to the RDT tool support include:

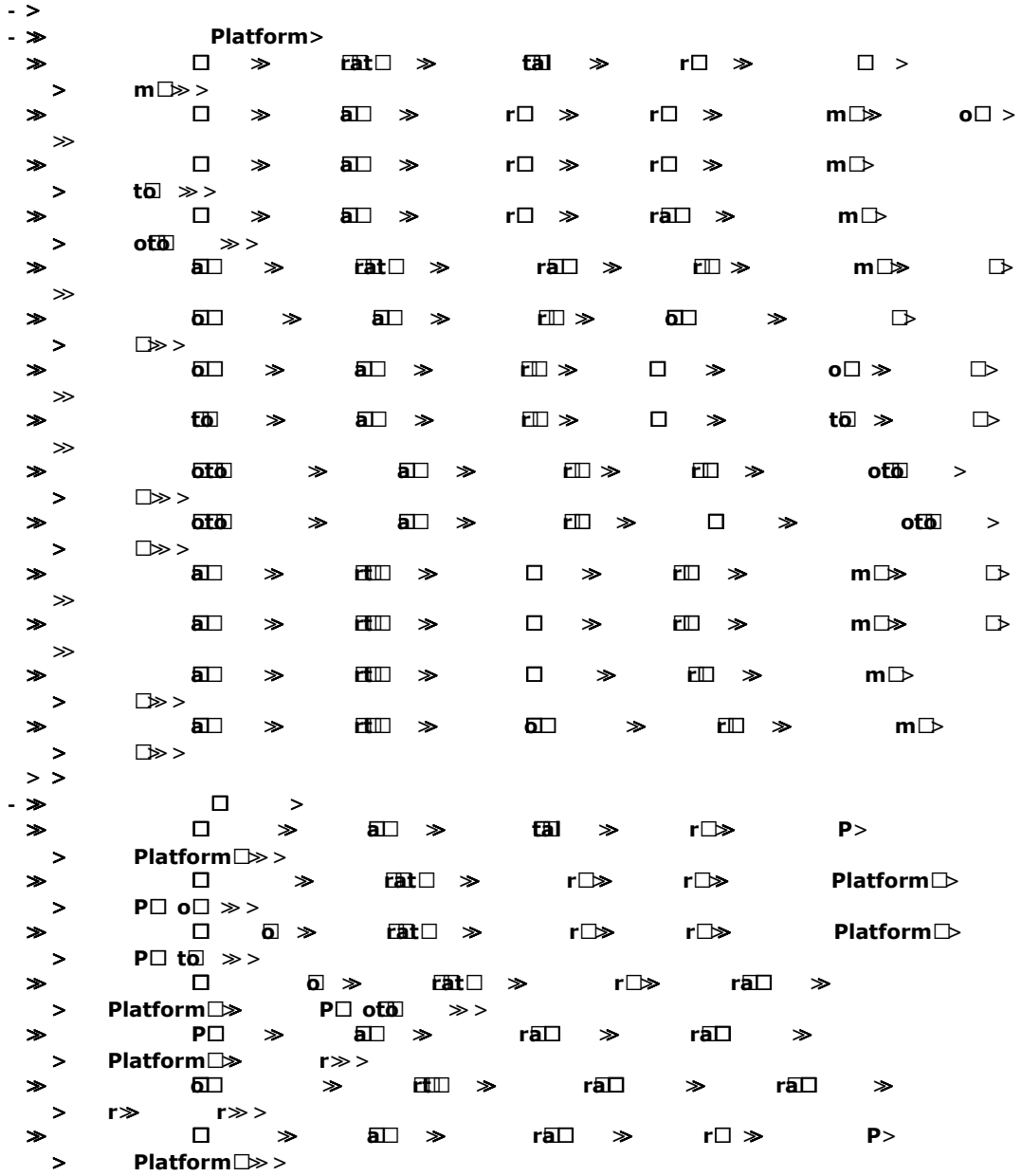
- Addressing the issue of the create event when a model is translated into Promela. Although the language permits places no restriction on the number of times an event of type "Create" may occur, the translation to Promela in the current version of the RDTtoSPIN tool imposes a (user definable) limit. At first sight, it would appear that not limiting the number of times such an event can occur would require the use of an unlimited number of channels. However, since a channel becomes irretrievably lost if no process holds a reference to it, the maximum number of channels which can ever take further part in the execution any RDT model is the sum of the number of ports on each process instance in the model. Consequently, if the execution tool were to generate Promela code which instantiates a suitable number of channels and "reclaimed" those which became "remaindered" for subsequent re-allocation, a "Create" event could be permitted to occur an unlimited number of times. Further investigation would be necessary before adopting such a scheme since it may have a deleterious impact on the performance of the model checker

- The more obvious, and perhaps the more ambitious, would be the integration of the current collection of three tools into a single application which would enable the modeller to progress smoothly from building a model to interactive execution of a model and finally to generate the Promela code for further analysis. It may also be practical to provide the modeller with facilities to perform analysis of their model with SPIN from within the same environment in the style of XSPIN.

Appendix A Example models in XML[1]

The typical examples of the models described in Chapter 9 are given below (as formatted by Internet Explorer)

A.1 Defence (without communications)



[1] In the implementation of the RDT tools, the “Send” and “Receive” events have been renamed to “Write” and “Read” respectively.

```
  >     ≫            P□≫      Platform≫ >
- >       >
  ≫       >       □□□   ≫         P≫ >
  ≫       >       P□≫       □  ≫ >
  > >     >
- >       >
  ≫       >       □□□   ≫         P≫ >
  ≫       >       P□≫       □  ≫ >
  > >     >
  >     ≫            P□≫      Platform≫ >
- >       >
  ≫       >       □□□   ≫         P≫ >
  ≫       >       P□≫       □  ≫ >
  > >     >
  > ≫
  > >
```

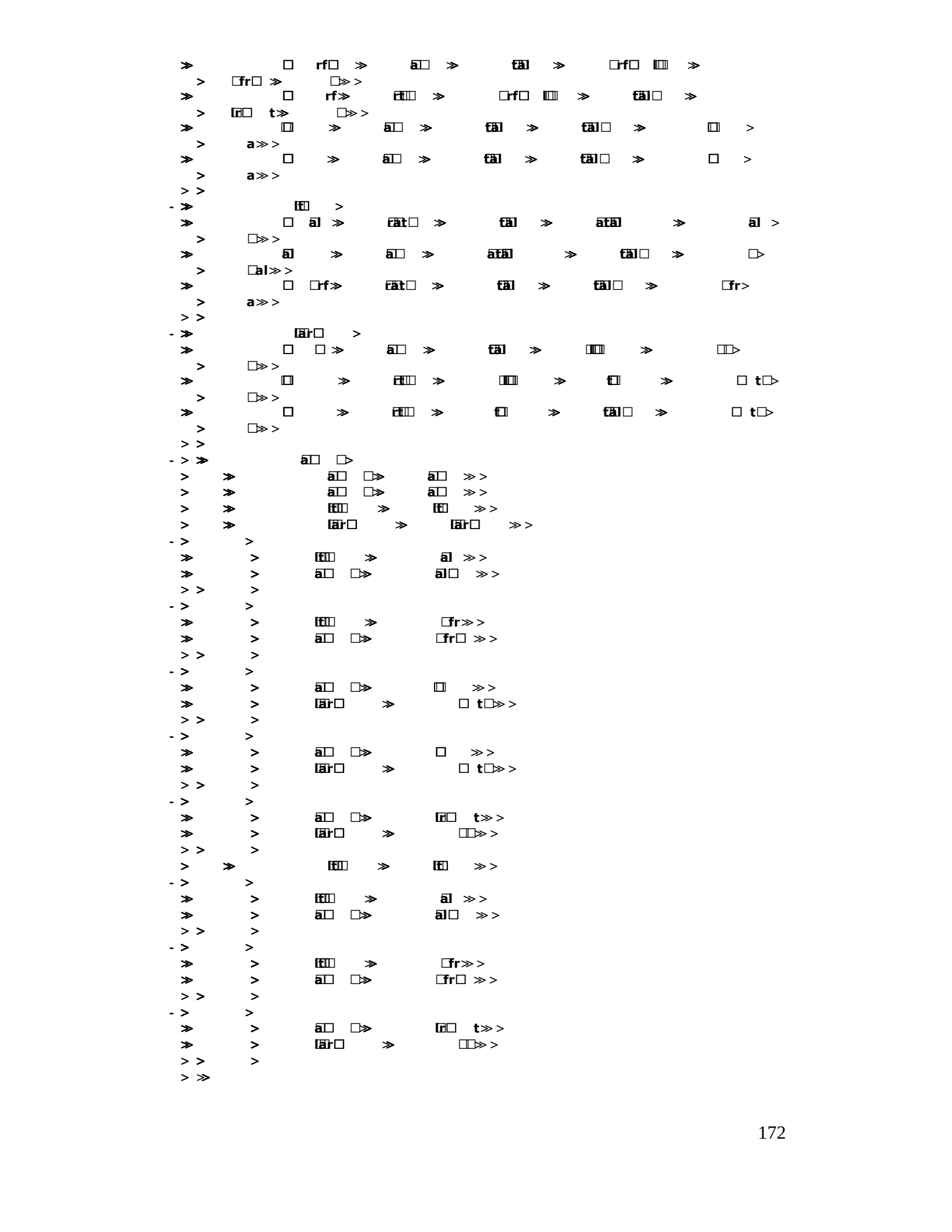
A.2 Defence (with communication)

```
- >
- ≫              Platform>
  ≫          □  ≫     rät□  ≫      tä□  ≫     r□  ≫       □  >
    >     m□≫ >
  ≫          □  ≫     a□  ≫      r□  ≫      r□  ≫      m□>
    >     o□m□ ≫ >
  ≫          □  ≫     a□  ≫      r□  ≫      r□  ≫      m□>
    >     o□m□ ≫ >
  ≫          □  ≫     a□  ≫      r□  ≫      r□  ≫      m□>
    >     o□  ≫ >
  ≫          □  ≫     a□  ≫      r□  ≫      r□  ≫      m□≫     o□ >
    ≫
  ≫          □  ≫     a□  ≫      r□  ≫      r□  ≫      m□>
    >     to□  ≫ >
  ≫          □  ≫     a□  ≫      r□  ≫      rä□  ≫      m□>
    >     oto□  ≫ >
  ≫          a□  ≫     rät□  ≫      rä□  ≫     r□  ≫      m□≫     □>
    ≫
  ≫          o□  ≫     a□  ≫      r□  ≫      o□  ≫      o□  >
    >     □≫ >
  ≫          o□  ≫     a□  ≫      r□  ≫      □  ≫      o□ ≫     □>
    ≫
  ≫          to□  ≫     a□  ≫      r□  ≫      □  ≫      to□ ≫     □>
    ≫
  ≫          oto□  ≫     a□  ≫      r□  ≫      r□  ≫      oto□  >
    >     □≫ >
  ≫          oto□  ≫     a□  ≫      r□  ≫      □  ≫      oto□  >
    >     □≫ >
  ≫          a□  ≫     rt□  ≫      □  ≫      r□  ≫      m□≫     □>
    ≫
  ≫          a□  ≫     rt□  ≫      □  ≫      r□  ≫      m□≫     □>
    ≫
  ≫          a□  ≫     rt□  ≫      □  ≫      r□  ≫      m□>
    >     □≫ >
  ≫          a□  ≫     rt□  ≫      o□  ≫      r□  ≫      m□>
    >     □≫ >
  ≫          o□m□ ≫     rt□  ≫      □  ≫      □o□mä□  ≫      □>
    >     o□m□ ≫ >
  ≫          o□mä□  ≫     a□  ≫      □o□mä□  ≫      □o□m□  ≫      o□m□ >
    >     r□≫ >
  ≫          o□m□  ≫     a□  ≫      □o□m□  ≫      □  ≫      o□m□ >
    >     □≫ >
```

```
≫            P□   ≫      ā□   ≫       rā□    ≫      rā□       ≫
  >   Platform□≫       r≫>
≫            P□   ≫      ā□   ≫       rā□    ≫      rā□       ≫
  >   Platform□≫       r≫>
≫            ō□       ≫      rtt□   ≫       rā□      ≫      rā□      ≫
  >   P□ ō□   ≫       P□ ō□   ≫>
≫              □      ō   ≫      rtt□   ≫       rā□      ≫      rā□      ≫
  >   P□ o□ ≫       ōm□ ≫>
≫              □      □r□ ≫      rtt□   ≫       rā□      ≫      rā□      ≫
  >   P□ tō  ≫       ōm□ ≫>
≫              □      ō□  □r□ ≫      rtt□   ≫       rā□      ≫      rā□          >
  >       P□ otō    ≫       ōm□ ≫>
≫              □      ō□  □r□  ≫      rtt□   ≫       rā□      ≫      rā□      >
  >       P□ otō    ≫       ōm□ ≫>
≫              ō□       ≫      rtt□   ≫       rā□      ≫      rā□      ≫
  >   P□ ō□   ≫       P□ ō□   ≫>
≫              □        ≫      rtt□   ≫       rā□      ≫      rā□      ≫
  >   P□ o□ ≫       ōm□ ≫>
≫              □      □r□ ≫      rtt□   ≫       rā□      ≫      rā□      ≫
  >   P□ tō  ≫       ōm□ ≫>
≫              □        □r□ ≫      rtt□   ≫       rā□      ≫      rā□          >
  >       P□ otō    ≫       ōm□ ≫>
≫              □        □r□  ≫      rtt□   ≫       rā□      ≫      rā□      >
  >       P□ otō    ≫       ōm□ ≫>
≫              ō□       ≫      rtt□   ≫       rā□      ≫      rā□      ≫
  >   P□ ō□   ≫       P□ ō□   ≫>
≫              □        ≫      rtt□   ≫       rā□      ≫      rā□      ≫
  >   P□ o□ ≫       ōm□ ≫>
≫              □      ō  ≫      rtt□   ≫       rā□      ≫      rā□      ≫
  >   P□ tō  ≫       ōm□ ≫>
≫              □        ō  ≫      rtt□   ≫       rā□      ≫      rā□          >
  >       P□ otō    ≫       ōm□ ≫>
≫              □        ō□  ≫      rtt□   ≫       rā□      ≫      rā□      >
  >       P□ otō    ≫       ōm□ ≫>
  > >
- > ≫            fōl□          >
  >     ≫            □□□   ≫      □      ≫>
  >     ≫            P□≫      Platform≫>
  >     ≫            P□≫      Platform≫>
- >        >
  ≫        >       □□□   ≫         P≫>
  ≫        >       P□≫          □  ≫>
  > >      >
- >        >
  ≫        >       □□□   ≫         P≫>
  ≫        >       P□≫          □  ≫>
  > >      >
  >     ≫            P□≫      Platform≫>
- >        >
  ≫        >       □□□   ≫         P≫>
  ≫        >       P□≫          □  ≫>
  > >      >
  > ≫
  > >
```

A.3 Banking

```
- >
- ≫             ā□  >
  ≫           āl□       ≫      a□   ≫       tāl   ≫      āl□   l□□  ≫
    >   āl□  ≫       □≫>
  ≫           āl       ≫      rāt□  ≫       āl□   l□□  ≫      tāl□  ≫         □▷
    >      □al≫>
```



```
  > >
```

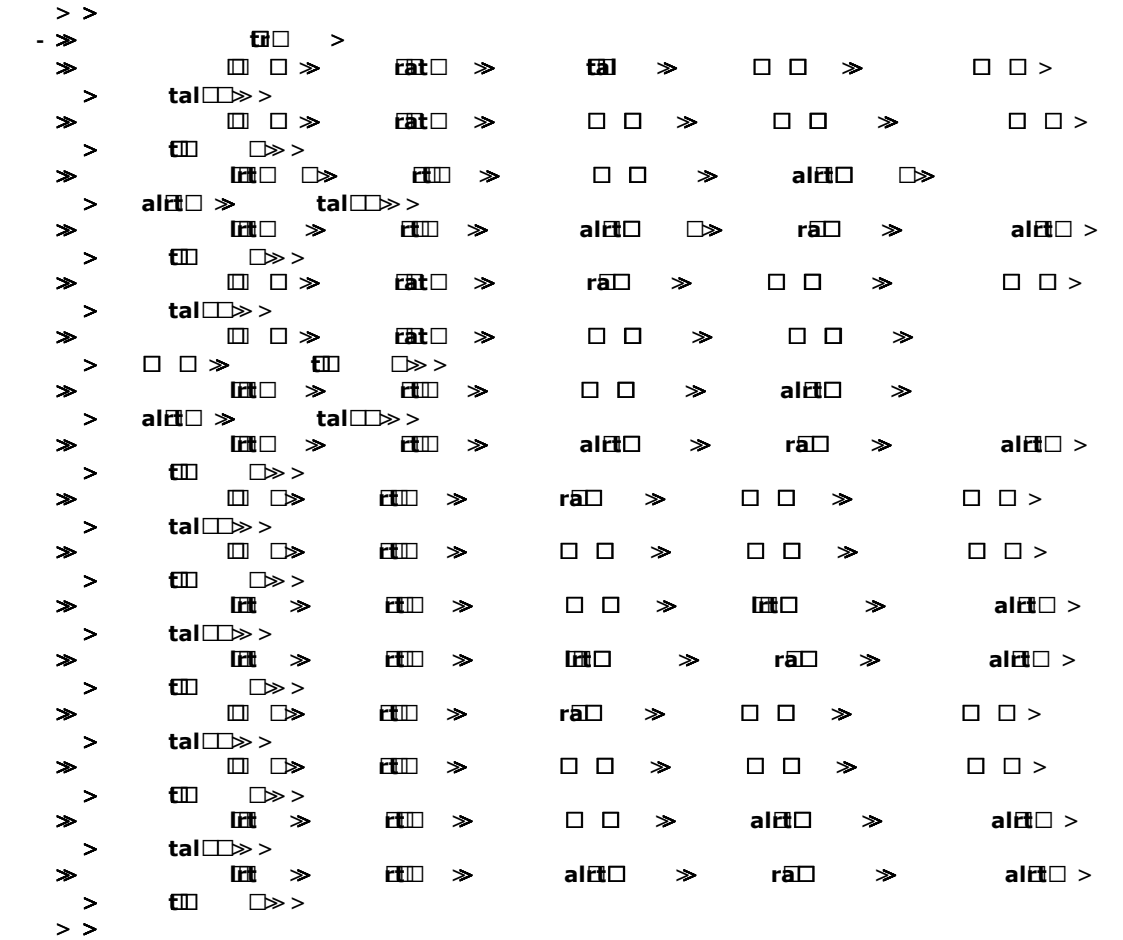
A.4 Buffers

```
- >
- >>              o□ r□ >
  >>              □    >>      rtt□   >>       tal    >>      tal□    >>          □ t>
    >      □ t>> >
  > >
- >>              □   >
  >>              al□  >>      a□   >>       tal    >>      tal□    >>          □□>
    >      al  >> >
  > >
- >>              □ ffr□ >
  >>              al□  >>      a□   >>       tal    >>      □□l>>           □□>>       al  >
    >>
  >>              rtt□   >>      rtt□   >>       □□l>>      tal□    >>          □ t>
    >      al  >> >
  > >
- > >>              o□   ff>
  >     >>              □ >>      o□ r□ >> >
  >     >>              □   □>>       □   >> >
- >        >
  >>        >        □ >>          □ t>> >
  >>        >        □   □>>          □□>> >
  > >       >
  > >>
- > >>              □      ff>
  >     >>              □ >>      o□ r□ >> >
  >     >>              □ >>      □ ffr□ >> >
  >     >>              □ □>>       □   >> >
- >        >
  >>        >        □ >>          □ t>> >
  >>        >        □ >>          □□>> >
  > >       >
- >        >
  >>        >        □ >>          □ t>> >
  >>        >        □ □>>           □□>> >
  > >       >
  > >>
- > >>              □o□   ff>
  >     >>              □ >>      o□ r□ >> >
  >     >>              □ ff□>>       □ ffr□ >> >
  >     >>              □ ff□>>       □ ffr□ >> >
  >     >>              □ □>>       □   >> >
- >        >
  >>        >        □ >>          □ t>> >
  >>        >        □ ff□>>          □□>> >
  > >       >
- >        >
  >>        >        □ ff□>>          □ t>> >
  >>        >        □ ff□>>          □□>> >
  > >       >
- >        >
  >>        >        □ ff□>>          □ t>> >
  >>        >        □ □>>           □□>> >
  > >       >
  > >>
- > >>              □□r□    ff>
  >     >>              □ >>      o□ r□ >> >
  >     >>              □ ff□>>       □ ffr□ >> >
  >     >>              □ ff□>>       □ ffr□ >> >
```

```
  >     ≫             □ ff□≫      □ fff□ ≫ >
  >     ≫             □ □≫        □    ≫ >
- >       >
  ≫       >       □ ≫           □ t≫ >
  ≫       >       □ ff□≫          □□≫ >
  > >     >
- >       >
  ≫       >       □ ff□≫          □ t≫ >
  ≫       >       □ ff□≫          □□≫ >
  > >     >
- >       >
  ≫       >       □ ff□≫          □ t≫ >
  ≫       >       □ ff□≫          □□≫ >
  > >     >
- >       >
  ≫       >       □ ff□≫          □ t≫ >
  ≫       >       □ □≫          □□≫ >
  > >     >
  > ≫
  > >
```

A.5 Mobile Phones

```
- >
- ≫             ā□r  >
  ≫           □al□≫     rtt□  ≫       tā□l   ≫       tā□l□   ≫         tal□
    >     tal□≫ >
  ≫           t□□   ≫     ā□□  ≫       tā□l   ≫       t□□   □ ≫         t□□   >
    >     tal□≫ >
  ≫           t□□   ≫     ā□□  ≫       t□□   □ ≫       tā□l□   ≫
    >   t□□   ≫       t□□   ≫ >
  > >
- ≫             ā□   >
  ≫           □al□≫     ā□□  ≫       tā□l   ≫       tā□l□   ≫         t≫       □
    ≫
  ≫           □□ □≫     ā□□  ≫       tā□l   ≫       □ □   ≫         □≫       t□
    ≫
  ≫           □□ □≫     ā□□  ≫       □ □   ≫       □ □   ≫         □
    >     □ ≫ >
  ≫           t□□   ≫     rtt□  ≫       □ □   ≫       t□□   □ ≫         □
    >     t□≫ >
  ≫           t□□   ≫     rtt□  ≫       t□□   □ ≫       āt□□   □≫         □
    >     □ ≫ >
  ≫           □rt   ≫     ā□□  ≫       āt□□   □≫       ā□□ □ ≫         a>
    >     t≫ >
  ≫           □rt   ≫     ā□□  ≫       ā□□ □ ≫       tā□l□   ≫         a>
    >     □≫ >
  > >
- ≫             ā□   >
  ≫           □rt   ≫     ā□□  ≫       tā□l   ≫       ā□□ □ ≫         a≫       t>
    ≫
  ≫           □rt   ≫     ā□□  ≫       ā□□ □ ≫       at□□ □≫         a≫       □
    ≫
  ≫           □al□≫     ā□□  ≫       at□□ □≫       at□□ □ ≫         t≫       □
    ≫
  ≫           □□ □≫     ā□□  ≫       at□□ □≫       □ □   ≫         □
    >     t□≫ >
  ≫           □□ □≫     ā□□  ≫       □ □   ≫       □ □   ≫         □
    >     □ ≫ >
  ≫           t□□   ≫     rtt□  ≫       □ □   ≫       t□□   □ ≫         □
    >     t□≫ >
  ≫           t□□   ≫     rtt□  ≫       t□□   □ ≫       tā□l□   ≫         □
    >     □ ≫ >
```



> >

Appendix B Example models in Promela[1]

Taking the example models and converting them to Promela using the RDT2SPIN tool produces the following code:

B.1 Defence(without communication)

```
/* Generated from file C:\rjw1\PhD\RDTModels\Defence3C.xml */

#define CHLEN 0
#define CHNO 20

proctype Platform(chan WV, me, none, one, two, onetwo, s, A, B)
{
int i = 0;
chan supp[CHNO] = [CHLEN] of {chan};

initial:
if
::i < CHNO; atomic { me = supp[i]; WV!me; i = i +1 }
goto reg0;
fi;

reg0:
if
:: me?none; goto reg1;
fi;

reg1:
if
:: me?one; goto reg2;
fi;

reg2:
if
:: me?two; goto reg3;
fi;

reg3:
if
:: me?onetwo; goto ready;
fi;

ready:
if
::i < CHNO; atomic { s = supp[i]; me!s; i = i +1 }
goto sr1;
fi;

sr1:
if
:: none?s; goto SeeNone;
:: one?A; goto See1;
:: two?B; goto See2;
:: onetwo?A; goto sr1A;
fi;

sr1A:
if
:: onetwo?B; goto See12;
```

[1] In the implementation of the RDT tools, the “Send” and “Receive” events have been renamed to “Write” and “Read” respectively.

```
fi;

See1:
if
:: me!s; goto sr1;
fi;

See2:
if
:: me!s; goto sr1;
fi;

See12:
if
:: me!s; goto sr1;
fi;

SeeNone:
if
:: me!s; goto sr1;
fi;

}

proctype WView(chan P, Platform1, P1_none, P1_one, P1_two, P1_onetwo, r,
Platform2, P2_none, P2_one, P2_two, P2_onetwo, Platform3, P3_none, P3_one, P3_two,
P3_onetwo)
{
int i = 0;
chan supp[CHNO] = [CHLEN] of {chan};

initial:
if
:: P?Platform1; goto r0;
fi;

r0:
if
::i < CHNO; atomic { P1_none = supp[i]; Platform1!P1_none; i = i +1 }
goto r1;
fi;

r1:
if
::i < CHNO; atomic { P1_one = supp[i]; Platform1!P1_one; i = i +1 }
goto r2;
fi;

r2:
if
::i < CHNO; atomic { P1_two = supp[i]; Platform1!P1_two; i = i +1 }
goto r3;
fi;

r3:
if
::i < CHNO; atomic { P1_onetwo = supp[i]; Platform1!P1_onetwo; i = i +1 }
goto ready1;
fi;

ready1:
if
:: Platform1?r; goto reading1;
:: P?Platform2; goto r10;
fi;

reading1:
if
:: r!r; goto ready1;
fi;

r10:
if
```

```
::i < CHNO; atomic { P2_none = supp[i]; Platform2!P2_none; i = i +1 }
goto r11;
fi;

r11:
if
::i < CHNO; atomic { P2_one = supp[i]; Platform2!P2_one; i = i +1 }
goto r12;
fi;

r12:
if
::i < CHNO; atomic { P2_two = supp[i]; Platform2!P2_two; i = i +1 }
goto r13;
fi;

r13:
if
::i < CHNO; atomic { P2_onetwo = supp[i]; Platform2!P2_onetwo; i = i +1 }
goto ready2;
fi;

ready2:
if
:: Platform1?r; goto reading2;
:: Platform2?r; goto reading22;
:: P?Platform3; goto r20;
fi;

reading2:
if
:: P1_none!r; goto ready2;
:: P1_one!r; goto ready2;
fi;

reading22:
if
:: P2_none!r; goto ready2;
:: P2_one!r; goto ready2;
fi;

r20:
if
::i < CHNO; atomic { P3_none = supp[i]; Platform3!P3_none; i = i +1 }
goto r21;
fi;

r21:
if
::i < CHNO; atomic { P3_one = supp[i]; Platform3!P3_one; i = i +1 }
goto r22;
fi;

r22:
if
::i < CHNO; atomic { P3_two = supp[i]; Platform3!P3_two; i = i +1 }
goto r23;
fi;

r23:
if
::i < CHNO; atomic { P3_onetwo = supp[i]; Platform3!P3_onetwo; i = i +1 }
goto ready3;
fi;

ready3:
if
:: Platform1?r; goto reading31;
:: Platform2?r; goto reading32;
:: Platform3?r; goto reading33;
fi;

reading31:
```

```
if
:: P1_none!P1_none; goto ready3;
:: P1_one!Platform2; goto ready3;
:: P1_two!Platform3; goto ready3;
:: P1_onetwo!Platform2; goto reading3123;
fi;

reading3123:
if
:: P1_onetwo!Platform3; goto ready3;
fi;

reading32:
if
:: P2_none!P2_none; goto ready3;
:: P2_one!Platform1; goto ready3;
:: P2_two!Platform3; goto ready3;
:: P2_onetwo!Platform1; goto reading3213;
fi;

reading3213:
if
:: P2_onetwo!Platform3; goto ready3;
fi;

reading33:
if
:: P3_none!P3_none; goto ready3;
:: P3_one!Platform1; goto ready3;
:: P3_two!Platform2; goto ready3;
:: P3_onetwo!Platform1; goto reading3312;
fi;

reading3312:
if
:: P3_onetwo!Platform2; goto ready3;
fi;

}



 init
{ atomic {
chan ch0 = [CHLEN] of {chan};
chan nch0 = [0] of {chan};
chan nch1 = [0] of {chan};
chan nch2 = [0] of {chan};
chan nch3 = [0] of {chan};
chan nch4 = [0] of {chan};
chan nch5 = [0] of {chan};
chan nch6 = [0] of {chan};
chan nch7 = [0] of {chan};
chan nch8 = [0] of {chan};
chan nch9 = [0] of {chan};
chan nch10 = [0] of {chan};
chan nch11 = [0] of {chan};
chan nch12 = [0] of {chan};
chan nch13 = [0] of {chan};
chan nch14 = [0] of {chan};
chan nch15 = [0] of {chan};
chan nch16 = [0] of {chan};
chan nch17 = [0] of {chan};
chan nch18 = [0] of {chan};
chan nch19 = [0] of {chan};
chan nch20 = [0] of {chan};
chan nch21 = [0] of {chan};
chan nch22 = [0] of {chan};
chan nch23 = [0] of {chan};
chan nch24 = [0] of {chan};
chan nch25 = [0] of {chan};
chan nch26 = [0] of {chan};
chan nch27 = [0] of {chan};
```

```
chan nch28 = [0] of {chan};
chan nch29 = [0] of {chan};
chan nch30 = [0] of {chan};
chan nch31 = [0] of {chan};
chan nch32 = [0] of {chan};
chan nch33 = [0] of {chan};
chan nch34 = [0] of {chan};
chan nch35 = [0] of {chan};
chan nch36 = [0] of {chan};
chan nch37 = [0] of {chan};
chan nch38 = [0] of {chan};
chan nch39 = [0] of {chan};

run WView(ch0, nch0, nch1, nch2, nch3, nch4, nch5, nch6, nch7, nch8, nch9, nch10,
nch11, nch12, nch13, nch14, nch15);
run Platform(ch0, nch16, nch17, nch18, nch19, nch20, nch21, nch22, nch23);
run Platform(ch0, nch24, nch25, nch26, nch27, nch28, nch29, nch30, nch31);
run Platform(ch0, nch32, nch33, nch34, nch35, nch36, nch37, nch38, nch39);
} };
```

B.2 Defence (with communication)

```
/* Generated from file C:\rjw1\PhD\RDTModels\ComDefence8.xml */

#define CHLEN 0
#define CHNO 20

proctype Platform(chan WV, me, com1, com2, none, one, two, onetwo, s, x, A, B, r1,
A2, B2)
{
int i = 0;
chan supp[CHNO] = [CHLEN] of {chan};

initial:
if
::i < CHNO; atomic { me = supp[i]; WV!me; i = i +1 }
goto reg0;
fi;

reg0:
if
:: me?com1; goto reg1;
fi;

reg1:
if
:: me?com2; goto reg2;
fi;

reg2:
if
:: me?none; goto reg3;
fi;

reg3:
if
:: me?one; goto reg4;
fi;

reg4:
if
:: me?two; goto reg5;
fi;

reg5:
if
:: me?onetwo; goto ready;
fi;

ready:
```

```
if
::i < CHNO; atomic { s = supp[i]; me!s; i = i +1 }
goto sr1;
:: A!com1; goto xCom1a;
:: B!com2; goto yCom2a;
:: com1?A; goto RecCom11A;
:: com2?B; goto RecCom11B;
fi;

sr1:
if
:: none?x; goto ready;
:: one?A; goto ready;
:: two?B; goto ready;
:: onetwo?A; goto sr1A;
fi;

sr1A:
if
:: onetwo?B; goto ready;
fi;

xCom1a:
if
:: com1?r1; goto xCom1b;
fi;

xCom1b:
if
:: com1?x; goto ready;
:: r1?B; goto ready;
fi;

yCom2a:
if
:: com2?r1; goto yCom2b;
fi;

yCom2b:
if
:: com2?x; goto ready;
:: r1?A; goto ready;
fi;

RecCom11A:
if
::i < CHNO; atomic { A2 = supp[i]; A!A2; i = i +1 }
goto RecCom12A;
fi;

RecCom12A:
if
:: A!A; goto ready;
:: A2!B; goto ready;
fi;

RecCom11B:
if
::i < CHNO; atomic { B2 = supp[i]; B!B2; i = i +1 }
goto RecCom12B;
fi;

RecCom12B:
if
:: B!B; goto ready;
:: B2!A; goto ready;
fi;

}

proctype WView(chan P, Platform1, Com1, Com2, P1_none, P1_one, P1_two, P1_onetwo,
r, Platform2, Com3, Com4, P2_none, P2_one, P2_two, P2_onetwo, Platform3, Com5,
Com6, P3_none, P3_one, P3_two, P3_onetwo)
```

```
{
int i = 0;
chan supp[CHNO] = [CHLEN] of {chan};

initial:
if
:: P?Platform1; goto r0;
fi;

r0:
if
::i < CHNO; atomic { Com1 = supp[i]; Platform1!Com1; i = i +1 }
goto r1;
fi;

r1:
if
::i < CHNO; atomic { Com2 = supp[i]; Platform1!Com2; i = i +1 }
goto r2;
fi;

r2:
if
::i < CHNO; atomic { P1_none = supp[i]; Platform1!P1_none; i = i +1 }
goto r3;
fi;

r3:
if
::i < CHNO; atomic { P1_one = supp[i]; Platform1!P1_one; i = i +1 }
goto r4;
fi;

r4:
if
::i < CHNO; atomic { P1_two = supp[i]; Platform1!P1_two; i = i +1 }
goto r5;
fi;

r5:
if
::i < CHNO; atomic { P1_onetwo = supp[i]; Platform1!P1_onetwo; i = i +1 }
goto ready1;
fi;

ready1:
if
:: Platform1?r; goto reading1;
:: P?Platform2; goto r10;
fi;

reading1:
if
:: P1_none!r; goto ready1;
fi;

r10:
if
::i < CHNO; atomic { Com3 = supp[i]; Platform2!Com3; i = i +1 }
goto r11;
fi;

r11:
if
::i < CHNO; atomic { Com4 = supp[i]; Platform2!Com4; i = i +1 }
goto r12;
fi;

r12:
if
::i < CHNO; atomic { P2_none = supp[i]; Platform2!P2_none; i = i +1 }
goto r13;
fi;
```

```
r13:
if
::i < CHNO; atomic { P2_one = supp[i]; Platform2!P2_one; i = i +1 }
goto r14;
fi;

r14:
if
::i < CHNO; atomic { P2_two = supp[i]; Platform2!P2_two; i = i +1 }
goto r15;
fi;

r15:
if
::i < CHNO; atomic { P2_onetwo = supp[i]; Platform2!P2_onetwo; i = i +1 }
goto ready2;
fi;

ready2:
if
:: Platform1?r; goto reading2;
:: Platform2?r; goto reading22;
:: P?Platform3; goto r20;
fi;

reading2:
if
:: P1_none!r; goto ready2;
:: P1_one!Com3; goto ready2;
fi;

reading22:
if
:: P2_none!r; goto ready2;
:: P2_one!Com1; goto ready2;
fi;

r20:
if
::i < CHNO; atomic { Com5 = supp[i]; Platform3!Com5; i = i +1 }
goto r21;
fi;

r21:
if
::i < CHNO; atomic { Com6 = supp[i]; Platform3!Com6; i = i +1 }
goto r22;
fi;

r22:
if
::i < CHNO; atomic { P3_none = supp[i]; Platform3!P3_none; i = i +1 }
goto r23;
fi;

r23:
if
::i < CHNO; atomic { P3_one = supp[i]; Platform3!P3_one; i = i +1 }
goto r24;
fi;

r24:
if
::i < CHNO; atomic { P3_two = supp[i]; Platform3!P3_two; i = i +1 }
goto r25;
fi;

r25:
if
::i < CHNO; atomic { P3_onetwo = supp[i]; Platform3!P3_onetwo; i = i +1 }
goto ready3;
fi;
```

```
ready3:
if
:: Platform1?r; goto reading31;
:: Platform2?r; goto reading32;
:: Platform3?r; goto reading33;
fi;

reading31:
if
:: P1_none!P1_none; goto ready3;
:: P1_one!Com3; goto ready3;
:: P1_two!Com5; goto ready3;
:: P1_onetwo!Com3; goto reading3123;
fi;

reading3123:
if
:: P1_onetwo!Com5; goto ready3;
fi;

reading32:
if
:: P2_none!P2_none; goto ready3;
:: P2_one!Com1; goto ready3;
:: P2_two!Com6; goto ready3;
:: P2_onetwo!Com1; goto reading3213;
fi;

reading3213:
if
:: P2_onetwo!Com6; goto ready3;
fi;

reading33:
if
:: P3_none!P3_none; goto ready3;
:: P3_one!Com2; goto ready3;
:: P3_two!Com4; goto ready3;
:: P3_onetwo!Com2; goto reading3312;
fi;

reading3312:
if
:: P3_onetwo!Com4; goto ready3;
fi;

}



 init
{ atomic {
chan ch0 = [CHLEN] of {chan};
chan nch0 = [0] of {chan};
chan nch1 = [0] of {chan};
chan nch2 = [0] of {chan};
chan nch3 = [0] of {chan};
chan nch4 = [0] of {chan};
chan nch5 = [0] of {chan};
chan nch6 = [0] of {chan};
chan nch7 = [0] of {chan};
chan nch8 = [0] of {chan};
chan nch9 = [0] of {chan};
chan nch10 = [0] of {chan};
chan nch11 = [0] of {chan};
chan nch12 = [0] of {chan};
chan nch13 = [0] of {chan};
chan nch14 = [0] of {chan};
chan nch15 = [0] of {chan};
chan nch16 = [0] of {chan};
chan nch17 = [0] of {chan};
chan nch18 = [0] of {chan};
```

```
chan nch19 = [0] of {chan};
chan nch20 = [0] of {chan};
chan nch21 = [0] of {chan};
chan nch22 = [0] of {chan};
chan nch23 = [0] of {chan};
chan nch24 = [0] of {chan};
chan nch25 = [0] of {chan};
chan nch26 = [0] of {chan};
chan nch27 = [0] of {chan};
chan nch28 = [0] of {chan};
chan nch29 = [0] of {chan};
chan nch30 = [0] of {chan};
chan nch31 = [0] of {chan};
chan nch32 = [0] of {chan};
chan nch33 = [0] of {chan};
chan nch34 = [0] of {chan};
chan nch35 = [0] of {chan};
chan nch36 = [0] of {chan};
chan nch37 = [0] of {chan};
chan nch38 = [0] of {chan};
chan nch39 = [0] of {chan};
chan nch40 = [0] of {chan};
chan nch41 = [0] of {chan};
chan nch42 = [0] of {chan};
chan nch43 = [0] of {chan};
chan nch44 = [0] of {chan};
chan nch45 = [0] of {chan};
chan nch46 = [0] of {chan};
chan nch47 = [0] of {chan};
chan nch48 = [0] of {chan};
chan nch49 = [0] of {chan};
chan nch50 = [0] of {chan};
chan nch51 = [0] of {chan};
chan nch52 = [0] of {chan};
chan nch53 = [0] of {chan};
chan nch54 = [0] of {chan};
chan nch55 = [0] of {chan};
chan nch56 = [0] of {chan};
chan nch57 = [0] of {chan};
chan nch58 = [0] of {chan};
chan nch59 = [0] of {chan};
chan nch60 = [0] of {chan};
chan nch61 = [0] of {chan};
chan nch62 = [0] of {chan};
chan nch63 = [0] of {chan};

run WView(ch0, nch0, nch1, nch2, nch3, nch4, nch5, nch6, nch7, nch8, nch9, nch10,
nch11, nch12, nch13, nch14, nch15, nch16, nch17, nch18, nch19, nch20, nch21);
run Platform(ch0, nch22, nch23, nch24, nch25, nch26, nch27, nch28, nch29, nch30,
nch31, nch32, nch33, nch34, nch35);
run Platform(ch0, nch36, nch37, nch38, nch39, nch40, nch41, nch42, nch43, nch44,
nch45, nch46, nch47, nch48, nch49);
run Platform(ch0, nch50, nch51, nch52, nch53, nch54, nch55, nch56, nch57, nch58,
nch59, nch60, nch61, nch62, nch63);
} };
```

B.3 Banking

```
/* Generated from file C:\rjw1\PhD\RDTModels\bank2.xml */

#define CHLEN 2
#define CHNO 8

proctype Bank(chan BalCh, v, bal, TfrCh, ClrOut, DRch, a, CRch)
{
int i = 0;
chan supp[CHNO] = [CHLEN] of {chan};

initial:
if
```

```
:: BalCh?v; goto BalReqHeld;
:: TfrCh?v; goto TrfReqHeld;
:: DRch?a; goto initial;
:: CRch?a; goto initial;
fi;

BalReqHeld:
if
::i < CHNO; atomic { bal = supp[i]; v!bal; i = i +1 }
goto initial;
fi;

TrfReqHeld:
if
:: ClrOut!v; goto initial;
fi;

}

proctype Client(chan Bal, q, bal, Tfr, a)
{
int i = 0;
chan supp[CHNO] = [CHLEN] of {chan};

initial:
if
::i < CHNO; atomic { q = supp[i]; Bal!q; i = i +1 }
goto WaitingBal;
::i < CHNO; atomic { a = supp[i]; Tfr!a; i = i +1 }
goto initial;
fi;

WaitingBal:
if
:: q?bal; goto initial;
fi;

}

proctype Clearing(chan In, v, Out1, Out2)
{
initial:
if
:: In?v; goto InsHeld;
fi;

InsHeld:
if
:: Out1!v; goto DRsent;
fi;

DRsent:
if
:: Out2!v; goto initial;
fi;

}



 init
{ atomic {
chan ch0 = [CHLEN] of {chan};
chan ch1 = [CHLEN] of {chan};
chan ch2 = [CHLEN] of {chan};
chan ch3 = [CHLEN] of {chan};
chan ch4 = [CHLEN] of {chan};
chan ch5 = [CHLEN] of {chan};
chan ch6 = [CHLEN] of {chan};
chan nch0 = [0] of {chan};
chan nch1 = [0] of {chan};
chan nch2 = [0] of {chan};
chan nch3 = [0] of {chan};
```

```
chan nch4 = [0] of {chan};
chan nch5 = [0] of {chan};
chan nch6 = [0] of {chan};
chan nch7 = [0] of {chan};
chan nch8 = [0] of {chan};
chan nch9 = [0] of {chan};
chan nch10 = [0] of {chan};
chan nch11 = [0] of {chan};
chan nch12 = [0] of {chan};
chan nch13 = [0] of {chan};
chan nch14 = [0] of {chan};

run Bank(ch0, nch0, nch1, ch1, ch4, ch2, nch2, nch3);
run Bank(ch5, nch4, nch5, ch6, ch4, nch6, nch7, ch3);
run Client(ch0, nch8, nch9, ch1, nch10);
run Clearing(ch4, nch11, ch2, ch3);
run Client(ch5, nch12, nch13, ch6, nch14);
} };
```

B.4 Mobile Phones

```
/* Generated from file C:\Spin\spinstuff\keep\Car\Car1.xml */

#define CHLEN 0
#define CHNO 4

proctype Car(chan talk, switch)
{
chan tmp;
initial:
if
:: talk!talk; goto initial;
:: switch?talk; goto switching;
fi;

switching:
if
:: atomic{switch?tmp; switch = tmp; } goto initial;
fi;

}

proctype BaseA(chan t, x, g, t2, s2, s, a)
{
initial:
if
:: t?x; goto initial;
:: g?t2; goto giving1;
fi;

giving1:
if
:: g?s2; goto giving2;
fi;

giving2:
if
:: s!t2; goto switching;
fi;

switching:
if
:: s!s2; goto inactive;
fi;

inactive:
if
:: a?t; goto waking;
fi;
```

```
waking:
if
:: a?s; goto initial;
fi;

}

proctype BaseI(chan a, t, s, x, g, t2, s2)
{
initial:
if
:: a?t; goto waking;
fi;

waking:
if
:: a?s; goto active;
fi;

active:
if
:: t?x; goto active;
:: g?t2; goto giving1;
fi;

giving1:
if
:: g?s2; goto giving2;
fi;

giving2:
if
:: s!t2; goto switching;
fi;

switching:
if
:: s!s2; goto initial;
fi;

}

proctype Centre(chan give1, talk2, switch2, alert2, give2, talk1, switch1, alert1)
{
int i = 0;
chan supp[CHNO] = [CHLEN] of {chan};

initial:
if
::i < CHNO; atomic { talk2 = supp[i]; give1!talk2; i = i +1 }
goto giving1;
fi;

giving1:
if
::i < CHNO; atomic { switch2 = supp[i]; give1!switch2; i = i +1 }
goto giving11;
fi;

giving11:
if
:: alert2!talk2; goto alertingI1;
fi;

alertingI1:
if
:: alert2!switch2; goto ready1;
fi;

ready1:
if
::i < CHNO; atomic { talk1 = supp[i]; give2!talk1; i = i +1 }
goto giving21;
```

```
fi;

giving21:
if
::i < CHNO; atomic { switch1 = supp[i]; give2!switch1; i = i +1 }
goto giving22;
fi;

giving22:
if
:: alert1!talk1; goto alerting2;
fi;

alerting2:
if
:: alert1!switch1; goto readyA;
fi;

readyA:
if
:: give1!talk2; goto givingA;
fi;

givingA:
if
:: give1!switch2; goto givingB;
fi;

givingB:
if
:: alert2!talk2; goto AlertingA;
fi;

AlertingA:
if
:: alert2!switch2; goto readyB;
fi;

readyB:
if
:: give2!talk1; goto givingX;
fi;

givingX:
if
:: give2!switch1; goto givingY;
fi;

givingY:
if
:: alert1!talk1; goto alertingX;
fi;

alertingX:
if
:: alert1!switch1; goto readyA;
fi;

}



 init
{ atomic {
chan ch0 = [CHLEN] of {chan};
chan ch1 = [CHLEN] of {chan};
chan ch2 = [CHLEN] of {chan};
chan ch3 = [CHLEN] of {chan};
chan ch4 = [CHLEN] of {chan};
chan ch5 = [CHLEN] of {chan};
chan nch0 = [0] of {chan};
chan nch1 = [0] of {chan};
chan nch2 = [0] of {chan};
```

```
chan nch3 = [0] of {chan};
chan nch4 = [0] of {chan};
chan nch5 = [0] of {chan};
chan nch6 = [0] of {chan};
chan nch7 = [0] of {chan};
chan nch8 = [0] of {chan};
chan nch9 = [0] of {chan};
chan nch10 = [0] of {chan};
chan nch11 = [0] of {chan};

run Centre(ch0, nch0, nch1, ch1, ch2, nch2, nch3, ch3);
run BaseA(ch4, nch4, ch0, nch5, nch6, ch5, ch3);
run BaseI(ch1, nch7, nch8, nch9, ch2, nch10, nch11);
run Car(ch4, ch5);
} };
```

Note: This is the raw code generated by the tool before changes to elminiate the problematic identifiers, “switch” and “acitve”

Appendix C The modelling tools[1]

This appendix will be a description of the model generation tool, the execution tool and using the translation tool to generate Promela code for SPIN.

C.1 Initial considerations

Before work could start on building tool support for our new language, a number of decisions needed to be made. The most obvious of these were what (if anything) to bring forward from RolEnact, which programming language to use and how to store data about models.

C.1.1 Carrying work forward from RolEnact

There is a significant investment of effort both in the existing RolEnact/RaDraw code and the ideas and techniques it uses. However, on balance it was decided to keep some of the features of the existing system but not to adopt any of the code. As well as giving more freedom in designing the new system, this decision meant that the new system would not be bound to use the mixture of Visual Basic, C++ and Enact [Henderson] used in the RolEnact tools.

C.1.2 Programming language

The decision not to adopt the existing code gave us a free choice of development tools for the new system.

The RolEnact stepper was programmed using a mixture of Visual Basic and Enact, as was the simulation tool whilst the drawing tool (RaDraw) was built with C++ (Borland C++ Builder). ARE also uses COM to drive its use of Excel for recording results. This mixture of languages came about partly by design: it permitted the various languages to be used for the parts of the system to which they were best suited, but the additional effort associated with integrating the parts cancelled out part of this benefit. It was decided to use a single language/development environment for the new project.

From the many development environments available, the choice was narrowed to one of three: Visual Basic, C++, JAVA. Each has its merits. Although popular in the development community at present, JAVA [Flanagan 1997] was the first to be dismissed. Ironically, a major reason for dismissing JAVA amounted to the fact that it was not used for the previous tools leading to a relative lack of familiarity with the language and its development environment.

---

[1] In the implementation of the RDT tools, the “Send” and “Receive” events have been renamed to “Write” and “Read” respectively.

Having already decided to avoid mixed language development, this left a straight choice between Visual Basic and (Visual) C++ [Henderson 1993]. Both have their merits. C++ is arguably the more powerful language and would have greatly simplified adopting or adapting previous work from RaDraw. However, building applications which have significant user interaction is still much simpler using Visual Basic than C++, even using one of the modern integrated development environments.

The eventual decision was that the advantage of Visual Basic in terms of reducing the time and effort required in developing the user interface related items more than offsets any costs there may be associated with any other areas.

C.1.3 Data Storage

The RolEnact/RaDraw package of programs stores information about models in simple text files using the syntax of the RolEnact language which was initially designed and the input language for RolEnact with the expectation that users would read and write this code themselves either using a text editor (or possibly guided by a dedicated editor). However, RDT does not expect users to read or write code. This means that there is not the same need for the method of storage to be "human readable".

The method eventually selected for storage was XML [Hunter, Cagle et al. 2000]. The compelling advantage of XML is that, should it become necessary, additional structures and data fields may be added to files without impairing the ability of existing programs to read and manipulate them. There is also a Visual Basic implementation of XML which handles much of the work associated with manipulating XML documents. A final advantage to this data format is that, being based on simple text, reading and manipulating XML files by hand is feasible should it become necessary.

Currently, there are three RDT support tools: RDT, RDX and RDT2SPIN. RDT is the model generation tool which provides facilities for creating and viewing models using the RDT language. RDX is a stand alone execution tool with which the modeller can execute a completed model. RDT2SPIN provides an automated translation from RDT into Promela, the input language for the SPIN model checker.

C.2 RDT

C.2.1 RDT: Model creation tool

The model creation tool is called RDT. It uses lessons learned in the development of RaDraw, though it does not re-use any of the code since it is written using Visual Basic instead of C++. The RDT model creation tool offers the user the means to construct models. This task is divided into two distinct parts: describing proc-

esses, assembling processes into an executable model. This division of the task into two sub-tasks is a feature born out of the experience building the RaDraw tool. The purpose the division is to draw a clear distinction between when describing the behaviour of a type of a process (or the process in general) from then describing matters which relate to a particular instance of a process. Experience of the RAD-like representation of RolEnact models suggests that, despite the diagram being of a type in which it is the behaviour of the processes as a class that is being described, users invariably slip into thinking about a mixture of the behaviour of an instance of the process and the behaviour of the process as a class.

When the program is started, the user is presented with a simple window with "File", "Edit" and "View" offered on the menu.

The "File" menu offers the usual types of action, including opening a file, saving a file and closing the programme.

The "Edit" menu offers the user options to create a new event of delete an existing one.

From the "View" menu, the user can open a window showing either a process or a "model" (an assembly of process instances into a executable system).

C.2.1.1 Building processes

The first stage to constructing a complete, executable model is to describe the processes in it. A process is built by describing its events. Selecting "New Event" from the "Edit" menu causes a dialogue box to be displayed. In this box, the user is required to describe the various features of the new event. Whenever possible, the user selects a value for the various attributes by making a selection using the standard mechanism of the windows "combo box". When a new entry needs to be added to a combo box list, there is a button which, when pressed causes the display of a further dialogue box which will illicit suitable details from the user.

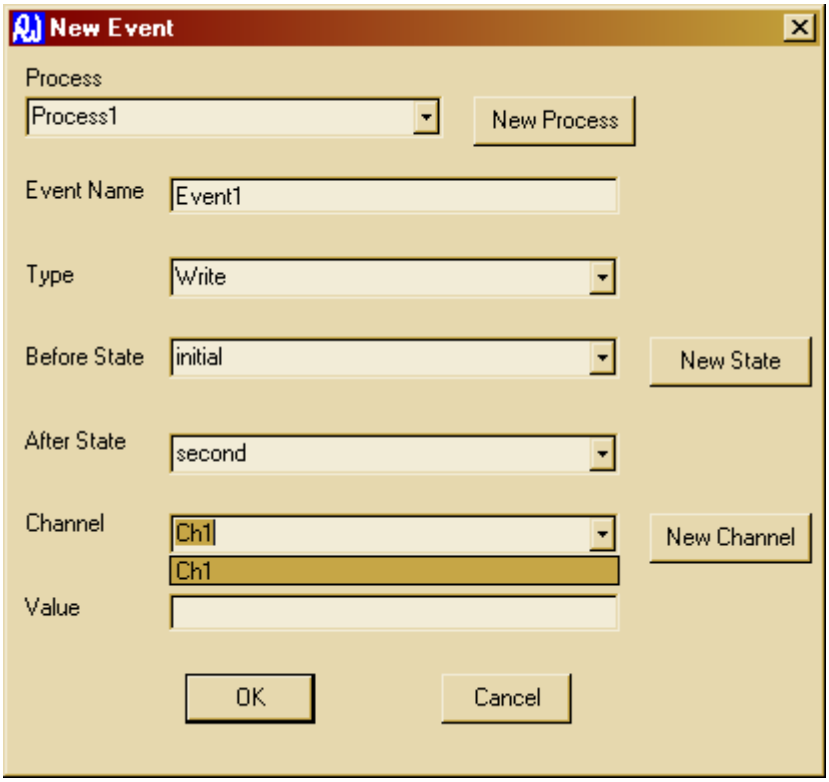


Figure 59: The new event dialogue box showing the user selecting a channel from those available

A complete description of an event comprises the following data:
The name of the process to which this event belongs.
The name of the event.
The type of the event (Read, Write or Create).
The name of the state the process must be in before the event may happen.
The name of the state the process moves to after the event takes place.
The process' name for the channel which will be used by the event.
The process' name for the value to be read from /written to the channel.

Each process is required to start execution in a state known as "initial".

The name of each process type must be unique within any particular file as must be the state names used within a process. However, an event name may be re-used in a process provided the before states for each of the events is different.

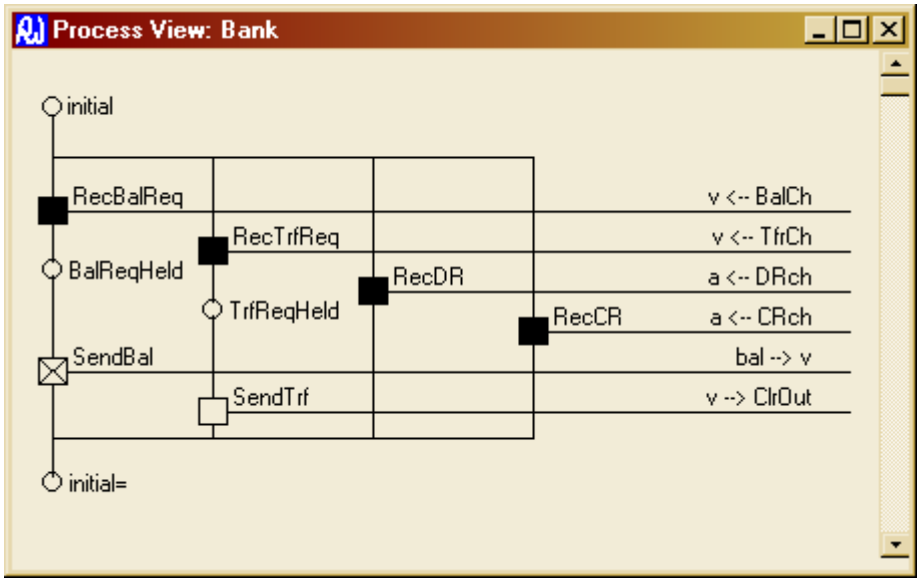


Figure 60: A typical process diagram

At any time after a process's first event has been defined, the user may user the "View" menu and select a process to open a window in which a diagram of the process will be drawn. The tool tries to keep these diagrams up to date as events are added to and removed from the model.

C.2.1.2 Building models

Once processes have been defined, the user is able to progress to the second part of the construction activity and assemble some number of these processes into a model for execution. To do this, the modeller uses the menu system to open a view of a "model". This action opens another new window with which to view and manipulate a particular collection of process instances. Unlike the process view window, the model view window has a menu similar to that in the main application window. The file representation used permits the modeller to create more than one assembly from a single set of process descriptions. These descriptions are referred to as "models" by the tool. The "model" menu offers commands for selecting an existing model to work with as well as creating and deleting models.

There are two parts to the task of assembling instances of processes into a model for execution: adding process instances to the model and making the required (initial) connections between those instances.

To add a process instance, the user selects "Instance" from the "New" menu. The user then selects the type of process required and inputs the name which will be used for this particular process instance. The new process instance is then drawn on the left-hand side of the window as a box labelled with the name of this instance and its type. The process' channels appear along the right-hand side of the

box as horizontal lines with a "blob" at the end. These channels are each labelled with the name used by the process for that channel.

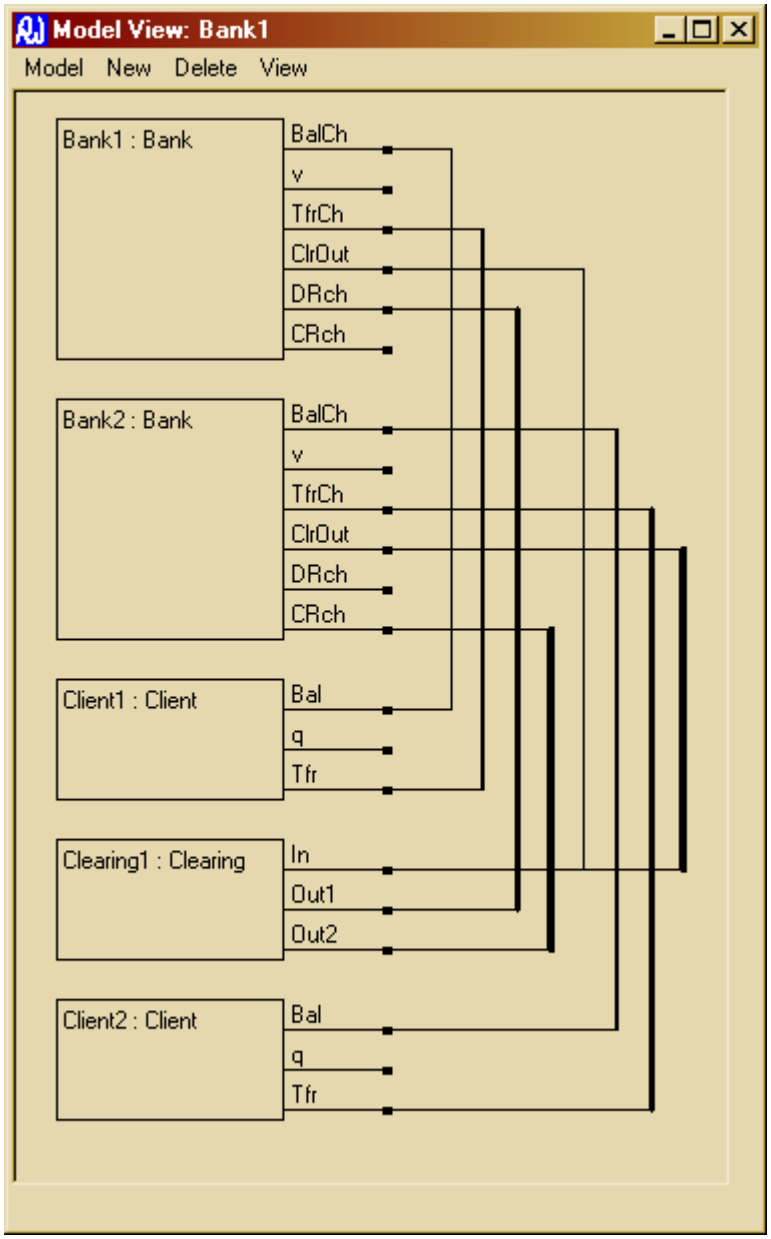


Figure 61: An example "model" diagram

Once the required instances have been added to the model, the user is able to add connections between them using the "New", "Connection" command. This command uses a dialogue box to illicit the required information about which channel of which process instance is at the end of each connection.

Process instances and connections between them may be removed from the model using appropriate command from the "Delete" menu.

C.3 RDX

RDX is the model execution tool. It takes the XML file of a model generated by RDT and executes the model. At startup, the main application window is empty. When a file is loaded into the tool, a sub-window is created for each of the process instances in the model. A window is also created for each of the channels required to make the defined connections between these process instances.

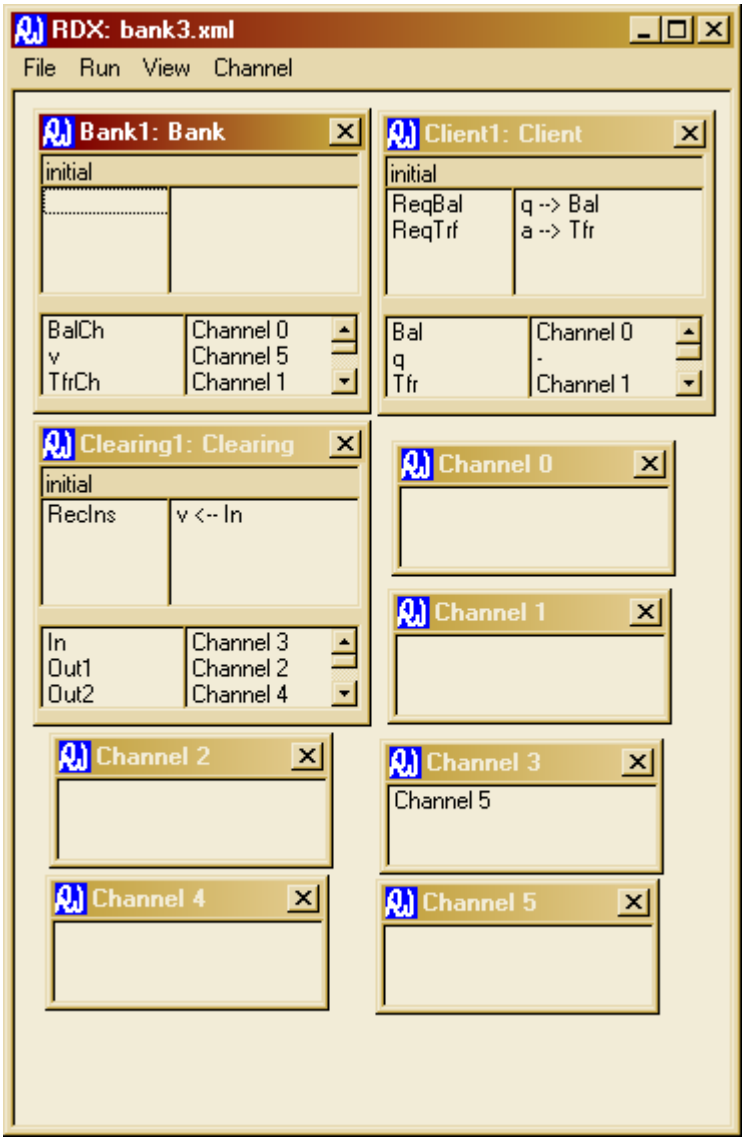


Figure 62: A model during execution

Each process lists the events which it is currently able to initiate. The user steps through the model my "double-clicking" on the event they wish to cause to occur. After each event, the list of available events is re-calculated for each process.

C.3.1 Process

A process window which is similar in style to a RolEnact stepper process window is created for each process instance in the model. It shows the name of the instance and its type in its caption. Below the caption, the name of the current state of the process is shown. The process window has two further pairs of data areas. In the upper pair is a list of the events which are presently available for the instance to action. The name of the event is given in the left-hand list and the name of the channel and the value to be used are shown in the right-hand list. The direction of the communication is indicated by the direction of the arrow. In the final pair of boxes, the names known to the process instance are listed with the instance's local name on the left and the "global" name of the channel (if any) to which the name refers on the right. This "global" name is the name by which the channel is addressed if the process is to read from or write to the channel. Internally, processes refer to channels by their own (internal) names. When the need arises for a process to address a channel, the required "global" name is looked up in the table formed by the final pair of boxes are the lowest part of a process' window.

The user causes the tool to make a step in the execution of the model by selecting an event and "double clicking" on its name to make it happen.

Where the event is a "read", a value is removed from the appropriate channel and associated with the name specified in the description of the event (to the right of its name). The appropriate channel is identified by looking up the name of the channel in the list of name/channel pairs in the lower part of the process window. The value is then associated with the appropriate name in the lower part of the process window.

In the case of a "write" event, the sequence of events is similar, except that the value is looked up before the event takes place and is placed into the appropriate channel which is found in the same manner.

Where the event is a "create", the effect of the event is similar to a "write" with the variation that, before the event takes place, a new channel (and corresponding window) are created. The name of this new channel is then associated with the appropriate name in the writing channel's collection. The event then proceeds as if it were a "write".

C.3.2 Channel

Although the communications of the RDT language are inspired by the pi-calculus, it is more flexible in that the RDT modeller has the option to make their channels asynchronous. Since there is no need for it to be specified earlier, this election to use synchronous (zero length channels) or asynchronous communications (non-zero length channels) is made at the time that the model is executed. This facility for channels to be buffered brings with it a requirement for channels to have an existence independent of the process instances which they

connect: the execution tool has to be able to show values written into a channel, but not yet read out if the modeller is to be able to see and understand what is happening. For this reason, in the execution the channels which are anonymous in the model generation tool are each named and given an explicit presence in the window of the execution tool.

A window is also created for each channel in the model. The initial collection of channels in a model is created by RDX as the model file is being loaded using the information about the connections between the various process instances in the model. Each time a process instance actions a "create" event a new channel and corresponding window are created.

Each of these channels has a unique identity. As the system runs, it keeps track of how many channels which have been created and uses this number to allocate a unique name to each of them. A channel's name is shown in the title bar of its window in the form, "ChannelX", where X is the number of this channel. Where a process "knows" of a channel it is this name that appears in the process's window as the "global" name of the channel.

When a process writes a value into a channel, the appropriate "global" name is added to the list of data items held in the channel. A process reading from a channel receives one of these names. In the present implementation, channels operate on a "first in, first out" basis.

The major part of a channel's window is devoted to a list which shows the values (if any) which have been written into the channel, but not yet read out. Apart from being able to see the contents of a channel, the user has no interaction with the channel windows.

C.3.3 Making connections

Loading a model into the stepper proceeds in several steps. First, a process window is created for each process instance in the model. Once this is completed, channels (and their windows) are created for the connections between those process instances as described in the file. As the model is constructed, each connection has two ends each of which is applied to a "port" on a process. When a connection joins two ports which are not involved with any other connection, a channel window is opened for that connection and the name of that channel is associated with the appropriate "local" name is the processes concerned. The connection is established by virtue of both process instances having access to the same channel (each using their own, local name for it). Where a connection is more complex than "pair-wise", the connection is established in the same manner, but as further ports are connected after the first pair, no new channel is required and the connection is established by associating the name of the existing channel with the new port.

Making a connection between a pair of ports where both are already connected to others is problematic. What would be required is for all of the ports concerned to

be associated with the just one channel. So, one channel is not needed (and has to be deleted). All references to the channel which is not to be used need to be replaced by references to the channel which is to be retained. Doing this thoroughly would require a full search of all "ports" on all process instances to guarantee that there are no remaindered references to the deleted channel. In its present form, the tool expects the modeller wishing to connect more than two ports together to add them one at a time to a single collection thus enabling the tool to connect ports as required without needing to address this particular issue. Nevertheless, for each connection the tool does check to see if both “ends” are already connected and issues a warning should this be the case.

C.3.4 Styles of channels

The user is able to vary the behaviour of channels at the time that a model is executed by setting their length. When the tool is started, the length of channels is set to a default value. For any non-zero value of the channel length, the behaviour of channels is as follows: Where a process wishes to write a value into a channel it will permitted to do so, provided the channel is not currently full. Where a channel is full, no additional writes may be made into that channel until at least one value has been removed by a read being performed on it. A process is always able to read from a channel provided it is not empty. This gives an asynchronous style of communication where, subject to space and values being available, events which write and read channels occur independently.

The user may also set the channel length to zero which causes the model behaviour to change. Now no process is able to read from a channel as, since they are unable to hold even a single item of data, no channel can have a value available to be read. Also, no process is able to write (or perform a create and write) a value into a channel unless there exists another process which is in a state where it would be able to read a value from the channel, should it contain one. As soon as a process does write a value into a channel, the model enters an intermediate state in which the value is held by the channel even though the channel length is zero. The tool now forces the user to execute an event which restores the model to an acceptable state by removing this value from the channel. It does this by preventing any event to occur which does not read from the "overfull" channel in which the value is held. To aid the user trying to follow the sequence of execution, the process responsible for writing the value into the channel is identified by a change of its background colour. This process is also prevented from carrying out the "read" event because we wish to simulate the effect of a synchronous communication and, permitting it to read the written value back would amount to the process carrying out both the write and the read at the same time - and no process is able to carry out two actions simultaneously. Once the read event has take place the model is again in a "reasonable" state and the cycle repeats. Thus write and read events take place in pairs in the style of synchronous communication.

As execution of the model proceeds, the processes may carry out "Create" events. When one of these occurs, a new channel and a window for it are created. The name of this new channel is written into the (existing) channel specified by the

definition of the event and the name of this new channel is associated with the appropriate local name in the "creating" process.

Since the only ways for a process to learn the name of a channel are to receive its name in a "Read" event or create it and these actions cause an existing association between a process "local" name and another channel to be overwritten, a process may lose all references it has to a channel. Should this happen throughout the model, the channel affected is lost and there is no mechanism by which any of the processes will be able to re-discover it.

Where a model includes behaviours in which channels continue to be lost, the display can become cluttered with channels which can play no further part in execution. The Auto Hide option from the "Channel" menu causes these channels to be hidden. They can be brought back into view by selecting the "Show all" from the "Channel" menu. (The modeller may wish to see these channels because despite being lost to all of the processing of the executing model, they need not be empty.)

C.4 RDT2SPIN

The final tool in the collection is RDT2SPIN. The tool provides an automated “source to source” translation from an RDT model into Promela, the input language of the SPIN model checker.

Input to the tool is provided as a file of XML generated by the RDT tool. Its output is written as text to a file named by the user. This tool has a simple interface which elicits filenames for the input and output files from the user. The interface also shows the current setting for the length of channels which the modeller desires. Finally, where the model contains a process instance which is able to perform an event of type “Create”, the modeller is required to select a number for the “Channel stock” to be used.

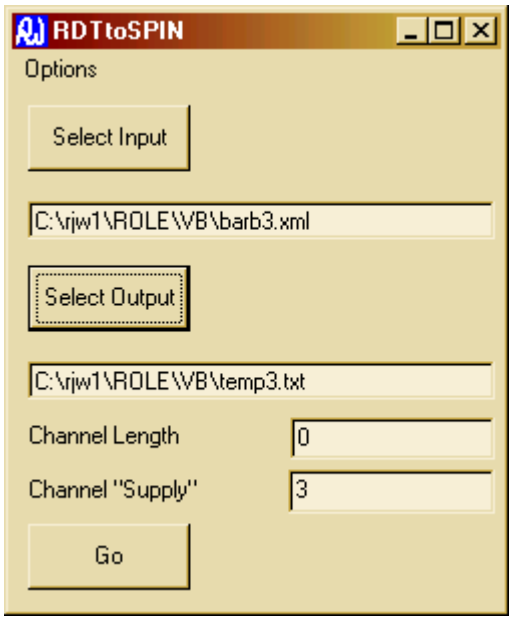


Figure 63: User interface of RDTtoSPIN

Chapter 7 outlines the Promela language, describes the mechanism of the conversion from RDT to Promela and discusses issues which arise in the translation.

Appendix D Source code of the tools[1]

The tools were developed using Microsoft Visual Basic version 6.0 (SP5).

Applications in Visual Basic are constructed using an integrated development environment. Each of the "windows" which an application is able to present on the screen has its own file. These files are constructed partly by drawing using visual tools with the remainder of the code typed into an editor using a variant of the well known BASIC programming language.

A Visual Basic program does not have a start point and flow of control in the style of a conventional programming language. Instead, the system defines a collection of functions which are called by the system in response to particular events. For example, when an application is started its main form is loaded (and usually appears on screen) and a function named, "Form_Load()" is called. The application responds to menu selections and other user interactions (button presses, mouse operations) by calling appropriate functions.

An application may also include code in other files ("modules") which do include visible elements.

For each of the applications, the names of the files they use are listed followed by the code itself. For each "form" file, a view of the form is presented followed by the code associated with the form. Buttons and other elements of the forms with which the user interacts are labelled with the names of the functions which they invoke. Where appropriate, a diagram indicates the structure of each form's menu and the names of the functions called in response to menu selections.

Of the remaining system-called functions, they have standard names (such as "Form_Load()" mentioned above) which are reasonably self-explanatory.

D.1 RDT to SPIN

This application takes a description in XML of a model (created using the RDT model generation tool) and converts it into Promela, the input language for the SPIN model checker. It has two forms:

ToSpin.frm
PickModelFrm.frm

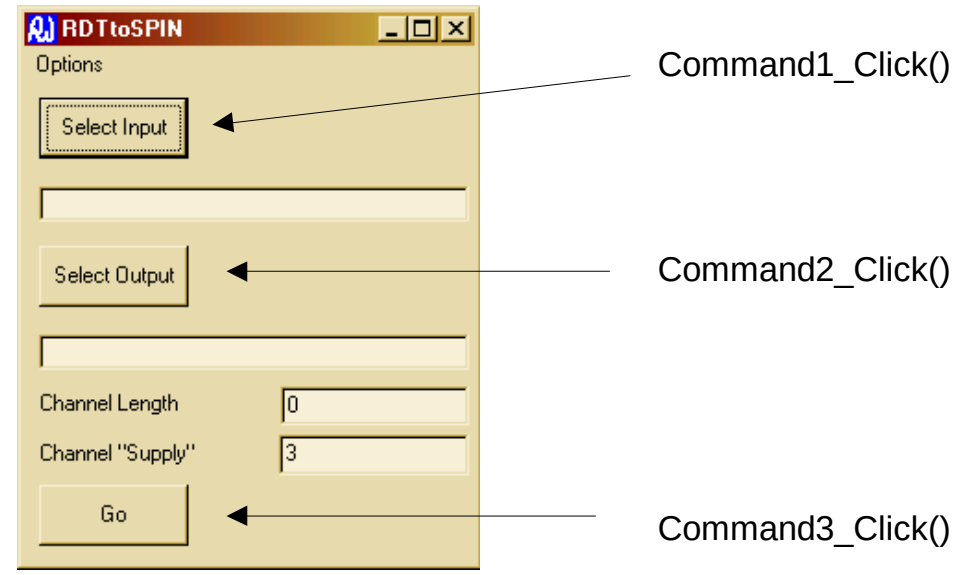
[1] In the implementation of the RDT tools, the "Send" and "Receive" events have been renamed to "Write" and "Read" respectively.

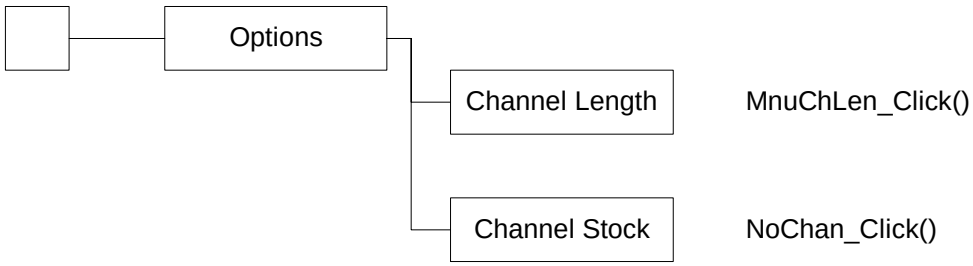
ToSpin.frm is the main form of the application. PickModelFrm.frm is used as a "pop-up" where the input file contains more than one model to elicit which of the models they application should convert into Promela.

In addition to the two forms, it has a single module in which a data structure for holding details of process states is described:

States.cls

D.1.1 ToSpin.frm






```
Option Explicit
Public ChLen As Integer
Public NChan As Integer
Public doc As DOMDocument
Private fileSysObject As Object
```

```
Private Type proc_type
Name As String 'The name of a type of process
ports() As String 'Collection of all channel names known by this type of process
End Type

Private Type inst_type
Name As String 'Name of the process instance
type As String
ports() As String 'Name used by THIS process for the channel
channels() As String 'Global name of the channel
End Type


Private ptypes() As proc_type
Private pinsts() As inst_type


Private Sub Command1_Click()
CommonDialog1.Filter = "All Files (*.*)|*.*|XML Files(*.xml)|*.xml"
CommonDialog1.FilterIndex = 2
CommonDialog1.ShowOpen
If CommonDialog1.FileName <> "" Then Text1.Text = CommonDialog1.FileName
End Sub

Private Sub Command2_Click()
Dim arr As Variant

CommonDialog2.Filter = "All Files (*.*)|*.*|TXT Files(*.txt)|*.txt"
CommonDialog2.FilterIndex = 2
'If Text1.Text <> "" Then
'    arr = Split(Text1.Text, ".")
'    'CommonDialog2.FileName = arr(0) & ".txt"
'End If
CommonDialog2.ShowOpen
If CommonDialog2.FileName <> "" Then Text2.Text = CommonDialog2.FileName
End Sub

Private Sub Command3_Click()
Dim txtStream As Object
Dim el As IXMLDOMElement
Dim el2 As IXMLDOMElement
Dim tmpel As IXMLDOMElement
Dim nodes As IXMLDOMNodeList
Dim nodes2 As IXMLDOMNodeList
Dim i As Integer
Dim j As Integer
Dim k As Integer
Dim l As Integer
Dim s As String
Dim s2 As String
Dim s3 As String
Dim end1proc As Integer
Dim end1port As Integer
Dim end2proc As Integer
Dim end2port As Integer
Dim chs As Integer  'How many channels we have used so far

Dim s_list As New State_thing

Call s_list.Reset

chs = 0
If Text1.Text = "" Then
    Call MsgBox("Select a file to process")
    Exit Sub
End If

If Text2.Text = "" Then
    Call MsgBox("Select output filename")
    Exit Sub
End If
```

```
'Read the input file into a structure and write some Promela!!!!
Set txtStream = fileSysObject.OpenTextFile(CommonDialog1.FileName, 1)
Set doc = New DOMDocument
doc.loadXML (txtStream.ReadAll)
txtStream.Close

'If there is more than one model in the file, get the user to pick one...
PickModelFrm.Show (vbModal)
'Call MsgBox(PickModelFrm.Selection)
If PickModelFrm.Selection = "" Then Exit Sub


'Collect some information about the channels known to a process
'For each processs type, find all of the names of channels it knows...
ReDim ptypes(10)

Set nodes = doc.getElementsByTagName("Process")
i = 0
While i < nodes.length 'for each process
    If i + 1 > UBound(ptypes) Then ReDim Preserve ptypes(UBound(ptypes) + 10)
    Set el = nodes.Item(i)
    Set nodes2 = el.getElementsByTagName("Event")
    ReDim ptypes(i).ports(10)


    ptypes(i).Name = el.getAttribute("Name")

    j = 0
    While j < nodes2.length 'for each event of the process

        'add the names of the channels in this event to the list (if not already
there)
        If j + 3 > UBound(ptypes(i).ports) Then
            ReDim Preserve ptypes(i).ports(UBound(ptypes(i).ports) + 10)
        End If

        'look for the channel name in the existing list
        k = 0
        Set tmpel = nodes2.Item(j)

 ''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
        'Add the names of its before and after states to the "thing"
        Call s_list.AddState(tmpel.getAttribute("Before"), ptypes(i).Name)
        Call s_list.AddState(tmpel.getAttribute("After"), ptypes(i).Name)
 ''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''

        s = tmpel.getAttribute("Channel")
        While ptypes(i).ports(k) <> s And ptypes(i).ports(k) <> ""  'k <
UBound(ptypes(i).ports)
            Set tmpel = nodes2.Item(j)
            s = tmpel.getAttribute("Channel")
            k = k + 1
        Wend
        ptypes(i).ports(k) = s  'Doesn't matter if we found it - just overwrite
with the same string
        'repeat for the value...
        k = 0
        Set tmpel = nodes2.Item(j)
        s = tmpel.getAttribute("Value")
        While ptypes(i).ports(k) <> s And ptypes(i).ports(k) <> ""  'k <
UBound(ptypes(i).ports)
            Set tmpel = nodes2.Item(j)
            s = tmpel.getAttribute("Value")
            k = k + 1
        Wend
        ptypes(i).ports(k) = s  'Doesn't matter if we found it - just overwrite
with the same string

        j = j + 1
    Wend
    i = i + 1
Wend  'Now ptypes contains information about what names processes know
```

```
'Now find the description of the particular model we should create
Set nodes = doc.getElementsByTagName("Instance")
i = 0
Set el = nodes(i)
s = el.getAttribute("Name")
While s <> PickModelFrm.Selection And i < nodes.length
    i = i + 1
    Set el = nodes(i)
    s = el.getAttribute("Name")
Wend

Set el = nodes(i) 'Now el is the element in doc that refers to our instance


'Create an instance in pinsts for each process instance...
Set nodes = el.getElementsByTagName("ProcInstance")
ReDim pinsts(nodes.length + 2)
i = 0
While i < nodes.length
    Set el2 = nodes.Item(i)
    pinsts(i).Name = el2.getAttribute("Name")
    pinsts(i).type = el2.getAttribute("Type")
    'Find the type and add the names of the channels it uses...
    j = 0
    While j < UBound(ptypes) And ptypes(j).Name <> pinsts(i).type
        j = j + 1
    Wend
    If ptypes(j).Name <> pinsts(i).type Then
        Call MsgBox("Error finding process type, "" " & pinsts(i).type)
        Exit Sub 'No point in proceeding further
    End If

    ReDim pinsts(i).ports(UBound(ptypes(j).ports))
    ReDim pinsts(i).channels(UBound(ptypes(j).ports))
    k = 0
    While k < UBound(ptypes(j).ports)
        pinsts(i).ports(k) = ptypes(j).ports(k)
        k = k + 1
    Wend
    i = i + 1
Wend

Set nodes = el.getElementsByTagName("Connection")
i = 0
While i < nodes.length
    Set el2 = nodes.Item(i)
    Set nodes2 = el2.getElementsByTagName("End")
    If nodes2.length <> 2 Then
        Call MsgBox("Connection with wrong number of ends!")
        Exit Sub
    End If

    'Find the where the two ends are in the pinsts and "do the connection"
    Set tmpel = nodes2.Item(0)
    s = tmpel.getAttribute("ProcInstance")
    end1proc = 0
    While pinsts(end1proc).Name <> s
        end1proc = end1proc + 1
    Wend

    s = tmpel.getAttribute("Channel")
    end1port = 0
    While pinsts(end1proc).ports(end1port) <> s
        end1port = end1port + 1
    Wend

    Set tmpel = nodes2.Item(1)
    s = tmpel.getAttribute("ProcInstance")
    end2proc = 0
    While pinsts(end2proc).Name <> s
        end2proc = end2proc + 1
    Wend
```

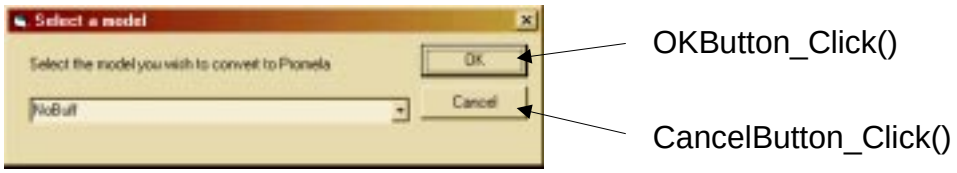
```
    s = tmpel.getAttribute("Channel")
    end2port = 0
    While pinsts(end2proc).ports(end2port) <> s
        end2port = end2port + 1
    Wend

    'Call MsgBox(end1proc & end1port & end2proc & end2port)
    'Here need to code up the connections.......
    '4 possibilities: neither port is connected to anything,
    '                 just one is connected to a channel
    '                 both are already connected to something

    If pinsts(end1proc).channels(end1port) <> "" And
pinsts(end2proc).channels(end2port) <> "" Then
    'Both already connected: bit of a problem... have to discard a channel, fix up
existing connections, etc
    Call MsgBox("Not able to connect channels properly")
    End If

    If pinsts(end1proc).channels(end1port) = "" And
pinsts(end2proc).channels(end2port) = "" Then
    'Neither end is connected to anything yet: add a new channel and "connect"
both ends to it
        pinsts(end1proc).channels(end1port) = "ch" & chs
        pinsts(end2proc).channels(end2port) = "ch" & chs
        chs = chs + 1
    End If

    If pinsts(end1proc).channels(end1port) <> "" And
pinsts(end2proc).channels(end2port) = "" Then
    'First end is connected, second is not: "connect" second end to the channel
        pinsts(end2proc).channels(end2port) = pinsts(end1proc).channels(end1port)
    End If

    If pinsts(end1proc).channels(end1port) = "" And
pinsts(end2proc).channels(end2port) <> "" Then
    'Second end is connected, first is not: "connect" first end to channel
        pinsts(end1proc).channels(end1port) = pinsts(end2proc).channels(end2port)
    End If
    i = i + 1
Wend

'Now write out to the file...
Set txtStream = fileSysObject.OpenTextFile(Text2.Text, 2, True)
txtStream.write "/* Generated from file " & Text1.Text & " */" & vbNewLine &
vbNewLine

txtStream.write "#define CHLEN " & ChLen & vbNewLine
txtStream.write "#define CHNO " & NChan & vbNewLine & vbNewLine

'First the process types
Set nodes = doc.getElementsByTagName("Process")
i = 0
While i < nodes.length 'for each process: write the start line, write each event,
write the end
    Set el = nodes(i)
    txtStream.write ("proctype " & el.getAttribute("Name") & "(chan ")
    j = 0
    While j < UBound(ptypes) And ptypes(j).Name <> el.getAttribute("Name") 'find
the process in the array of types
        j = j + 1
    Wend
    k = 0
    While k < UBound(ptypes(j).ports) And ptypes(j).ports(k) <> ""
        If k <> 0 Then txtStream.write (", ")
        txtStream.write (ptypes(j).ports(k))
        k = k + 1
    Wend
    txtStream.write (")" & vbNewLine & "{" & vbNewLine)

    Set nodes2 = el.getElementsByTagName("Event")
```

```
'''''' Before the events, run though the events looking for a create...
    'If found, declare a collection of channels that this process may use for
"creations"
    j = 0
    Do While j < nodes2.length
        Set el2 = nodes2.Item(j)
        If el2.getAttribute("Type") = "Create" Then
            txtStream.write ("int i = 0; " & vbNewLine)
            txtStream.write ("chan supp[CHNO] = [CHLEN] of {chan}; " & vbNewLine &
vbNewLine)
            Exit Do
        End If
        j = j + 1
    Loop
''''''

'''''' Before the events, run though the events again looking for a read where the
name of the channel becomes the value read ...
    'If found, declare a channel for temporary storage
    j = 0
    Do While j < nodes2.length
        Set el2 = nodes2.Item(j)
        If el2.getAttribute("Type") = "Read" And el2.getAttribute("Channel") =
el2.getAttribute("Value") Then
            txtStream.write ("chan tmp;" & vbNewLine)
            Exit Do
        End If
        j = j + 1
    Loop
''''''

'  Start of main loop writing process descriptions
    j = 0
    While j < nodes2.length
        Set el2 = nodes2.Item(j)
        If el2.getAttribute("Name") <> "" Then
            txtStream.write (el2.getAttribute("Before") & ":" & vbNewLine)
            txtStream.write ("if" & vbNewLine)
''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
''''''''''''
    'Show this state as one with a label in the code
        Call s_list.MarkUsed(el2.getAttribute("Before"), el.getAttribute("Name"))
''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
''''''''''''
        s2 = el2.getAttribute("Before")
        k = j
        While k < nodes2.length
                Set el2 = nodes2.Item(k)
                If el2.getAttribute("Before") = s2 Then
                If el2.getAttribute("Type") = "Write" Then txtStream.write (":: "
& el2.getAttribute("Channel") & "!" & el2.getAttribute("Value") & "; ")
                If el2.getAttribute("Type") = "Read" Then
                    If el2.getAttribute("Channel") = el2.getAttribute("Value")
Then
                    txtStream.write (":: atomic{" & el2.getAttribute("Channel") &
"?tmp; " & el2.getAttribute("Value") & " = tmp; } ")
                    Else
                    txtStream.write (":: " & el2.getAttribute("Channel") & "?" &
el2.getAttribute("Value") & "; ")
                    End If
                End If
'////////////////////////////////////////////////
                If el2.getAttribute("Type") = "Create" Then
                    txtStream.write ("::i < CHNO; atomic { " &
el2.getAttribute("Value") & " = supp[i]; " & el2.getAttribute("Channel") & "!" &
el2.getAttribute("Value") & "; i = i +1 } " & vbNewLine)
                End If
'////////////////////////////////////////////////

                'Remove any trailing "=" from the new state name...
                l = InStr(el2.getAttribute("After"), "=")
                If l > 0 Then s = Left(el2.getAttribute("After"), l - 1) Else s =
el2.getAttribute("After")
```

```
                txtStream.write ("goto " & s & ";" & vbNewLine)
                Call el2.setAttribute("Name", "")
            End If
        k = k + 1
        Wend
        txtStream.write ("fi;" & vbNewLine & vbNewLine)
        End If
        j = j + 1
    Wend

''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
'Add labels for the states the process never leaves
    s3 = s_list.MkUnusedLabelString(el.getAttribute("Name"))
    If Len(s3) > 0 Then
        txtStream.write (s3)
    End If
''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''

    txtStream.write ("}" & vbNewLine & vbNewLine)
    i = i + 1
Wend

'Then the init process (which runs the various instances of the defined processes)

txtStream.write (vbNewLine & vbNewLine & " init" & vbNewLine & "{ atomic {" &
vbNewLine)
'Declare channels
i = 0
While i < chs
    txtStream.write ("chan ch" & i & " = [CHLEN] of {chan};" & vbNewLine)
    i = i + 1
Wend

i = 0
k = 0
While i < UBound(pinsts) And pinsts(i).Name <> ""
    j = 0
    While pinsts(i).ports(j) <> "" And j < UBound(pinsts(i).channels)
        If pinsts(i).channels(j) = "" Then k = k + 1
        j = j + 1
    Wend
    i = i + 1
Wend

i = 0
While i < k
    'txtStream.write ("chan nch" & i & ";" & vbNewLine) '21/8/01 Changed to
eliminate "Receive/send on uninitialised channel" errors from Spin
    txtStream.write ("chan nch" & i & " = [0] of {chan};" & vbNewLine)
    i = i + 1
Wend

txtStream.write (vbNewLine)
i = 0
k = 0
While i < UBound(pinsts) And pinsts(i).Name <> ""
    txtStream.write ("run " & pinsts(i).type & "(")
    txtStream.write (pinsts(i).channels(0))
    j = 1
    While pinsts(i).ports(j) <> "" And j < UBound(pinsts(i).channels)
        If pinsts(i).channels(j) = "" Then
            txtStream.write (", nch" & k)
            k = k + 1
        Else
            txtStream.write (", " & pinsts(i).channels(j))
        End If
        j = j + 1
    Wend
    txtStream.write (");" & vbNewLine)
    i = i + 1
Wend
```

```
txtStream.write ("} };")

txtStream.Close
Call MsgBox("Done! No of channels = " & chs)
End Sub

Private Sub Form_Load()
Set fileSysObject = CreateObject("Scripting.FileSystemObject")

ChLen = 0
Text3.Text = ChLen
NChan = 3
Text4.Text = NChan
End Sub

Private Sub MnuChLen_Click()
Dim str As String

str = InputBox("Enter the required length of channels" & vbNewLine & "Present
length: " & ChLen, "Channel Length", 0)

If str = "" Then Exit Sub 'Assume cancel has been used

If Val(str) < 0 Or (Val(str) = 0 And str <> "0") Then
    Call MsgBox(str & " is not a valid value, Channel length will be set to 0",
vbOKOnly, "Error")
    ChLen = 0
Else
    ChLen = Val(str)
    Text3.Text = str
End If
End Sub

Private Sub NoChan_Click()
Dim str As String

str = InputBox("Enter the required channels ""Stock size"" " & vbNewLine &
"Present size: " & NChan, "Channel Stock Size", 3)

If str = "" Then Exit Sub 'Assume cancel has been used

If Val(str) < 0 Or (Val(str) = 0 And str <> "0") Then
    Call MsgBox(str & " is not a valid value, Channel Stock size will be set to
3", vbOKOnly, "Error")
    NChan = 3
Else
    NChan = Val(str)
    Text4.Text = str
End If
End Sub

Private Sub Text3_Change()
ChLen = Val(Text3.Text)
End Sub

Private Sub Text4_Change()
NChan = Val(Text4.Text)
End Sub
```

D.1.2 PickModel.frm




```
Option Explicit
Public Selection As String

Private Sub CancelButton_Click()
Selection = ""
PickModelFrm.Hide
End Sub

Private Sub Form_Activate()
Dim i As Integer
Dim nodes As IXMLDOMNodeList
Dim el As IXMLDOMElement

Combo1.Clear
Set nodes = ToSpin.doc.getElementsByTagName("Model")
Set el = nodes.Item(0)
Set nodes = el.getElementsByTagName("Instance")

nodes.Reset
Set el = nodes.nextNode
i = 0
While i < nodes.length
    Combo1.AddItem (el.getAttribute("Name"))
    Set el = nodes.nextNode
    i = i + 1
Wend
If Combo1.ListCount > 0 Then Combo1.ListIndex = 0
If Combo1.ListCount = 1 Then
    Selection = Combo1.List(0)
    PickModelFrm.Hide
End If
End Sub

Private Sub OKButton_Click()
Selection = Combo1.List(Combo1.ListIndex)
PickModelFrm.Hide
End Sub
```

D.1.3 States.cls

```
Option Explicit

Private states() As String
Private p_names() As String
Private used() As Boolean

Private Sub Class_Initialize()
ReDim states(40)
ReDim p_names(40)
ReDim used(40)
End Sub

Public Sub AddState(Name As String, PName As String)
'See if its already there, if not, add it
Dim SName As String
Dim i As Integer
Dim l As Integer
```

```
'Remove trailing = from state name
l = InStr(Name, "=")
If l > 0 Then SName = Left(Name, l - 1) Else SName = Name

i = 0
Do While i < UBound(states) And states(i) <> ""
    If states(i) = SName And p_names(i) = PName Then Exit Sub
    i = i + 1
Loop
If i + 2 > UBound(states) Then
    ReDim Preserve states(UBound(states) + 20)
    ReDim Preserve used(UBound(used) + 20)
    ReDim Preserve p_names(UBound(p_names) + 20)
End If
states(i) = SName
p_names(i) = PName
used(i) = False
End Sub

Public Sub Reset()
ReDim states(40)
ReDim used(40)
ReDim p_names(40)
End Sub

Public Sub MarkUsed(SName As String, PName As String)
Dim i As Integer
i = 0

Do While i < UBound(states) And states(i) <> ""
    If states(i) = SName And p_names(i) = PName Then used(i) = True
    i = i + 1
Loop
End Sub

Public Function MkUnusedLabelString(PName As String) As String
Dim s As String
Dim i As Integer
i = 0

Do While i < UBound(states) And states(i) <> ""
    If used(i) = False And p_names(i) = PName Then
        s = s & states(i) & ": skip" & vbNewLine
    End If
    i = i + 1
Loop

'Call MsgBox(s)
MkUnusedLabelString = s
End Function
```

D.2 RDX

This application loads an XML file describing a model (normally created using the RDT model generation tool and permits the user to execute this model interactively. It has five forms:

ChannelFrm.frm
MDIForm1.frm
ProcessFrm.frm
SelChanLenFrm.frm
SelModelFrm.frm

MDIForm1.frm is the main form of the application. This is the form which appears when the application is started. It acts as a container for the windows representing the processes and channels of the model being executed. ChannelFrm.frm and ProcessFrm.frm describe the “mini-windows” used to represent channels and processes. SelChanLenFrm.frm and SelModelFrm.frm are used as “pop-ups” to elicit information from the user regarding the required length for channels during execution and, in the event that the input file contains more than one model, which model to execute.

It also has a single module in which a data structure for holding details of process states is described:

RDXModule.bas

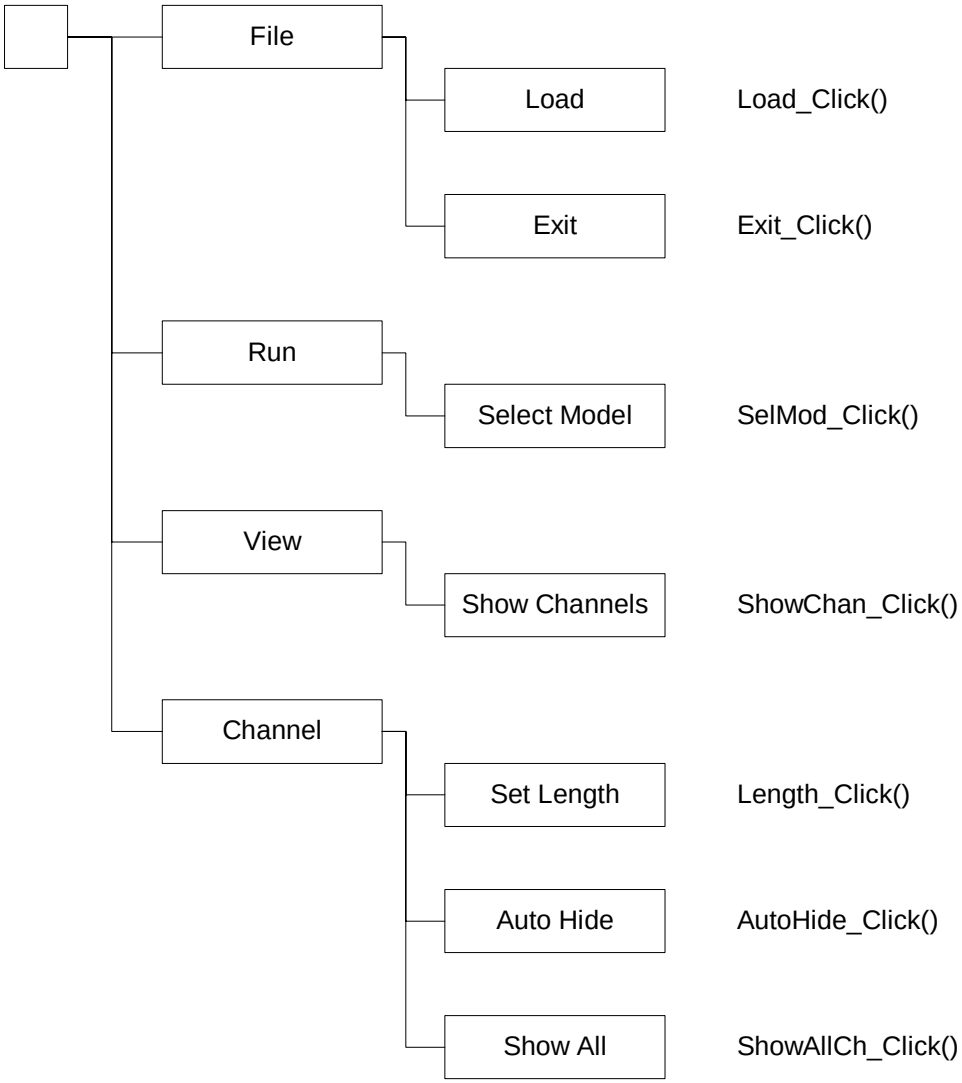
D.2.1 ChannelFrm.frm

This form is used to show a channel in the main MDI window of the application. It has no interaction with the user beyond showing the values, if any which have been written into the channel awaiting being read out.


```
Option Explicit
Private vals() As ChannelFrm
Private nextin As Integer
Private nextout As Integer

Public Sub WriteVal(v As ChannelFrm)
'Add a value to the list
List1.AddItem (v.Caption)
If UBound(vals) <= Module1.ChLength Then ReDim Preserve vals(Module1.ChLength + 2)
Set vals(nextin) = v
nextin = (nextin + 1) Mod Module1.ChLength
End Sub

Public Function ReadVal() As ChannelFrm
'Return a value from the list
Set ReadVal = vals(nextout)
List1.RemoveItem (0)
nextout = (nextout + 1) Mod Module1.ChLength
End Function

Public Function AbleToRead() As Boolean
If List1.ListCount > 0 Then AbleToRead = True Else AbleToRead = False
End Function

Public Function AbleToWrite(p As ProcessFrm) As Boolean
Dim i As Integer
```

```
If Module1.ChLengthZero = False Then
    If List1.ListCount < Module1.ChLength Then AbleToWrite = True Else AbleToWrite
= False
End If

If Module1.ChLengthZero = True Then 'harder - need to see if any process will read
a value from us
    If MDIForm1.MDIWouldRead(Me, p) Then AbleToWrite = True Else AbleToWrite =
False
End If
End Function

Private Sub Form_Load()
List1.Clear
ReDim vals(Module1.ChLength + 2)
nextin = 0
nextout = 0
End Sub


Public Sub ClearBuf()
List1.Clear
ReDim vals(Module1.ChLength + 2)
nextin = 0
nextout = 0
End Sub
```

D.2.2 MDIForm1.frm




```
Option Explicit

Private Sub AutoHide_Click()
'When set, hide any channels which are empty and no longer known to any process
If AutoHide.Checked = True Then
    AutoHide.Checked = False
Else
    AutoHide.Checked = True
End If
End Sub

Private Sub Exit_Click()
Call Unload(Me)
End Sub

Private Sub Go_Click()

End Sub

Private Sub Length_Click()
SelChanLenFrm.Show (vbModal)
End Sub

Private Sub Load_Click()
Dim arr As Variant
CommonDialog1.Flags = cdlOFNHideReadOnly
CommonDialog1.Filter = "All Files (*.*)|*.*|XML Files(*.xml)|*.xml"
CommonDialog1.FilterIndex = 2
```

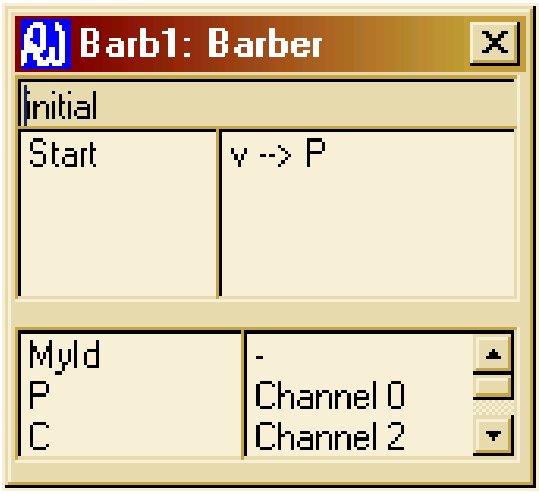
```
Call CommonDialog1.ShowOpen
If CommonDialog1.filename <> "" Then
    arr = Split(CommonDialog1.filename, "\")
    Caption = "RDX: " & arr(UBound(arr))
    Module1.filename = CommonDialog1.filename
    'Read the file into doc
    Set Module1.txtStream = fileSysObj.OpenTextFile(Module1.filename, 1)
    Module1.doc.loadXML (Module1.txtStream.ReadAll)
    Module1.txtStream.Close
    'As a default, select any process in the file:
    Set Module1.nodes = Module1.doc.getElementsByTagName("Model")
    Set Module1.el = Module1.nodes.Item(0)
    Set Module1.nodes = Module1.el.getElementsByTagName("Process")
    Set Module1.el = Module1.nodes.nextNode
    Module1.CurrentModel = Module1.el.getAttribute("Name")
    Call SelMod_Click
End If
End Sub

Private Sub MDIForm_Load()
'Call MsgBox("this is mdi loading")
Set Module1.doc = New DOMDocument
Set Module1.fileSysObj = CreateObject("Scripting.FileSystemObject")
ReDim Module1.Channels(10)
ReDim Module1.Processes(10)
Module1.ChLength = 5 'Default value
Module1.ChLengthZero = False 'Also the default value
End Sub

Private Sub SelMod_Click()
Dim arr As Variant
Dim i As Integer
Dim j As Integer
Dim k As Integer
Dim l As Integer
Dim b As Boolean
Dim ch1 As String
Dim ch2 As String
Dim pinst1 As String
Dim pinst2 As String
Dim obj1 As Object
Dim obj2 As Object

'Unload any forms already existing and clear the arrays
i = 0
While i < UBound(Module1.Processes)
    If Not Module1.Processes(i) Is Nothing Then Call Unload(Module1.Processes(i))
    i = i + 1
Wend
ReDim Module1.Processes(10)
i = 0
While i < UBound(Module1.Channels)
    If Not Module1.Channels(i) Is Nothing Then Call Unload(Module1.Channels(i))
    i = i + 1
Wend
ReDim Module1.Channels(10)

'If there is more than one in the file, find out which model to load...
Set Module1.nodes = Module1.doc.getElementsByTagName("Model")
Set Module1.el = Module1.nodes.Item(0)
Set Module1.nodes = Module1.el.getElementsByTagName("Instance")
Module1.nodes.Reset
If Module1.nodes.Length > 1 Then
    Call SelModelFrm.Show(vbModal)
'Else
'    Module1.CurrentModel = nodes.Item(0).Attributes.getNamedItem("Name")
'    'Set Module1.el = nodes.nextNode
'    'Module1.CurrentModel = el.getAttribute("Instance")
End If

'10/7/02 Put the name of the model being executed on the title bar
If Module1.filename <> "" Then
    arr = Split(Module1.filename, "\")
```

```
    MDIForm1.Caption = "RDX: " & arr(UBound(arr)) & " - " & Module1.CurrentModel
End If

'Find the model to execute and make process and channel windows
Set Module1.nodes = Module1.doc.getElementsByTagName("Model")
Set Module1.el = Module1.nodes.Item(0)
Set Module1.nodes = Module1.el.getElementsByTagName("Instance")
Module1.nodes.Reset
i = 0
Do While i < Module1.nodes.Length
    Set Module1.el = Module1.nodes.nextNode
    If Module1.el.getAttribute("Name") = Module1.CurrentModel Then Exit Do
    i = i + 1
Loop

Set Module1.nodes2 = el.getElementsByTagName("ProcInstance")
Module1.nodes.Reset
i = 0
Do While i < Module1.nodes2.Length
    Set Module1.el2 = Module1.nodes2.nextNode
    Set Processes(i) = New ProcessFrm
    Processes(i).SetName (el2.getAttribute("Name"))
    Processes(i).SetType (el2.getAttribute("Type"))
    Processes(i).SetCaption
    Processes(i).Show
    i = i + 1
Loop
'Call MsgBox(Module1.CurrentModel & " =?= " & Module1.el.getAttribute("Name"))


'Now add in the names of channels that we know at the start
Set nodes = Module1.doc.getElementsByTagName("Model")
Set el = nodes.Item(0)
Set nodes = el.getElementsByTagName("Instance")
nodes.Reset
i = 0
Do While i < nodes.Length
    Set el = nodes.nextNode
    If el.getAttribute("Name") = Module1.CurrentModel Then Exit Do
    i = i + 1
Loop
DoEvents

ReDim Channels(10)
Set nodes = el.getElementsByTagName("Connection")
nodes.Reset
i = 0
Do While i < nodes.Length
    If i >= UBound(Channels) Then ReDim Preserve Channels(UBound(Channels) + 15)
    Set el = nodes.nextNode 'Get the details of both ends:
    Set nodes2 = el.getElementsByTagName("End")
    Set el2 = nodes2.nextNode
    ch1 = el2.getAttribute("Channel")
    pinst1 = el2.getAttribute("ProcInstance")
    Set el2 = nodes2.nextNode
    ch2 = el2.getAttribute("Channel")
    pinst2 = el2.getAttribute("ProcInstance")

    'Find the first end
    j = 0
    Do While Not Processes(j) Is Nothing
        If Processes(j).ProcName = pinst1 Then 'find the channel
        Set obj1 = Processes(j).GetChan(ch1)
        Exit Do
        End If
    j = j + 1
    Loop
    'Find the second end
    k = 0
    Do While Not Processes(k) Is Nothing
        If Processes(k).ProcName = pinst2 Then 'find the channel
        Set obj2 = Processes(k).GetChan(ch2)
        Exit Do
```

```
        End If
    k = k + 1
    Loop

    If obj1 Is Nothing And obj2 Is Nothing Then 'Make a new channel and connect it
    l = 0
    Do While Not Channels(l) Is Nothing
        l = l + 1
    Loop
    Set Channels(l) = New ChannelFrm
    Call Processes(j).SetChan(ch1, Channels(l))
    Call Processes(k).SetChan(ch2, Channels(l))
    Channels(l).Caption = "Channel " & l
    Channels(l).Show
    DoEvents
    End If

    If Not obj1 Is Nothing And Not obj2 Is Nothing Then
    'There is a problem here: connecting a port on an instance which is already
connected to a channel
    'to a port on an instance which is also already connected to a channel makes
one of the channels redundant
    'and is not handled properly here. 25/7/01
        Call MsgBox("Connection being made between two ""pairs"": expect incorrect
result", vbOKOnly, "Warning")
    End If

    If Not obj1 Is Nothing Then 'connect the existing channel
        Call Processes(k).SetChan(ch2, obj1)
    End If

    If Not obj2 Is Nothing Then 'connect the existing channel
        Call Processes(j).SetChan(ch1, obj2)
    End If

    Set obj1 = Nothing
    Set obj2 = Nothing
    'Call MsgBox("connection" & pinst1 & ch1 & pinst2 & ch2)
    i = i + 1
Loop
Call MdiInsertChannels
Call UpdateEvents

End Sub

Public Sub UpdateEvents()
'Cause each process to re-evaluate its list of available events
Dim i As Integer
DoEvents
Do While Not Processes(i) Is Nothing And i < UBound(Processes)
    Call Processes(i).EvaluateEvents
    i = i + 1
Loop
DoEvents

If AutoHide.Checked = True Then Call MDIKillChannels
DoEvents
End Sub

Public Sub MdiInsertChannels()
'Cause each process to re-evaluate its list of available events
Dim i As Integer
i = 0
DoEvents
'Runtime error here: i
Do While Not Processes(i) Is Nothing And i < UBound(Processes)
    Call Processes(i).InsertChannels
    i = i + 1
Loop
DoEvents
End Sub

Public Function MDIWouldRead(ch As ChannelFrm, p As ProcessFrm) As Boolean
```

```
'Search for a process that would read ch
Dim i As Integer
i = 0
Do While i < UBound(Processes) And Not Processes(i) Is Nothing
    If Processes(i).WillingToRead(ch) = True And Processes(i).Caption <> p.Caption
Then
        MDIWouldRead = True
        Exit Function
    End If
    i = i + 1
Loop
MDIWouldRead = False
End Function

Public Sub MDIClearChannels()
Dim i As Integer
i = 0
Do While i < UBound(Channels) And Not Channels(i) Is Nothing
    Call Channels(i).ClearBuf
    i = i + 1
Loop
End Sub


Public Sub MDIKillChannels()
'For each empty channel, see if a process knows it.  If not, hide it.
Dim i As Integer
Dim j As Integer
Dim known As Boolean

i = 0
Do While i < UBound(Channels) And Not Channels(i) Is Nothing
    'see if any process knows this channel....
    known = False
    j = 0
    Do While j < UBound(Processes) And Not Processes(j) Is Nothing
        If Processes(j).KnowsChannel(Channels(i)) = True Then known = True
        j = j + 1
    Loop
    If known = False Then Channels(i).Visible = False
    i = i + 1
Loop
End Sub

Private Sub ShowAllCh_Click()
Dim i As Integer
i = 0

Do While i < UBound(Channels) And Not Channels(i) Is Nothing
    Channels(i).Visible = True
    i = i + 1
Loop
End Sub

Private Sub ShowChan_Click()
Dim i As Integer
i = 0

If ShowChan.Checked = True Then ShowChan.Checked = False
If ShowChan.Checked = False Then ShowChan.Checked = True

Do While i < UBound(Channels) And Not Channels(i) Is Nothing
    Channels(i).Visible = ShowChan.Checked
    i = i + 1
Loop

End Sub
```

D.2.3 ProcessFrm.frm




```
Option Explicit

'Colours used for the background of the events list with zero-length channels:
Const SENDER_ZERO = &H80000010  '&HFF&  'The colour of the initiating process in
the interim state
Const BACK_COLOUR = &H80000005          'The colour of the background otherwise
'It is important not to set these two colours the same as they are used in the
code to identify the process which is forbidden to
' perform a read of a channel on account of being the one that did the write (or
create)


Private Type EventType
name As String
Type As String
Before As String
After As String
Channel As String
Value As String
Drawn As Boolean
End Type

Private Type ChannelType
MyName As String
GlobalName As ChannelFrm
End Type

Private ProcType As String
Public ProcName As String
Private nodes As IXMLDOMNodeList
Private el As IXMLDOMElement
Private Events() As EventType
Private Channels() As ChannelType

Public Sub SetType(t As String)
ProcType = t
End Sub

Public Sub SetName(n As String)
ProcName = n
ProcessFrm.Caption = ProcName & ": " & ProcType
End Sub

Public Sub SetCaption()
```

```
Caption = ProcName & ": " & ProcType
End Sub

Private Sub Form_Load()

Dim i As Integer
Dim j As Integer
Dim k As Integer
Dim nodes As IXMLDOMNodeList
Dim nodes2 As IXMLDOMNodeList
Dim el As IXMLDOMElement
Dim el2 As IXMLDOMElement

If ProcName <> "" Then 'included so as to skip the form created automatically by
VB
    'load information about ourselves
    ReDim Events(10)
    ReDim Channels(10)
    Set nodes = Module1.doc.getElementsByTagName("Model")
    Set el = nodes.Item(0)
    Set nodes = el.getElementsByTagName("Process")
    nodes.Reset
    i = 0
    Do While i < nodes.length
        Set el = nodes.nextNode
        If el.getAttribute("Name") = ProcType Then Exit Do
        i = i + 1
    Loop
    Set nodes = el.getElementsByTagName("Event")
    nodes.Reset
    i = 0
    Do While i < nodes.length
        If i >= UBound(Events) Then ReDim Preserve Events(UBound(Events) + 15)
        Set el = nodes.nextNode
        Events(i).name = el.getAttribute("Name")
        Events(i).Type = el.getAttribute("Type")
        Events(i).Before = el.getAttribute("Before")
        Events(i).After = el.getAttribute("After")
        Events(i).Channel = el.getAttribute("Channel")
        Events(i).Value = el.getAttribute("Value")
        'Add the channel used in this event to our list, if necessary
        j = 0
        Do While j < UBound(Channels) And Channels(j).MyName <> ""
            If Channels(j).MyName = Events(i).Channel Then Exit Do
            j = j + 1
        Loop
        If j >= UBound(Channels) Then ReDim Preserve Channels(UBound(Channels) +
10)
        If Channels(j).MyName = "" Then Channels(j).MyName = Events(i).Channel
        i = i + 1
    Loop
End If
End Sub

Public Function GetChan(chname As String) As ChannelFrm
'Returns the global name of the channel we know as chname, or nothing
Dim i As Integer
Do While i < UBound(Channels)
    If Channels(i).MyName = chname Then
        Set GetChan = Channels(i).GlobalName
        Exit Function
    End If
    i = i + 1
Loop
Set GetChan = Nothing
End Function


Public Sub SetChan(chname As String, gname As ChannelFrm)
'Returns the global name of the channel we know as chname, or nothing
Dim i As Integer
Do While i < UBound(Channels)
    If Channels(i).MyName = chname Then
```

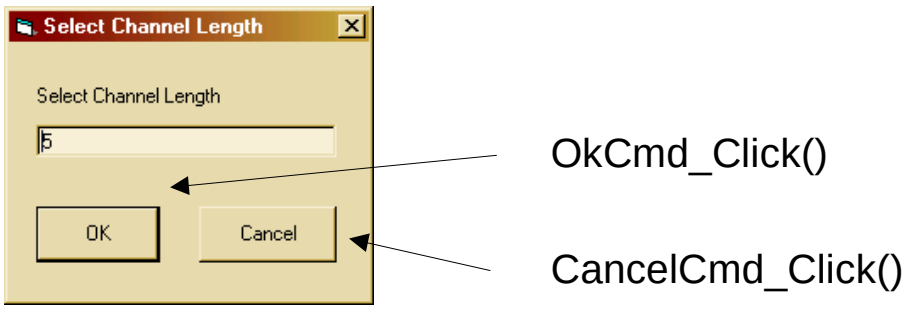
```
        Set Channels(i).GlobalName = gname
        Exit Sub
    End If
    i = i + 1
Loop
End Sub

Public Sub EvaluateEvents()
'See what events this process can do...
Dim i As Integer
Dim j As Integer
Dim k As Integer
List1.Clear
List2.Clear
List5.Clear

'Reset the background colour if required
If Module1.ChLengthZero = True And Module1.NoWrites = False Then
     List1.BackColor = BACK_COLOUR
     List5.BackColor = BACK_COLOUR
End If

If List1.BackColor = SENDER_ZERO Then 'We just did a send or a create so we cannot
have any available events
    Exit Sub
End If

i = 0
Do While Not Events(i).name = ""
    'Find the channel for this event and see if it is available
    If Events(i).Before = StateText.Text Then
        j = 0
        Do While Not Channels(j).MyName = ""
            If Not Channels(j).GlobalName Is Nothing And Events(i).Channel =
Channels(j).MyName Then
                If Events(i).Type = "Write" And
Channels(j).GlobalName.AbleToWrite(Me) = True Then
                    'Check that the value to be written "exists"
                    k = 0
                    Do While Not Channels(k).MyName = ""
                        If Channels(k).MyName = Events(i).Value Then Exit Do
                        k = k + 1
                    Loop
                    If Not Channels(k).GlobalName Is Nothing Then
                        If Module1.ChLengthZero = False Then
                            List1.AddItem (Events(i).name)
                            List5.AddItem (Events(i).Value & " --> " &
Events(i).Channel)
                        End If
                        If Module1.ChLengthZero = True And Module1.NoWrites =
False Then
                            List1.AddItem (Events(i).name)
                            List5.AddItem (Events(i).Value & " --> " &
Events(i).Channel)
                        End If
                    End If
                End If
                If Events(i).Type = "Create" And
Channels(j).GlobalName.AbleToWrite(Me) = True Then
                    If Module1.ChLengthZero = False Then
                        List1.AddItem (Events(i).name)
                        List5.AddItem (Events(i).Value & " --> " &
Events(i).Channel)
                    End If
                    If Module1.ChLengthZero = True And Module1.NoWrites = False
Then
                        List1.AddItem (Events(i).name)
                        List5.AddItem (Events(i).Value & " --> " &
Events(i).Channel)
                    End If
                End If
                If Events(i).Type = "Read" And Channels(j).GlobalName.AbleToRead =
True Then
```

```
                    If Module1.ChLengthZero = False Then
                        List1.AddItem (Events(i).name)
                        List5.AddItem (Events(i).Value & " <-- " &
Events(i).Channel)
                    End If
                    If Module1.ChLengthZero = True And Module1.NoWrites = True
Then
                        List1.AddItem (Events(i).name)
                        List5.AddItem (Events(i).Value & " <-- " &
Events(i).Channel)
                    End If
                End If
            End If
            j = j + 1
        Loop
    End If
    List2.AddItem (Events(i).name)
    i = i + 1
Loop
'List the channels and values we know about
List3.Clear
List4.Clear
i = 0
Do While Not Channels(i).MyName = ""
    List3.AddItem (Channels(i).MyName)
    If Not Channels(i).GlobalName Is Nothing Then List4.AddItem
(Channels(i).GlobalName.Caption) Else List4.AddItem ("-")
    i = i + 1
Loop
End Sub



Private Sub List1_DblClick()
'Perform the action (if we can)
Dim i As Integer
Dim j As Integer
Dim k As Integer
Dim s As String

i = 0
'>>>>>>>>>>>>>>>>>>>>>>>>>>>>>. Attempt to save problems where event names are re-
used...
Do While i < UBound(Events) And Not (List1.List(List1.ListIndex) = Events(i).name
And Events(i).Before = StateText.Text)
'Do While i < UBound(Events) And List1.List(List1.ListIndex) <> Events(i).name
    i = i + 1
Loop
j = 0
Do While j < UBound(Channels) And Channels(j).MyName <> Events(i).Channel
    j = j + 1
Loop
Do While k < UBound(Channels) And Channels(k).MyName <> Events(i).Value
    k = k + 1
Loop

'Now the event is events(i), the channel is channels(j) and the value is
channels(k)
If Events(i).Type = "Write" Then
    Module1.NoWrites = True
    If Module1.ChLengthZero = True Then
        List1.BackColor = SENDER_ZERO
        List5.BackColor = SENDER_ZERO
    End If

    Call Channels(j).GlobalName.WriteVal(Channels(k).GlobalName)
End If

If Events(i).Type = "Read" Then
    Module1.NoWrites = False
'    If Module1.ChLengthZero = True Then    'colour is reset by sub EvaluateEvents
'        List1.BackColor = &H80000005
'        List5.BackColor = &H80000005
```

```
'    End If
    Set Channels(k).GlobalName = Channels(j).GlobalName.ReadVal
End If

If Events(i).Type = "Create" Then
    Module1.NoWrites = True
        If Module1.ChLengthZero = True Then
        List1.BackColor = SENDER_ZERO
        List5.BackColor = SENDER_ZERO
    End If
    'Create a new channel, insert its name into our array
    Dim c As Integer
    c = 0
    Do While Not Module1.Channels(c) Is Nothing And c < UBound(Module1.Channels)
'Find an empty place in the channels array
        c = c + 1
    Loop
    If c = UBound(Module1.Channels) Then ReDim Preserve
Module1.Channels(UBound(Module1.Channels) + 20) 'Enlarge the array if it is full
    Set Module1.Channels(c) = New ChannelFrm
    Module1.Channels(c).Caption = "Channel " & c
    Module1.Channels(c).Show
    Set Channels(k).GlobalName = Module1.Channels(c)
    Call Channels(j).GlobalName.WriteVal(Channels(k).GlobalName)
    DoEvents
End If
'Remove any trailing "=" from the new state name...
s = Events(i).After
i = InStr(s, "=")
If i > 0 Then StateText.Text = Left(s, i - 1) Else StateText.Text = s
MDIForm1.UpdateEvents
End Sub

Public Sub InsertChannels()
'After connections have been made, inserts entries in the array
'for channels that the process knows about, but are not (yet) connected anywhere
Dim i As Integer
Dim j As Integer

i = 0
Do While i < UBound(Events)
    j = 0
    Do While Not Channels(j).MyName = "" And j < UBound(Channels)
        If Channels(j).MyName = Events(i).Channel Then Exit Do
        j = j + 1
    Loop
    If Channels(j).MyName <> Events(i).Channel Then
        If j >= UBound(Channels) Then ReDim Preserve Channels(UBound(Channels) +
10)
        Channels(j).MyName = Events(i).Channel
    End If
    'Do the same for values:
    j = 0
    Do While Not Channels(j).MyName = "" And j < UBound(Channels)
        If Channels(j).MyName = Events(i).Value Then Exit Do
        j = j + 1
    Loop
    If Channels(j).MyName <> Events(i).Value Then
        If j >= UBound(Channels) Then ReDim Preserve Channels(UBound(Channels) +
10)
        Channels(j).MyName = Events(i).Value
    End If
    i = i + 1
Loop
End Sub

Private Sub List4_Scroll()
List3.TopIndex = List4.TopIndex
End Sub

Public Function WillingToRead(ch As ChannelFrm) As Boolean
Dim i As Integer
Dim j As Integer
```

```
Dim OurChName As String
'Given a channel returns true if, were that channel non-empty, we could read it

'See what we call this channel (if we know it at all)
j = 0
OurChName = ""
Do While j < UBound(Channels) And Channels(j).MyName <> ""
    If Not Channels(j).GlobalName Is Nothing Then 'We may know names that don't
correspond to a channel
        If Channels(j).GlobalName.Caption = ch.Caption Then
            OurChName = Channels(j).MyName
            Exit Do
        End If
    End If
    j = j + 1
Loop
If OurChName = "" Then 'We don't know the channel at all - we could never read it
    WillingToRead = False
    Exit Function
End If

'Look through events for one we could from present state that reads the channel
i = 0
Do While i < UBound(Events) And Not Events(i).name = ""
    If Events(i).Before = StateText.Text And Events(i).Type = "Read" And OurChName
= Events(i).Channel Then
        WillingToRead = True
        Exit Function
    End If
    i = i + 1
Loop
'Didn't find one...
WillingToRead = False
End Function

Public Function KnowsChannel(ch As ChannelFrm) As Boolean
'Returns true if ch is a channel this process has a name for
Dim i As Integer
i = 0

Do While i < UBound(Channels) And Not Channels(i).GlobalName Is Nothing
        If Channels(i).GlobalName.Caption = ch.Caption Then
        KnowsChannel = True
        Exit Function
    End If
    i = i + 1
Loop
'Its not there
KnowsChannel = False
End Function
```

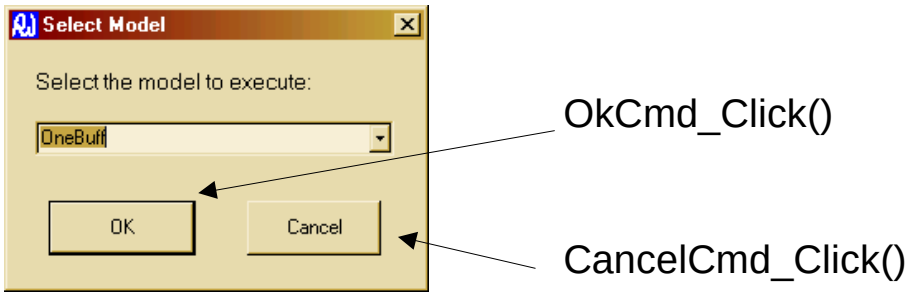
D.2.4 SelChanLenFrm.frm




```
Option Explicit
```

```
Private Sub CancelCmd_Click()
SelChanLenFrm.Hide
End Sub

Private Sub Form_Activate()

If Module1.ChLengthZero = True Then
    Text1.Text = "0"
Else
    Text1.Text = Module1.ChLength
End If
End Sub

Private Sub OkCmd_Click()
If Val(Text1.Text) <> 0 Then
    Module1.ChLengthZero = False
    Module1.ChLength = Val(Text1.Text)
End If

If Val(Text1.Text) = 0 Then
    Module1.ChLength = 1
    Module1.ChLengthZero = True
    Call MDIForm1.MDIClearChannels
End If
SelChanLenFrm.Hide
'Re-evaluate which events are available
Call MDIForm1.UpdateEvents
End Sub
```

D.2.5 SelModelFrm2.frm




```
Option Explicit

Private Sub CancelCmd_Click()
SelModelFrm.Hide
End Sub

Private Sub Form_Activate()
Dim i As Integer
Combo1.Clear
Set Module1.nodes = Module1.doc.getElementsByTagName("Model")
Set Module1.el = Module1.nodes.Item(0)
Set Module1.nodes = Module1.el.getElementsByTagName("Instance")
Module1.nodes.Reset
Set Module1.el = Module1.nodes.nextNode
i = 0
While i < Module1.nodes.length
    Combo1.AddItem (Module1.el.getAttribute("Name"))
    Set Module1.el = Module1.nodes.nextNode
    i = i + 1
Wend
Combo1.ListIndex = 0
End Sub
```

```
Private Sub OkCmd_Click()
If Combo1.List(Combo1.ListIndex) = "" Then Exit Sub
Module1.CurrentModel = Combo1.List(Combo1.ListIndex)
SelModelFrm.Hide
End Sub
```

D.2.6 RDXModule1.bas

```
Option Explicit

Public tempfile As String 'The name of the file viewed by the browser object
Public fileSysObj As Object
Public txtStream As Object

Public filename As String 'The name of the "real" file in use

Public doc As DOMDocument 'Used to store the present state of the model
Public el As IXMLDOMElement
Public el2 As IXMLDOMElement
Public nodes As IXMLDOMNodeList
Public nodes2 As IXMLDOMNodeList
Public node As IXMLDOMNode

Public CurrentModel As String

Public Processes() As Form
Public Channels() As Form


Public ChLength As Integer
Public ChLengthZero As Boolean
Public NoWrites As Boolean
```

D.3 RDT

This is the model generation tool. In its main window it displays the contents of the current model file as formatted XML text. Models are created by using a collection of dialogue boxes. The application is also able to display the model and the processes from which it is constructed as diagrams.

The application has twelve forms and two modules:

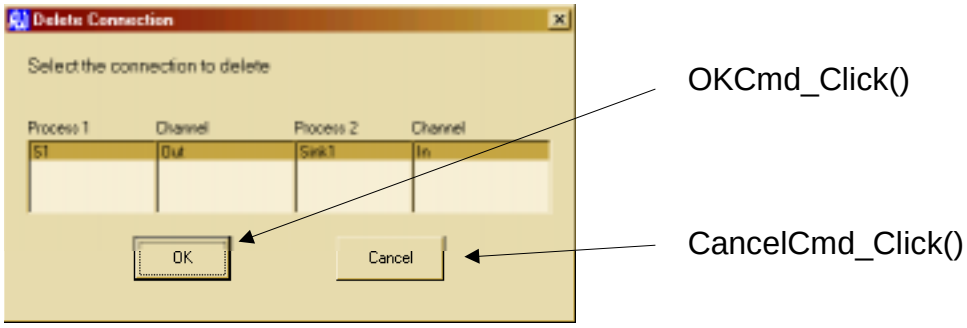
DelConnFrm.frm
DelEventFrm.frm
DelInstanceFrm.frm
DelModelFrm.frm
RDT1.frm
ModelView.frm
NewConnectionFrm.frm
NewEventFrom.frm
NewProcInstFrm.frm
ProcViewFrm.frm
SelModelFrm.frm

SelProcFrm.frm

RDTModule1.bas
RDTModule2.bas

RDT1.frm is loaded and displayed when the application is started.

D.3.1 DelConnFrm.frm

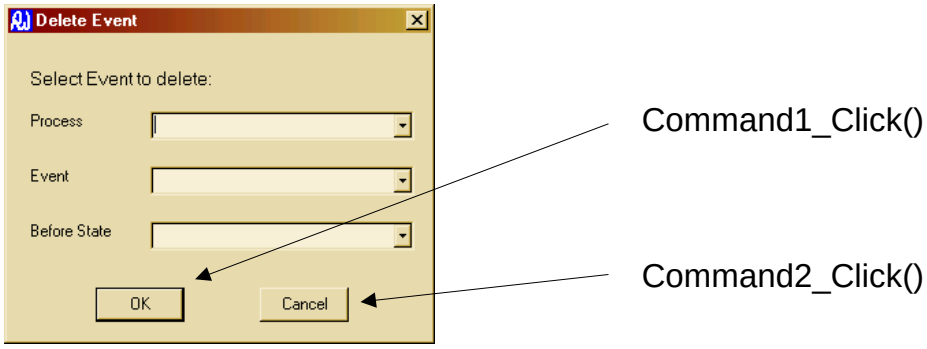


This form lists connections between processes instances in a model and permits the user to select and delete one of them.

```
Option Explicit

Private Sub CancelCmd_Click()
Call DelConnFrm.Hide
End Sub

Private Sub Form_Activate()
'List the connections in the current model
Dim i As Integer

List1.Clear
List2.Clear
List3.Clear
List4.Clear

Set Module1.nodes = Module1.doc.getElementsByTagName("Model")
Set Module1.el = Module1.nodes.Item(0)
Set Module1.nodes = Module1.el.getElementsByTagName("Instance")
'Find the current instance
Module1.nodes.Reset
Set Module1.el = Module1.nodes.nextNode
i = 0
While i < Module1.nodes.length And Module1.el.getAttribute("Name") <>
Module2.CurrentModel
    Set Module1.el = Module1.nodes.nextNode
    i = i + 1
Wend

Set Module1.el = Module1.nodes(i)
Set Module1.nodes = Module1.el.getElementsByTagName("Connection")
Module1.nodes.Reset
Set Module1.el = Module1.nodes.nextNode
i = 0
While i < Module1.nodes.length
    Set Module1.nodes2 = Module1.el.getElementsByTagName("End")
    Set el2 = Module1.nodes2.nextNode
```

```
    List1.AddItem (Module1.el2.getAttribute("ProcInstance"))
    List2.AddItem (Module1.el2.getAttribute("Channel"))
    Set el2 = Module1.nodes2.nextNode
    List3.AddItem (Module1.el2.getAttribute("ProcInstance"))
    List4.AddItem (Module1.el2.getAttribute("Channel"))
    Set Module1.el = Module1.nodes.nextNode
    i = i + 1
Wend
If List1.ListCount > 0 Then
    List1.ListIndex = 0
    List2.ListIndex = 0
    List3.ListIndex = 0
    List4.ListIndex = 0
End If
End Sub

Private Sub List1_Click()
List2.ListIndex = List1.ListIndex
List3.ListIndex = List1.ListIndex
List4.ListIndex = List1.ListIndex
End Sub

Private Sub List2_Click()
List1.ListIndex = List2.ListIndex
List3.ListIndex = List2.ListIndex
List4.ListIndex = List2.ListIndex
End Sub

Private Sub List3_Click()
List1.ListIndex = List3.ListIndex
List2.ListIndex = List3.ListIndex
List4.ListIndex = List3.ListIndex
End Sub

Private Sub List4_Click()
List1.ListIndex = List4.ListIndex
List2.ListIndex = List4.ListIndex
List3.ListIndex = List4.ListIndex
End Sub

Private Sub OKCmd_Click()
Dim i As Integer
Set Module1.nodes = Module1.doc.getElementsByTagName("Model")
Set Module1.el = Module1.nodes.Item(0)
Set Module1.nodes = Module1.el.getElementsByTagName("Instance")
'Find the current instance
Module1.nodes.Reset
Set Module1.el = Module1.nodes.nextNode
i = 0
While i < Module1.nodes.length And Module1.el.getAttribute("Name") <>
Module2.CurrentModel
    Set Module1.el = Module1.nodes.nextNode
    i = i + 1
Wend

Set Module1.el = Module1.nodes(i)
Set Module1.nodes = Module1.el.getElementsByTagName("Connection")
Module1.nodes.Reset
Set Module1.el2 = Module1.nodes.Item(List4.ListIndex)
Call Module1.el.removeChild(el2)
DelConnFrm.Hide
Set Module1.txtStream = Module1.fileSysObj.OpenTextFile(tempfile, 2)
Module1.txtStream.Write doc.xml
Module1.txtStream.Close
Set Module1.txtStream = Nothing
If XMLViewForm.Visible = True Then Call XMLViewForm.UpdateXMLView
End Sub
```

D.3.2 DelEventFrm.frm



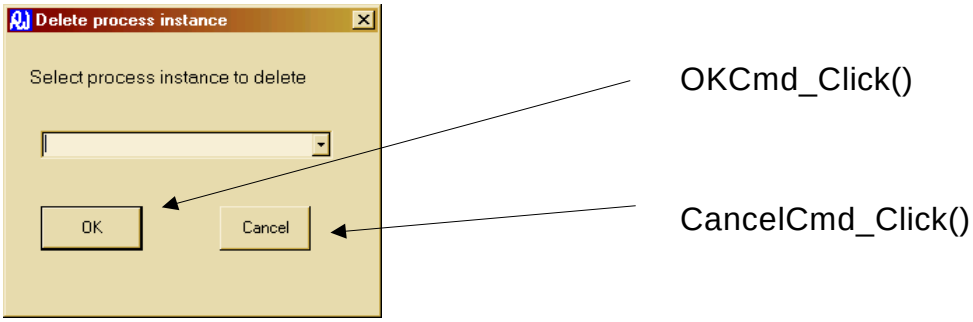
This form is used by the user defining a process. It lists the defined events of a selected process. Since process names may be re-used, the form also lists the "before" states of the processes to enable the user to identify and remove the correct event.

```
Option Explicit

Private Sub Combo1_Click()
Dim i As Integer
Dim j As Integer
Dim k As Integer

Combo2.Clear
Set Module1.nodes = Module1.doc.getElementsByTagName("Model")
Set Module1.el = Module1.nodes.Item(0)
Set Module1.nodes = Module1.el.getElementsByTagName("Process")

Module1.nodes.Reset
Set Module1.el = Module1.nodes.nextNode

i = 0
While i < Module1.nodes.length
    If Combo1.List(Combo1.ListIndex) = Module1.el.getAttribute("Name") Then
        Set Module1.nodes2 = Module1.el.getElementsByTagName("Event")
        Set Module1.el2 = Module1.nodes2.nextNode
        j = 0
        While j < Module1.nodes2.length
            'See if the name is in there already
            k = 0
            While k < Combo2.ListCount And Combo2.List(k) <>
Module1.el2.getAttribute("Name")
                k = k + 1
            Wend
            If k >= Combo2.ListCount Then Combo2.AddItem
(Module1.el2.getAttribute("Name"))
            Set Module1.el2 = Module1.nodes2.nextNode
            j = j + 1
        Wend
    End If
    Set Module1.el = Module1.nodes.nextNode
    i = i + 1
Wend
End Sub

Private Sub Combo2_Click()
Dim i As Integer
Dim j As Integer

Combo3.Clear
Set Module1.nodes = Module1.doc.getElementsByTagName("Model")
Set Module1.el = Module1.nodes.Item(0)
```

```
Set Module1.nodes = Module1.el.getElementsByTagName("Process")

Module1.nodes.Reset
Set Module1.el = Module1.nodes.nextNode

i = 0
While i < Module1.nodes.length
    If Combo1.List(Combo1.ListIndex) = Module1.el.getAttribute("Name") Then
        Set Module1.nodes2 = Module1.el.getElementsByTagName("Event")
        Set Module1.el2 = Module1.nodes2.nextNode
        j = 0
        While j < Module1.nodes2.length
            If Module1.el2.getAttribute("Name") = Combo2.List(Combo2.ListIndex)
Then
                Combo3.AddItem (Module1.el2.getAttribute("Before"))
            End If
            Set Module1.el2 = Module1.nodes2.nextNode
            j = j + 1
        Wend
    End If
    Set Module1.el = Module1.nodes.nextNode
    i = i + 1
Wend
End Sub

Private Sub Command1_Click()
Dim i As Integer
Dim j As Integer

If Combo1.List(Combo1.ListIndex) = "" Or Combo2.List(Combo2.ListIndex) = "" Then
    Call MsgBox("No event selected", vbOKOnly, "Error")
    Exit Sub
End If

Set Module1.nodes = Module1.doc.getElementsByTagName("Model")
Set Module1.el = Module1.nodes.Item(0)
Set Module1.nodes = Module1.el.getElementsByTagName("Process")

Module1.nodes.Reset
Set Module1.el = Module1.nodes.nextNode

i = 0
While i < Module1.nodes.length
    If Combo1.List(Combo1.ListIndex) = Module1.el.getAttribute("Name") Then
        Set Module1.nodes2 = Module1.el.getElementsByTagName("Event")
        Set Module1.el2 = Module1.nodes2.nextNode
        j = 0
        While j < Module1.nodes2.length
            If Module1.el2.getAttribute("Name") = Combo2.List(Combo2.ListIndex)
And Module1.el2.getAttribute("Before") = Combo3.List(Combo3.ListIndex) Then
                'delete the node and exit.......
                Call Module1.el.removeChild(el2)
                DelEventFrm.Hide
                Set Module1.txtStream = Module1.fileSysObj.OpenTextFile(tempfile,
2)
                Module1.txtStream.Write doc.xml
                Module1.txtStream.Close
                Set Module1.txtStream = Nothing
                If XMLViewForm.Visible = True Then Call XMLViewForm.UpdateXMLView
                Exit Sub
            End If
            Set Module1.el2 = Module1.nodes2.nextNode
            j = j + 1
        Wend
    End If
    Set Module1.el = Module1.nodes.nextNode
    i = i + 1
Wend
End Sub

Private Sub Command2_Click()
DelEventFrm.Hide
End Sub
```

```
Private Sub Form_Activate()
Dim i As Integer

Combo1.Clear
Combo2.Clear
Combo3.Clear
Set Module1.nodes = Module1.doc.getElementsByTagName("Model")
Set Module1.el = Module1.nodes.Item(0)
Set Module1.nodes = Module1.el.getElementsByTagName("Process")

Module1.nodes.Reset
Set Module1.el = Module1.nodes.nextNode
i = 0
While i < Module1.nodes.length
    Combo1.AddItem (Module1.el.getAttribute("Name"))
    Set Module1.el = Module1.nodes.nextNode
    i = i + 1
Wend
End Sub
```

D.3.3 DelInstanceFrm.frm

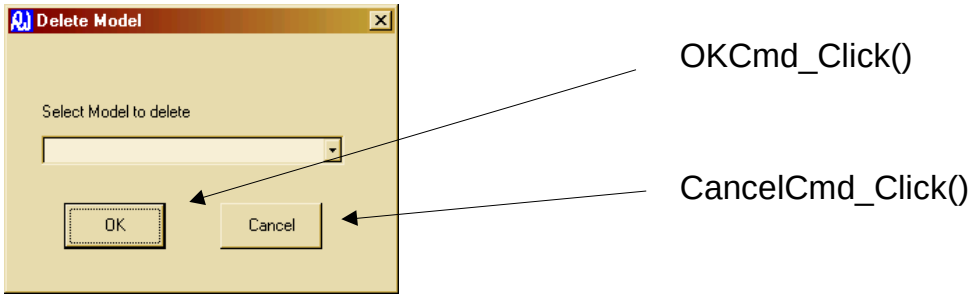


When working with a model, this form is used to permit the user to select and delete a process instance.

```
Option Explicit

Private Sub CancelCmd_Click()
Delinstance.Hide
End Sub

Private Sub Form_Activate()
Dim i As Integer
Combo1.Clear

Set Module1.nodes = Module1.doc.getElementsByTagName("Model")
Set Module1.el = Module1.nodes.Item(0)
Set Module1.nodes = Module1.el.getElementsByTagName("Instance")
'Find the current instance
Module1.nodes.Reset
Set Module1.el = Module1.nodes.nextNode
i = 0
While i < Module1.nodes.length And Module1.el.getAttribute("Name") <>
Module2.CurrentModel
    Set Module1.el = Module1.nodes.nextNode
    i = i + 1
Wend

Set Module1.el = Module1.nodes(i)
Set Module1.nodes = Module1.el.getElementsByTagName("ProcInstance")
Module1.nodes.Reset
Set Module1.el = Module1.nodes.nextNode
i = 0
```

```
While i < Module1.nodes.length
    Combo1.AddItem (el.getAttribute("Name"))
    Set Module1.el = Module1.nodes.nextNode
    i = i + 1
Wend
End Sub

Private Sub OKCmd_Click()
'delete the instance and update doc
Dim i As Integer
Set Module1.nodes = Module1.doc.getElementsByTagName("Model")
Set Module1.el = Module1.nodes.Item(0)
Set Module1.nodes = Module1.el.getElementsByTagName("Instance")
'Find the current instance
Module1.nodes.Reset
Set Module1.el = Module1.nodes.nextNode
i = 0
While i < Module1.nodes.length And Module1.el.getAttribute("Name") <>
Module2.CurrentModel
    Set Module1.el = Module1.nodes.nextNode
    i = i + 1
Wend
Set Module1.el2 = Module1.nodes(i)
Set Module1.nodes = Module1.el2.getElementsByTagName("ProcInstance")
Module1.nodes.Reset
Set Module1.el2 = Module1.nodes.nextNode
i = 0
While el2.getAttribute("Name") <> Combo1.List(Combo1.ListIndex) And i <
Module1.nodes.length
    Set Module1.el2 = Module1.nodes.nextNode
    i = i + 1
Wend
Call Module1.el.removeChild(el2)
Delinstance.Hide
Set Module1.txtStream = Module1.fileSysObj.OpenTextFile(tempfile, 2)
Module1.txtStream.Write doc.xml
Module1.txtStream.Close
Set Module1.txtStream = Nothing
If XMLViewForm.Visible = True Then Call XMLViewForm.UpdateXMLView
End Sub
```

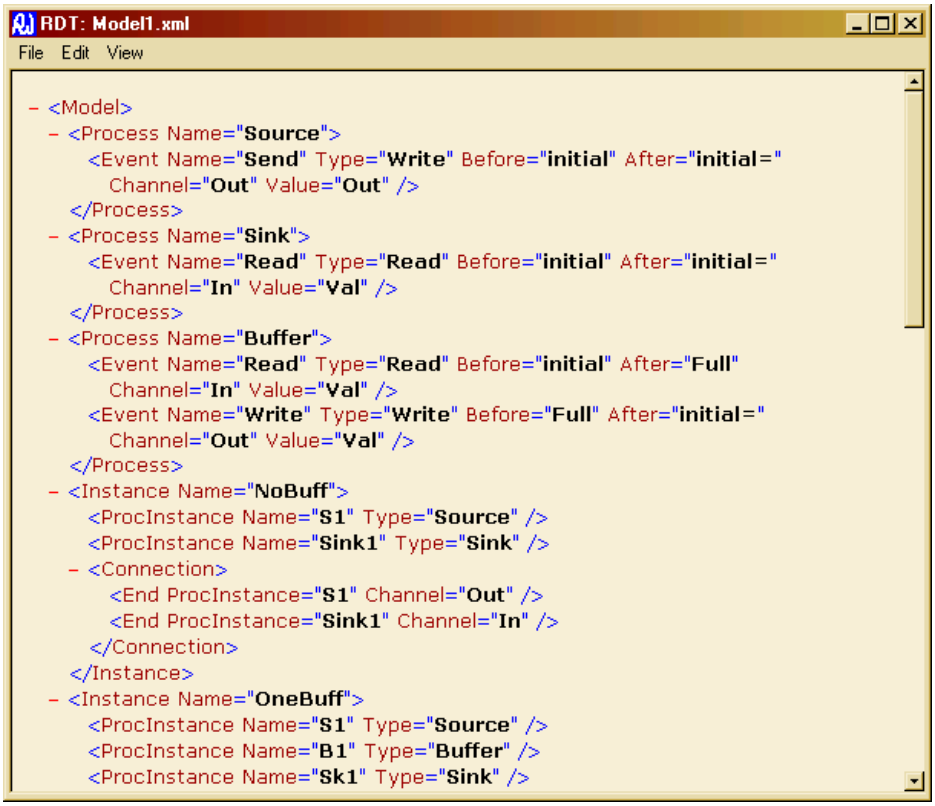
D.3.4 DelModelFrm.frm



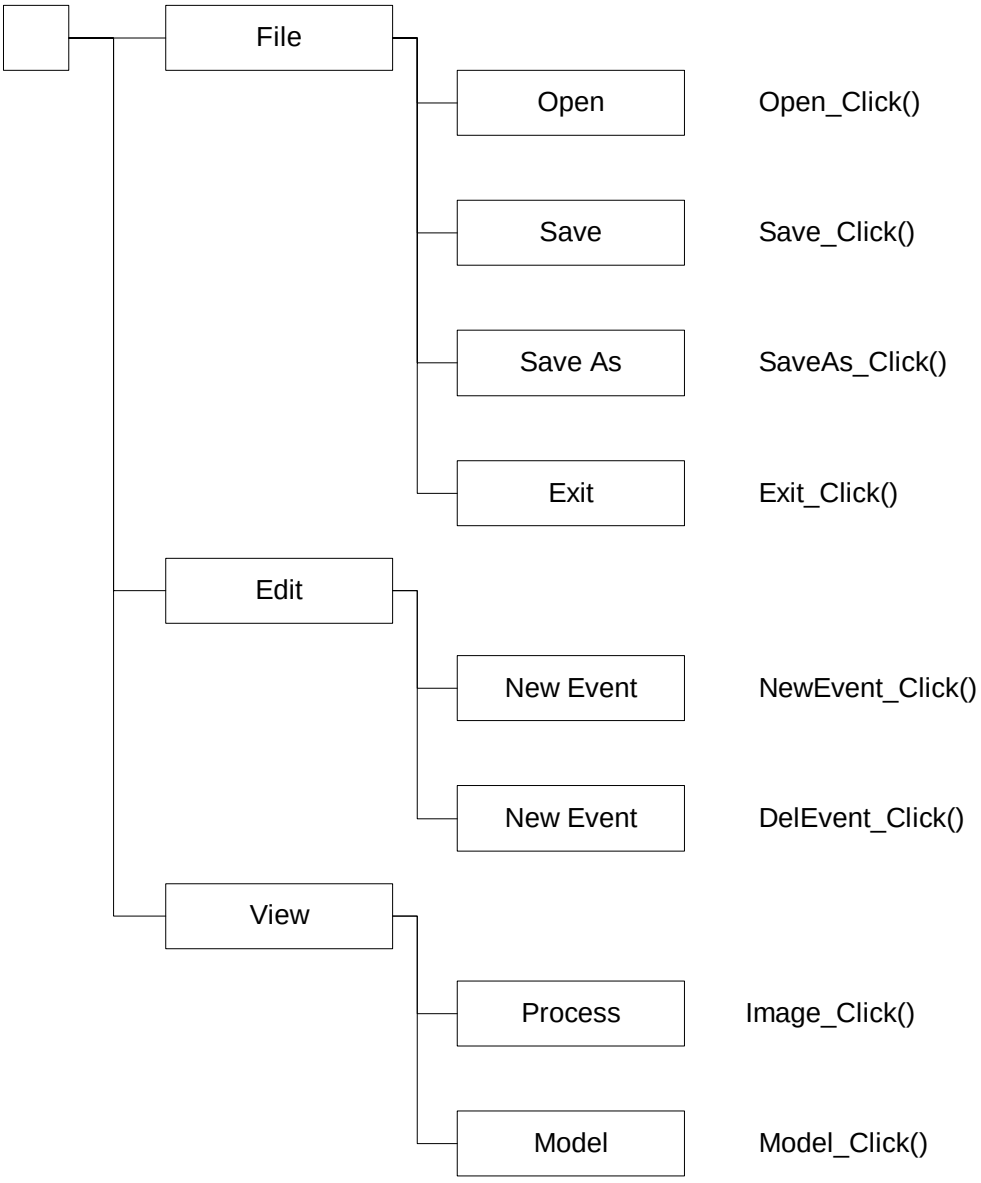

```
Option Explicit

Private Sub CancelCmd_Click()
DelModelFrm.Hide
End Sub

Private Sub Form_Activate()
Dim i As Integer

Combo1.Clear
Set Module1.nodes = Module1.doc.getElementsByTagName("Model")
Set Module1.el = Module1.nodes.Item(0)
Set Module1.nodes = Module1.el.getElementsByTagName("Instance")
Module1.nodes.Reset
```

```
Set Module1.el = Module1.nodes.nextNode
i = 0
While i < Module1.nodes.length
    Combo1.AddItem (Module1.el.getAttribute("Name"))
    Set Module1.el = Module1.nodes.nextNode
    i = i + 1
Wend
End Sub

Private Sub OKCmd_Click()
Dim i As Integer
If Combo1.List(Combo1.ListIndex) = "" Then
    Call MsgBox("No Model selected", vbOKOnly, "Error")
    Exit Sub
End If

Set Module1.nodes = Module1.doc.getElementsByTagName("Model")
Set Module1.el = Module1.nodes.Item(0)
Set Module1.nodes = Module1.el.getElementsByTagName("Instance")

Module1.nodes.Reset
Set Module1.el2 = Module1.nodes.nextNode

i = 0
While i < Module1.nodes.length
    If Combo1.List(Combo1.ListIndex) = Module1.el2.getAttribute("Name") Then
        'delete the node and exit.......
        Call Module1.el.removeChild(el2)
        Set Module1.txtStream = Module1.fileSysObj.OpenTextFile(tempfile, 2)
        Module1.txtStream.Write doc.xml
        Module1.txtStream.Close
        Set Module1.txtStream = Nothing
        If XMLViewForm.Visible = True Then Call XMLViewForm.UpdateXMLView
        DelModelFrm.Hide
        Module2.CurrentModel = ""
        Exit Sub
    End If
    Set Module1.el2 = Module1.nodes.nextNode
    i = i + 1
Wend
Call MsgBox("Failed to find Model", vbOKOnly, "Error")
DelModelFrm.Hide
End Sub
```

D.3.5 RDT1.frm



This is the main form of the application which is loaded when the application starts.

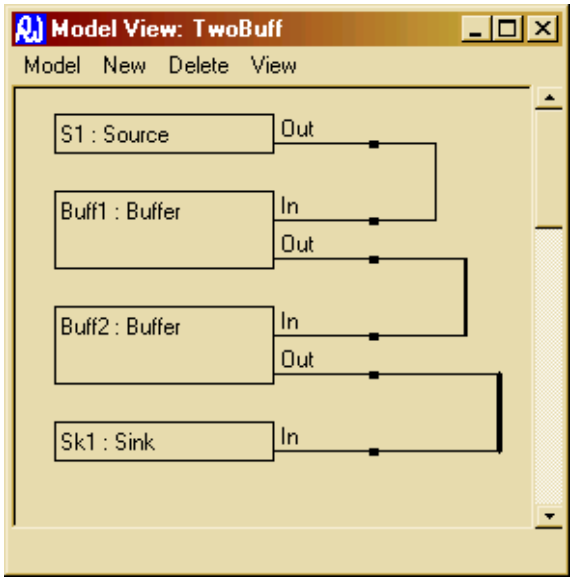



```
Option Explicit

Private Sub DelEvent_Click()
Call DelEventFrm.Show(vbModal)
Dim i As Integer
For i = Forms.Count - 1 To 0 Step -1
   Forms(i).Refresh
Next
WebBrowser1.Refresh
End Sub

Private Sub Exit_Click()
Call Form_Unload(1)
End Sub

Private Sub Form_Load()
'Initialise things:
'Make a minimal XML document
Set Module1.doc = New DOMDocument
Set Module1.el = Module1.doc.createElement("Model")
Call Module1.doc.appendChild(el)
'Set the name of the temp file...
```

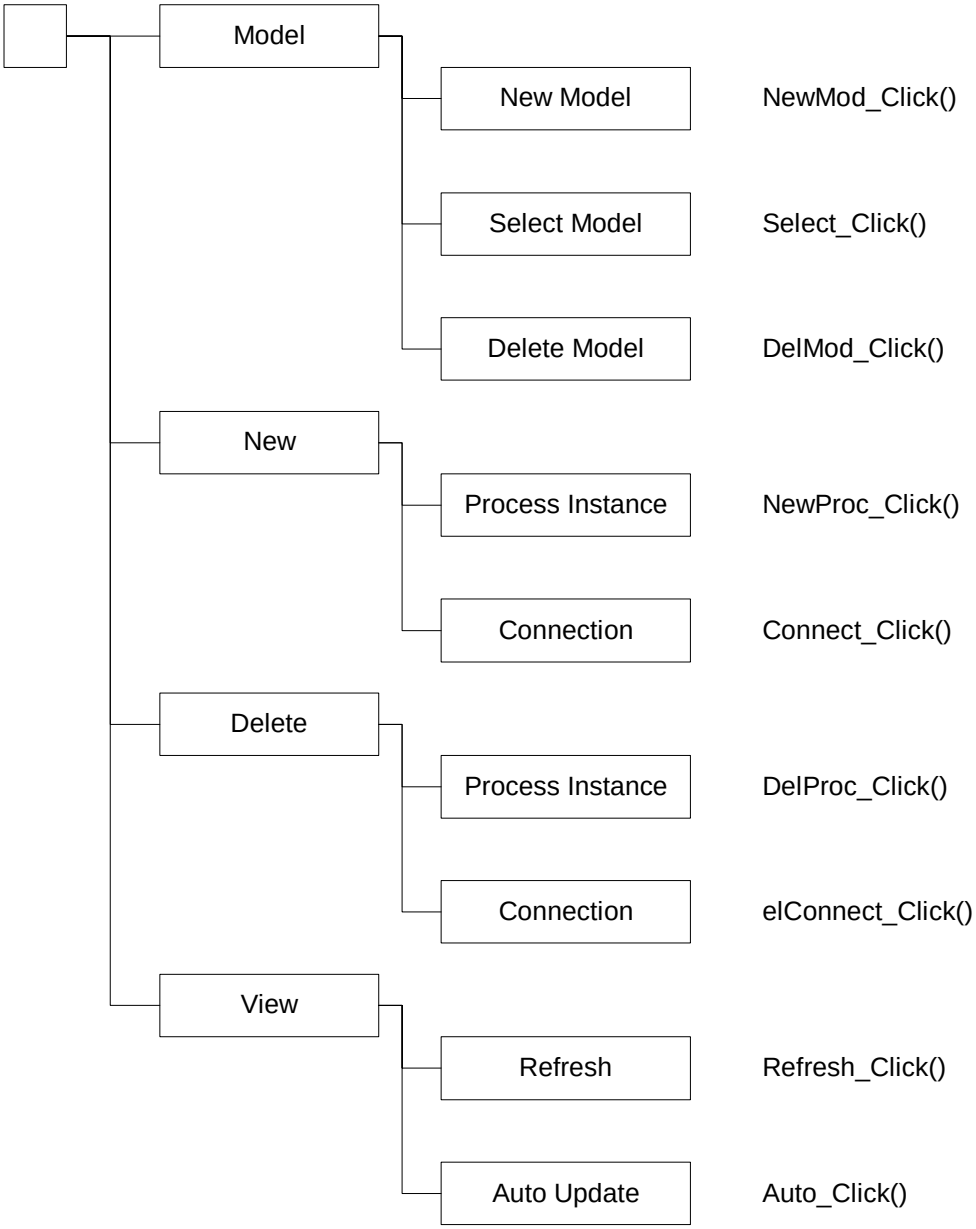
```
Module1.tempfile = App.Path & "\temp.xml"
Set Module1.fileSysObj = CreateObject("Scripting.FileSystemObject")
'Write the minimal XML into the tempfile
Set Module1.txtStream = Module1.fileSysObj.OpenTextFile(tempfile, 2, True)
Module1.txtStream.Write doc.xml
Module1.txtStream.Close
Set Module1.txtStream = Nothing

'Navigate browser to our temporary file
If Module1.fileSysObj.FileExists(Module1.tempfile) = False Then
    Module1.fileSysObj.CreateTextFile tempfile, True
    Set txtStream = fileSysObj.OpenTextFile(tempfile, 2, True)
    txtStream.Close
    Set txtStream = Nothing
End If
'Attempt to stop failures!
On Error Resume Next
WebBrowser1.Navigate Module1.tempfile
End Sub

Private Sub Form_Resize()
'Adjust the size of the web browser to match
If Height > 701 Then WebBrowser1.Height = Height - 700 Else WebBrowser1.Height =
100
If width > 101 Then WebBrowser1.width = width - 100 Else WebBrowser1.width = 100
End Sub

Private Sub Form_Unload(Cancel As Integer)
'Hit it hard!
End

End Sub

Private Sub Image_Click()
SelProcFrm.Show (vbModal)
SelProcFrm.Hide
Dim f As Form
If SelProcFrm.Selection <> "" Then
    Module1.DisProcess = SelProcFrm.Selection
    Set f = New ProcViewFrm
    f.Show (vbModeless)
End If
'Module1.DisProcess = SelProcFrm.Selection
'Picture1.Cls
'Call Module1.Draw(Picture1, Module1.DisProcess)
End Sub

Private Sub Model_Click()
    If Module2.CurrentModel = "" Then Call SelModelFrm.Show(vbModal)
    ModelViewFrm.Show (vbModeless)
End Sub

Private Sub NewEvent_Click()
Call NewEventForm.Show(vbModal)
'Call UpdateXMLView
Dim i As Integer
For i = Forms.Count - 1 To 0 Step -1
   Call Forms(i).Refresh
Next
WebBrowser1.Refresh
End Sub

Private Sub Open_Click()
Dim arr As Variant

CommonDialog1.CancelError = True
On Error GoTo ErrHandler
CommonDialog1.Flags = cdlOFNHideReadOnly
CommonDialog1.Filter = "All Files (*.*)|*.*|XML Files(*.xml)|*.xml"
CommonDialog1.FilterIndex = 2
CommonDialog1.ShowOpen
'Read the file into doc
Set Module1.txtStream = fileSysObj.OpenTextFile(CommonDialog1.filename, 1)
```

```
Module1.doc.loadXML (Module1.txtStream.ReadAll)
Module1.txtStream.Close
'Write the new file to tempfile
Set Module1.txtStream = fileSysObj.OpenTextFile(Module1.tempfile, 2)
Module1.txtStream.Write Module1.doc.xml
Module1.txtStream.Close
Set Module1.txtStream = Nothing
'Reflect the changes in the other window (if it is visible)
If XMLViewForm.Visible = True Then Call XMLViewForm.UpdateXMLView

'As a default, select any process in the file:
Set Module1.nodes = Module1.doc.getElementsByTagName("Model")
Set Module1.el = Module1.nodes.Item(0)
Set Module1.nodes = Module1.el.getElementsByTagName("Process")
Set Module1.el = Module1.nodes.nextNode
Module1.DisProcess = Module1.el.getAttribute("Name")

arr = Split(CommonDialog1.filename, "\")
Form1.Caption = "RDT: " & arr(UBound(arr))
Call UpdateXMLView
Exit Sub

ErrHandler:
  'User pressed the Cancel button
  'Call Module1.Draw(Picture1, DisProcess)
  Exit Sub
End Sub

Private Sub Refresh_Click()
End Sub

Private Sub Save_Click()
'Write the new file to tempfile
If Module1.filename = "" Then
    Call SaveAs_Click
    Exit Sub
End If
Set Module1.txtStream = fileSysObj.OpenTextFile(Module1.filename, 2)
Module1.txtStream.Write Module1.doc.xml
Module1.txtStream.Close
Set Module1.txtStream = Nothing
End Sub

Private Sub SaveAs_Click()
Dim s As String
Dim arr As Variant

CommonDialog1.ShowSave
If CommonDialog1.filename <> "" Then
    Module1.filename = CommonDialog1.filename
    Set Module1.txtStream = fileSysObj.OpenTextFile(Module1.filename, 2, True)
    Module1.txtStream.Write Module1.doc.xml
    Module1.txtStream.Close
    Set Module1.txtStream = Nothing
End If
arr = Split(CommonDialog1.filename, "\")
Form1.Caption = "RDT: " & arr(UBound(arr))
End Sub

Private Sub XML_Click()
    XMLViewForm.Show (vbModeless)
End Sub


Public Sub UpdateXMLView()
'Here code for ensuring the browser displays the latest version of the model
DoEvents
If WebBrowser1.width > 200 And WebBrowser1.Height > 300 And
WebBrowser1.LocationName <> "" Then WebBrowser1.Refresh
'If WebBrowser1.LocationName <> "" Then WebBrowser1.Refresh
End Sub
```

D.3.6 ModelView.frm



This is the window in which the user constructs models for execution from processes which they have already defined.



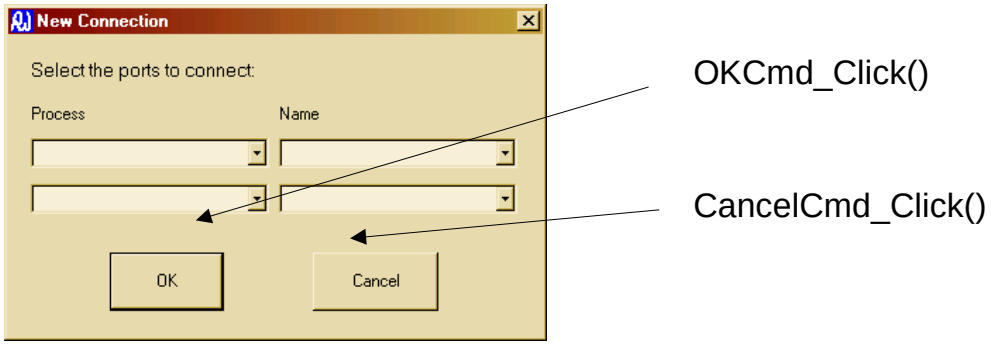

```
Option Explicit

Private Sub Auto_Click()
If Auto.Checked = True Then
    Auto.Checked = False
End If
End Sub

Private Sub Connect_Click()
If Module2.CurrentModel = "" Then Call SelModelFrm.Show(vbModal)
If Module2.CurrentModel = "" Then
    Call MsgBox("No model selected", vbOKOnly, "Error")
    Exit Sub
End If
NewConnectionFrm.Show
End Sub

Private Sub DelConnect_Click()
If Module2.CurrentModel = "" Then Call SelModelFrm.Show(vbModal)
If Module2.CurrentModel = "" Then
```

```
    Call MsgBox("No model selected", vbOKOnly, "Error")
    Exit Sub
End If
Call DelConnFrm.Show(vbModal)
End Sub

Private Sub DelMod_Click()
DelModelFrm.Show
End Sub

Private Sub DelProc_Click()
If Module2.CurrentModel = "" Then Call SelModelFrm.Show(vbModal)
If Module2.CurrentModel = "" Then
    Call MsgBox("No model selected", vbOKOnly, "Error")
    Exit Sub
End If
Delinstance.Show
End Sub

Private Sub Form_Activate()
If Module2.CurrentModel <> "" Then Call Refresh_Click
End Sub

Private Sub Form_Deactivate()
Form1.Model.Checked = False
End Sub

Private Sub Form_Paint()
If Auto.Checked = True And Module2.CurrentModel <> "" Then
    Caption = "Model View: " & CurrentModel
    Call Picture1.Cls
    Call Module2.Draw(Picture1)
End If
End Sub

Private Sub Form_Resize()
'Adjust the size of the picture window to match
DoEvents

If Height > 1001 Then PictureX.Height = Height - 1000 Else PictureX.Height = 200
If width > 351 Then PictureX.width = width - 350 Else PictureX.width = 200
VScroll1.Height = PictureX.Height
VScroll1.Left = PictureX.Left + PictureX.width
HScroll1.Top = PictureX.Top + PictureX.Height
HScroll1.width = PictureX.width

If Picture1.width > PictureX.width Then
    HScroll1.Visible = True
    HScroll1.SmallChange = Picture1.width / 20
    HScroll1.LargeChange = Picture1.width / 10
    HScroll1.Max = Picture1.width - PictureX.width
Else
    HScroll1.Visible = False
End If

If Picture1.Height > PictureX.Height Then
    VScroll1.Visible = True
    VScroll1.SmallChange = Picture1.Height / 20
    VScroll1.LargeChange = Picture1.Height / 10
    VScroll1.Max = Picture1.Height - PictureX.Height
Else
    VScroll1.Visible = False
End If
End Sub

Private Sub HScroll1_Change()
'If PictureX.width + HScroll1.Value + 200 > Picture1.width Then
'    Picture1.width = PictureX.width + HScroll1.Value + 500
'    Call Refresh_Click
'End If
Picture1.Left = -HScroll1.Value
End Sub
```

```
Private Sub NewMod_Click()
Dim s As String
'Get a new name
s = InputBox("Enter new Model name:", "New Model")
If s = "" Then Exit Sub
'Add this module to doc
Set Module1.nodes = Module1.doc.getElementsByTagName("Model")
Set Module1.el = Module1.doc.createElement("Instance")
Call el.setAttribute("Name", s)
Call nodes.Item(0).appendChild(el)

'Write the XML into the tempfile
Set Module1.txtStream = Module1.fileSysObj.OpenTextFile(tempfile, 2)
Module1.txtStream.Write doc.xml
Module1.txtStream.Close
Set Module1.txtStream = Nothing
If XMLViewForm.Visible = True Then Call XMLViewForm.UpdateXMLView

'When a new model is created, show it!
Module2.CurrentModel = s
Call Picture1.Cls
Picture1.width = 600
Picture1.Height = 600
Call Module2.Draw(Picture1)


End Sub

Private Sub NewProc_Click()
Dim s As String
Dim i As Integer

If Module2.CurrentModel = "" Then Call SelModelFrm.Show(vbModal)
If Module2.CurrentModel = "" Then
    Call MsgBox("No model selected", vbOKOnly, "Error")
    Exit Sub
End If

NewProcInstFrm.Show (vbModal)
's = InputBox("Enter a name for the new process", "Enter Name")
'If s = "" Then Exit Sub
'SelProcFrm.Show (vbModal)
If NewProcInstFrm.Selection = "" Then Exit Sub



Set Module1.el = Module1.doc.createElement("ProcInstance")
Call el.setAttribute("Name", NewProcInstFrm.ProcName)
Call el.setAttribute("Type", NewProcInstFrm.Selection)

Set Module1.nodes = Module1.doc.getElementsByTagName("Instance")
i = 0
Set Module1.el2 = Module1.nodes.nextNode
While Module2.CurrentModel <> Module1.el2.getAttribute("Name") And i <
Module1.nodes.length
    Set Module1.el2 = Module1.nodes.nextNode
    i = i + 1
Wend
Call el2.appendChild(el)

'Write the XML into the tempfile
Set Module1.txtStream = Module1.fileSysObj.OpenTextFile(tempfile, 2)
Module1.txtStream.Write doc.xml
Module1.txtStream.Close
Set Module1.txtStream = Nothing
If XMLViewForm.Visible = True Then Call XMLViewForm.UpdateXMLView
End Sub

Private Sub Refresh_Click()
If Module2.CurrentModel <> "" Then Caption = "Model View: " & CurrentModel
Call Module2.Draw(Picture1)
End Sub
```

```
Private Sub Select_Click()
Call SelModelFrm.Show(vbModal)
If Auto.Checked = True And Module2.CurrentModel <> "" Then
    Call Picture1.Cls
    Picture1.width = 600
    Picture1.Height = 600
    Call Module2.Draw(Picture1)
End If
End Sub

Private Sub VScroll1_Change()
'If PictureX.Height + VScroll1.Value + 200 > Picture1.Height Then
'    Picture1.Height = PictureX.Height + VScroll1.Value + 500
'    Call Refresh_Click
'End If
Picture1.Top = -VScroll1.Value
End Sub
```

D.3.7 NewConnectionFrm.frm

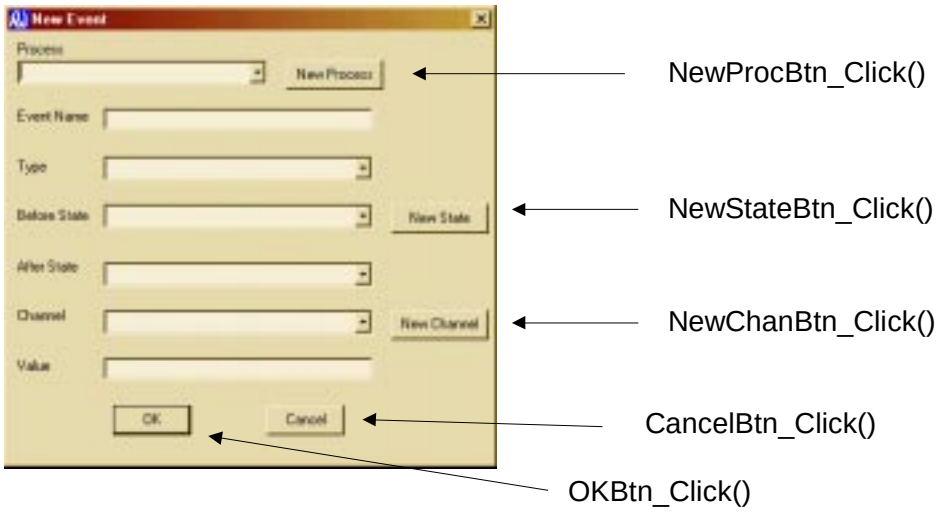



```
Option Explicit

Private Sub CancelCmd_Click()
NewConnectionFrm.Hide
End Sub

Private Sub Combo1_Click()
Dim i As Integer
Dim j As Integer
Combo2.Clear
Set Module1.nodes = Module1.doc.getElementsByTagName("Model")
Set Module1.el = Module1.nodes.Item(0)
Set Module1.nodes = Module1.el.getElementsByTagName("Process")

Module1.nodes.Reset
Set Module1.el = Module1.nodes.nextNode

i = 0
While i < Module1.nodes.length
    If List1.List(Combo1.ListIndex) = Module1.el.getAttribute("Name") Then
        Set Module1.nodes2 = Module1.el.getElementsByTagName("Event")
        Set Module1.el2 = Module1.nodes2.nextNode
        j = 0
        While j < Module1.nodes2.length
            If InList(Combo2, Module1.el2.getAttribute("Channel")) = False Then
Combo2.AddItem (Module1.el2.getAttribute("Channel"))
            Set Module1.el2 = Module1.nodes2.nextNode
            j = j + 1
        Wend
    End If
    Set Module1.el = Module1.nodes.nextNode
    i = i + 1
Wend
End Sub
```

```
Private Function InList(c As ComboBox, s As String) As Boolean
'True if s is already in the combobox list
Dim i As Integer
i = 0
While i < c.ListCount
    If s = c.List(i) Then
        InList = True
        Exit Function
    End If
i = i + 1
Wend
InList = False
End Function

Private Sub Combo3_Click()
Dim i As Integer
Dim j As Integer
Combo4.Clear
Set Module1.nodes = Module1.doc.getElementsByTagName("Model")
Set Module1.el = Module1.nodes.Item(0)
Set Module1.nodes = Module1.el.getElementsByTagName("Process")

Module1.nodes.Reset
Set Module1.el = Module1.nodes.nextNode

i = 0
While i < Module1.nodes.length
    If List1.List(Combo3.ListIndex) = Module1.el.getAttribute("Name") Then
        Set Module1.nodes2 = Module1.el.getElementsByTagName("Event")
        Set Module1.el2 = Module1.nodes2.nextNode
        j = 0
        While j < Module1.nodes2.length
            If InList(Combo4, Module1.el2.getAttribute("Channel")) = False Then
Combo4.AddItem (Module1.el2.getAttribute("Channel"))
            Set Module1.el2 = Module1.nodes2.nextNode
            j = j + 1
        Wend
    End If
    Set Module1.el = Module1.nodes.nextNode
    i = i + 1
Wend

End Sub

Private Sub Form_Activate()
Dim i As Integer
Dim j As Integer

Combo1.Clear
Combo2.Clear
Combo3.Clear
Combo4.Clear
List1.Clear

'List the processes in combos 1, 3
Set Module1.nodes = Module1.doc.getElementsByTagName("Model")
Set Module1.el = Module1.nodes.Item(0)
Set Module1.nodes = Module1.el.getElementsByTagName("ProcInstance")

Module1.nodes.Reset
Set Module1.el = Module1.nodes.nextNode

i = 0
While i < Module1.nodes.length
    Combo1.AddItem (Module1.el.getAttribute("Name"))
    Combo3.AddItem (Module1.el.getAttribute("Name"))
    List1.AddItem (Module1.el.getAttribute("Type"))
    Set Module1.el = Module1.nodes.nextNode
    i = i + 1
Wend
End Sub

Private Sub OKCmd_Click()
```

```
Dim i As Integer

'Then add the XML...
Set Module1.el = Module1.doc.createElement("Connection")
Set Module1.el2 = Module1.doc.createElement("End")
Call el2.setAttribute("ProcInstance", Combo1.List(Combo1.ListIndex))
Call el2.setAttribute("Channel", Combo2.List(Combo2.ListIndex))
Call el.appendChild(el2)
Set Module1.el2 = Module1.doc.createElement("End")
Call el2.setAttribute("ProcInstance", Combo3.List(Combo3.ListIndex))
Call el2.setAttribute("Channel", Combo4.List(Combo4.ListIndex))
Call el.appendChild(el2)

Set Module1.nodes = Module1.doc.getElementsByTagName("Instance")
i = 0
Set Module1.el2 = Module1.nodes.nextNode
While Module2.CurrentModel <> Module1.el2.getAttribute("Name") And i <
Module1.nodes.length
    Set Module1.el2 = Module1.nodes.nextNode
    i = i + 1
Wend
Call el2.appendChild(el)

'Write the XML into the tempfile
Set Module1.txtStream = Module1.fileSysObj.OpenTextFile(tempfile, 2)
Module1.txtStream.Write doc.xml
Module1.txtStream.Close
Set Module1.txtStream = Nothing
If XMLViewForm.Visible = True Then Call XMLViewForm.UpdateXMLView

NewConnectionFrm.Hide
End Sub
```

D.3.8 NewEventFrom.frm



This is the dialogue box with which the user defines new events in processes.

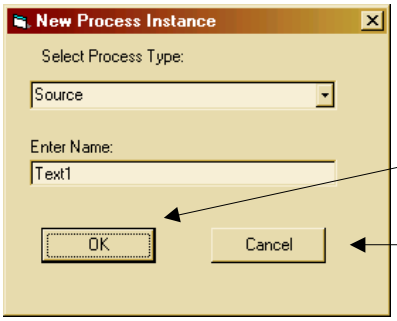
```
Option Explicit

Private Sub CancelBtn_Click()
NewEventForm.Hide
End Sub

Private Sub Command4_Click()
End Sub

Private Sub Combo1_Click()
```

```
Dim i As Integer
Dim j As Integer

Combo3.Clear
Combo4.Clear
Combo5.Clear

Set Module1.nodes = Module1.doc.getElementsByTagName("Model")
Set Module1.el = Module1.nodes.Item(0)
Set Module1.nodes = Module1.el.getElementsByTagName("Process")

Module1.nodes.Reset
Set Module1.el = Module1.nodes.nextNode

i = 0
While i < Module1.nodes.length
    If Combo1.List(Combo1.ListIndex) = Module1.el.getAttribute("Name") Then
        Set Module1.nodes2 = Module1.el.getElementsByTagName("Event")
        Set Module1.el2 = Module1.nodes2.nextNode
        j = 0
        While j < Module1.nodes2.length
            If InList(Combo3, Module1.el2.getAttribute("Before")) = False Then
Combo3.AddItem (Module1.el2.getAttribute("Before"))
            If InList(Combo3, Module1.el2.getAttribute("After")) = False Then
Combo3.AddItem (Module1.el2.getAttribute("After"))
            If InList(Combo4, Module1.el2.getAttribute("Before")) = False Then
Combo4.AddItem (Module1.el2.getAttribute("Before"))
            If InList(Combo4, Module1.el2.getAttribute("After")) = False Then
Combo4.AddItem (Module1.el2.getAttribute("After"))
            If InList(Combo5, Module1.el2.getAttribute("Channel")) = False Then
Combo5.AddItem (Module1.el2.getAttribute("Channel"))
            Set Module1.el2 = Module1.nodes2.nextNode
            j = j + 1
        Wend
    End If
    Set Module1.el = Module1.nodes.nextNode
    i = i + 1
Wend
End Sub

Private Sub Form_Activate()
Dim i As Integer
Dim j As Integer

'Code here to clear the form
Combo1.Clear
Text1.Text = ""
Combo2.ListIndex = -1
Combo3.Clear
Combo4.Clear
Combo5.Clear
Text2.Text = ""

'List the processes in combo1
Set Module1.nodes = Module1.doc.getElementsByTagName("Model")
Set Module1.el = Module1.nodes.Item(0)
Set Module1.nodes = Module1.el.getElementsByTagName("Process")

Module1.nodes.Reset
Set Module1.el = Module1.nodes.nextNode

i = 0
While i < Module1.nodes.length
    Set Module1.nodes2 = Module1.el.getElementsByTagName("Event")
    Set Module1.el2 = Module1.nodes2.nextNode
'    j = 0
'    While j < Module1.nodes2.length
'        If InList(Combo3, Module1.el2.getAttribute("Before")) = False Then
Combo3.AddItem (Module1.el2.getAttribute("Before"))
'        If InList(Combo3, Module1.el2.getAttribute("After")) = False Then
Combo3.AddItem (Module1.el2.getAttribute("After"))
'        If InList(Combo4, Module1.el2.getAttribute("Before")) = False Then
Combo4.AddItem (Module1.el2.getAttribute("Before"))
```

```
'        If InList(Combo4, Module1.el2.getAttribute("After")) = False Then
Combo4.AddItem (Module1.el2.getAttribute("After"))
'        If InList(Combo5, Module1.el2.getAttribute("Channel")) = False Then
Combo5.AddItem (Module1.el2.getAttribute("Channel"))
'        Set Module1.el2 = Module1.nodes2.nextNode
'        j = j + 1
'    Wend
    Combo1.AddItem (Module1.el.getAttribute("Name"))
    Set Module1.el = Module1.nodes.nextNode
    i = i + 1
Wend
End Sub

Private Sub Form_Load()
Combo2.AddItem ("Read")
Combo2.AddItem ("Write")
Combo2.AddItem ("Create")
Combo2.ListIndex = 0
End Sub

Private Sub NewChanBtn_Click()
'Code here to add a new channel to the combo...
Dim s As String
s = InputBox("Enter the new Channel name", "New Channel")
If s = "" Then Exit Sub
Combo5.AddItem (s)
Combo5.ListIndex = Combo5.ListCount - 1
End Sub

Private Sub NewProcBtn_Click()
Dim s As String
'Get a new process name and add it to Combo1
s = InputBox("Enter new process name:", "New Process")
If s = "" Then Exit Sub
Combo1.AddItem (s)
Combo1.ListIndex = Combo1.ListCount - 1
'Add this process to doc
Set Module1.nodes = Module1.doc.getElementsByTagName("Model")
Set Module1.el = Module1.doc.createElement("Process")
Call el.setAttribute("Name", s)
Call nodes.Item(0).appendChild(el)
End Sub

Private Sub NewStateBtn_Click()
'Code here to add a new state to the combo box...
Dim s As String
s = InputBox("Enter the new state name", "New State")
If s = "" Then Exit Sub
Combo3.AddItem (s)
Combo4.AddItem (s)
End Sub

Private Sub OKBtn_Click()
Dim i As Integer
'Code here to write the new event into the XML...
'First check to see that all required values are there...
If Combo1.List(Combo1.ListIndex) = "" Then
    Call MsgBox("A Process name is required", vbOKOnly, "Error")
    Exit Sub
End If


If Text1.Text = "" Then
    Call MsgBox("This event Must have a name", vbOKOnly, "Error")
    Exit Sub
End If

'See if the name already appears in the process....
If Module1.NewName(Text1.Text, Combo1.Text) = False Then
    If MsgBox("That event name is already in use.  Use it again?", vbYesNo,
"Error") = vbNo Then
    Exit Sub
    End If
```

```
End If

If Combo2.List(Combo2.ListIndex) = "" Then
    Call MsgBox("A Process type is required", vbOKOnly, "Error")
    Exit Sub
End If
If Combo3.List(Combo3.ListIndex) = "" Then
    Call MsgBox("A Before State is required", vbOKOnly, "Error")
    Exit Sub
End If
If Combo4.List(Combo4.ListIndex) = "" Then
    Call MsgBox("An After State is required", vbOKOnly, "Error")
    Exit Sub
End If
If Combo5.List(Combo5.ListIndex) = "" Then
    Call MsgBox("A Channel name is required", vbOKOnly, "Error")
    Exit Sub
End If
If Text2.Text = "" Then
    Call MsgBox("A Value is required", vbOKOnly, "Error")
    Exit Sub
End If

'Then add a new event node to the XML...
Set Module1.el = Module1.doc.createElement("Event")
Call el.setAttribute("Name", Text1.Text)
Call el.setAttribute("Type", Combo2.List(Combo2.ListIndex))
Call el.setAttribute("Before", Combo3.List(Combo3.ListIndex))
Call el.setAttribute("After", Combo4.List(Combo4.ListIndex))
Call el.setAttribute("Channel", Combo5.List(Combo5.ListIndex))
Call el.setAttribute("Value", Text2.Text)

Set Module1.nodes = Module1.doc.getElementsByTagName("Process")
i = 0
Set Module1.el2 = Module1.nodes.nextNode
While Combo1.List(Combo1.ListIndex) <> Module1.el2.getAttribute("Name") And i <
Module1.nodes.length
    Set Module1.el2 = Module1.nodes.nextNode
    i = i + 1
Wend
Call el2.appendChild(el)

'Write the XML into the tempfile
Set Module1.txtStream = Module1.fileSysObj.OpenTextFile(tempfile, 2)
Module1.txtStream.Write doc.xml
Module1.txtStream.Close
Set Module1.txtStream = Nothing
If XMLViewForm.Visible = True Then Call XMLViewForm.UpdateXMLView
NewEventForm.Hide
End Sub

Private Function InList(c As ComboBox, s As String) As Boolean
'True if s is already in the combobox list
Dim i As Integer
i = 0
While i < c.ListCount
    If s = c.List(i) Then
        InList = True
        Exit Function
    End If
i = i + 1
Wend
InList = False
End Function

Private Sub Text1_LostFocus()
'See if this name is already in use in the process
'If Module1.NewName(Text1.Text, Combo1.Text) = False Then
'    Call MsgBox("That event name is already in use in this model", vbOKOnly,
"Error")
'End If
End Sub
```

D.3.9 NewProcInstFrm.frm

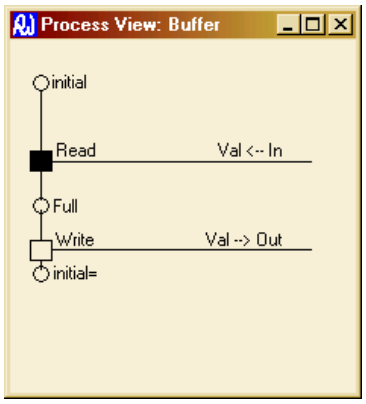


OKButton_Click()

CancelButton_Click()

```
Option Explicit
Public Selection As String
Public ProcName As String

Private Sub CancelButton_Click()
Selection = ""
ProcName = ""
NewProcInstFrm.Hide
End Sub

Private Sub Form_Activate()
'Fill the Combo Box
Dim i As Integer

Selection = ""
ProcName = ""
Combo1.Clear
Set Module1.nodes = Module1.doc.getElementsByTagName("Model")
Set Module1.el = Module1.nodes.Item(0)
Set Module1.nodes = Module1.el.getElementsByTagName("Process")

Module1.nodes.Reset
Set Module1.el = Module1.nodes.nextNode
i = 0
While i < Module1.nodes.length
    Combo1.AddItem (Module1.el.getAttribute("Name"))
    Set Module1.el = Module1.nodes.nextNode
    i = i + 1
Wend
If Combo1.ListCount > 0 Then Combo1.ListIndex = 0
End Sub

Private Sub OKButton_Click()
Selection = Combo1.List(Combo1.ListIndex)
ProcName = Text1.Text
NewProcInstFrm.Hide
End Sub
```

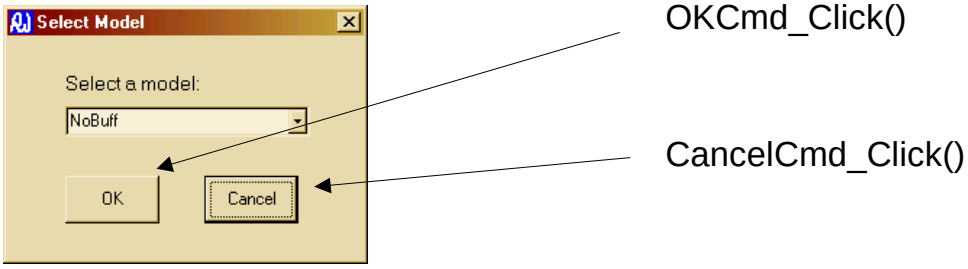
D.3.10ProcViewFrm.frm



This is the form in which the application draws diagrams of the users processes.

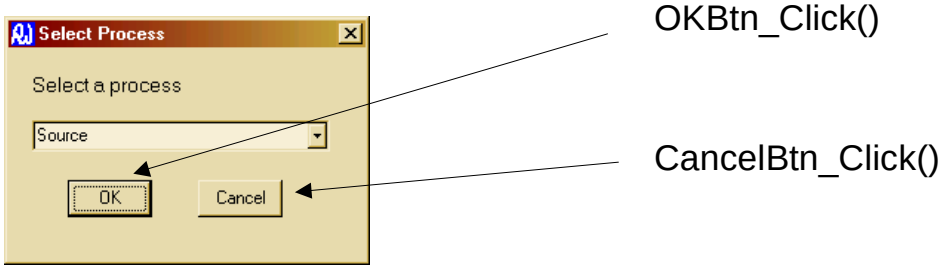
```
Option Explicit

Dim DisProcess As String

Private Sub Form_Load()
DisProcess = Module1.DisProcess
Caption = Caption & ": " & DisProcess
End Sub

Private Sub Form_Paint()
    Call Picture1.Cls
    Call Module1.Draw(Picture1, DisProcess)
End Sub

Private Sub Form_Resize()
''Adjust the size of the picture window to match

If Height > 901 Then PictureX.Height = Height - 500 Else PictureX.Height = 500
If width > 801 Then PictureX.width = width - 350 Else PictureX.width = 500
VScroll1.Height = PictureX.Height
VScroll1.Left = PictureX.Left + PictureX.width

Picture1.width = PictureX.width

If Picture1.Height > PictureX.Height Then
    VScroll1.Visible = True
    VScroll1.SmallChange = Picture1.Height / 20
    VScroll1.LargeChange = Picture1.Height / 10
    VScroll1.Max = Picture1.Height - PictureX.Height
Else
    VScroll1.Visible = False
End If
End Sub

Private Sub HScroll1_Change()
'Picture1.Left = -HScroll1.Value
End Sub

Private Sub VScroll1_Change()
Picture1.Top = -VScroll1.Value
End Sub
```

D.3.11SelModelFrm.frm




```
Option Explicit

Private Sub CancelCmd_Click()
Module2.CurrentModel = ""
SelModelFrm.Hide
End Sub

Private Sub Form_Activate()
Dim i As Integer
Combo1.Clear
Set Module1.nodes = Module1.doc.getElementsByTagName("Model")
Set Module1.el = Module1.nodes.Item(0)
Set Module1.nodes = Module1.el.getElementsByTagName("Instance")
Module1.nodes.Reset
Set Module1.el = Module1.nodes.nextNode
i = 0
While i < Module1.nodes.length
    Combo1.AddItem (Module1.el.getAttribute("Name"))
    Set Module1.el = Module1.nodes.nextNode
    i = i + 1
Wend
If Combo1.ListCount > 0 Then Combo1.ListIndex = 0
End Sub

Private Sub OKCmd_Click()
If Combo1.List(Combo1.ListIndex) = "" Then Exit Sub
Module2.CurrentModel = Combo1.List(Combo1.ListIndex)
SelModelFrm.Hide
End Sub
```

D.3.12SelProcFrm.frm




```
Option Explicit
Public Selection As String

Private Sub CancelBtn_Click()
Selection = ""
```

```
SelProcFrm.Hide
End Sub

Private Sub Form_Activate()
Dim i As Integer

Combo1.Clear
Set Module1.nodes = Module1.doc.getElementsByTagName("Model")
Set Module1.el = Module1.nodes.Item(0)
Set Module1.nodes = Module1.el.getElementsByTagName("Process")

Module1.nodes.Reset
Set Module1.el = Module1.nodes.nextNode
i = 0
While i < Module1.nodes.length
    Combo1.AddItem (Module1.el.getAttribute("Name"))
    Set Module1.el = Module1.nodes.nextNode
    i = i + 1
Wend
If Combo1.ListCount > 0 Then Combo1.ListIndex = 0
End Sub

Private Sub OKBtn_Click()
Selection = Combo1.List(Combo1.ListIndex)
SelProcFrm.Hide
End Sub
```

D.3.13RDTModule1.bas

```
'This module contains DOMDocument code
'Also contains stuff relating to drawing pictures of processes

Option Explicit
Public tempfile As String 'The name of the file viewed by the browser object
Public fileSysObj As Object
Public txtStream As Object

Public filename As String 'The name of the "real" file in use

Public doc As DOMDocument 'Used to store the present state of the model
Public el As IXMLDOMElement
Public el2 As IXMLDOMElement
Public nodes As IXMLDOMNodeList
Public nodes2 As IXMLDOMNodeList
Public node As IXMLDOMNode

Public DisProcess As String
Public ProcViews(20) As Form

Const box = 200
Const circ = 75
Const width = 1200
Const abit = 300
Const Xstart = 300
Const Ystart = 300


Public Type EventType
Name As String
Type As String
Before As String
After As String
Channel As String
Value As String
Drawn As Boolean
End Type

Public Type ThreadType
X As Integer
```

```
Y As Integer
State As String
InUse As Boolean
Joined As Boolean
Forked As Boolean
End Type

Public Type StateType
Name As String
Befores As String
Afters As String
End Type

Public Function NewName(n As String, p As String) As Boolean
'Returns true if n is not already the name of an event in process p
'If p = "" then searches whole model

Dim i As Integer
Dim j As Integer
i = 0
j = 0
Dim tmp As String

NewName = True

If p = "" Then
    Set nodes = doc.getElementsByTagName("Model")
    Set el = nodes.Item(0)
    Set nodes2 = el.getElementsByTagName("Process")

    nodes2.Reset
    Set el2 = nodes.nextNode
    i = 0
    j = 0
    While j < nodes2.length
        Set el2 = nodes2.nextNode
        Set nodes = el2.getElementsByTagName("Event")
        i = 0
        While i < nodes.length
            Set el = nodes(i)
            If el.getAttribute("Name") = n Then
                NewName = False
                Exit Function
            End If
            i = i + 1
        Wend
        j = j + 1
    Wend
    ' Didn't find it
    NewName = True
    Exit Function
Else
    Set nodes = doc.getElementsByTagName("Model")
    Set el = nodes.Item(0)
    Set nodes = el.getElementsByTagName("Process")

    nodes.Reset
    Set el = nodes.nextNode
    i = 0

    While i < nodes.length And el.getAttribute("Name") <> p
        Set el = nodes.nextNode
        i = i + 1
    Wend

    Set nodes = el.getElementsByTagName("Event")
    i = 0
    While i < nodes.length
        Set el = nodes(i)
        If el.getAttribute("Name") = n Then
            NewName = False
            Exit Function
        End If
```

```
        i = i + 1
    Wend
    ' Didn't find it
    NewName = True
    Exit Function
End If
End Function

Public Sub Draw(p As PictureBox, process As String)
Dim events() As EventType
Dim threads() As ThreadType
Dim states() As StateType
Dim threads_size As Integer
Dim states_size As Integer
Dim i As Integer
Dim j As Integer
Dim k As Integer
Dim n As Integer
Dim c As Integer

Dim breaker As Integer
Dim done As Boolean
Dim pos As Integer

ReDim events(20)
ReDim threads(30)
ReDim states(20)

'Read details of this process into the arrays
Set nodes = doc.getElementsByTagName("Model")
Set el = nodes.Item(0)
Set nodes = el.getElementsByTagName("Process")

nodes.Reset
Set el = nodes.nextNode
i = 0
While i < nodes.length And el.getAttribute("Name") <> process
    Set el = nodes.nextNode
    i = i + 1
Wend

Set nodes = el.getElementsByTagName("Event")
nodes.Reset
Set el = nodes.nextNode
i = 0
While i < nodes.length
    If i >= UBound(events) Then ReDim Preserve events(UBound(events) + 20)
    events(i).Name = el.getAttribute("Name")
    events(i).Before = el.getAttribute("Before")
    events(i).After = el.getAttribute("After")
    'See if the state names are in that array already, add them if not
    j = 0
    While states(j).Name <> events(i).Before And states(j).Name <> ""
        j = j + 1
    Wend
    If states(j).Name = events(i).Before Then
        states(j).Befores = Val(states(j).Befores) + 1
    Else
        If j >= UBound(states) Then ReDim Preserve states(UBound(states) + 20)
        states(j).Name = el.getAttribute("Before")
        states(j).Befores = 1

    End If

    j = 0
    While states(j).Name <> events(i).After And states(j).Name <> ""
        j = j + 1
    Wend
    If states(j).Name = events(i).After Then
        states(j).Afters = Val(states(j).Afters) + 1
    Else
        If j >= UBound(states) Then ReDim Preserve states(UBound(states) + 20)
        states(j).Name = el.getAttribute("After")
```

```
        states(j).Afters = 1
    End If

    events(i).Channel = el.getAttribute("Channel")
    events(i).Type = el.getAttribute("Type")
    events(i).Value = el.getAttribute("Value")
    events(i).Drawn = False
    Set el = nodes.nextNode
    i = i + 1
Wend

'Draw the picture
'Start off by finding initial and drawing a fork
i = 0
While states(i).Name <> "initial"
    i = i + 1
Wend

p.Circle (Xstart, Ystart), circ
p.Line (Xstart, Ystart + circ)-(Xstart, Ystart + circ + abit)
p.CurrentX = Xstart + 100
p.CurrentY = Ystart - circ - 25
p.Print ("initial")
threads(0).State = "initial"
threads(0).X = Xstart
threads(0).Y = Ystart + circ + abit
threads(0).InUse = True
threads(0).Forked = True
pos = 375 + abit

j = 1
While j < states(i).Befores
    p.Line (300 + ((j - 1) * width), 375 + abit)-(300 + (j * width), 375 + abit)
    p.Line (300 + (j * width), 375 + abit)-(300 + (j * width), 375 + 2 * abit)
    threads(j).InUse = True
    threads(j).Forked = True
    threads(j).State = "initial"
    threads(j).X = 300 + (j * width)
    threads(j).Y = 375 + 2 * abit
    j = j + 1
Wend

breaker = 0
done = False
While breaker < 1000 And done = False
    j = 0
    done = True
    While j < UBound(threads)
        If threads(j).InUse = True Then 'look for an event to draw
            k = 0
            Do While k < UBound(events)

            If events(k).Before = threads(j).State And events(k).Drawn <> True
Then
                pos = pos + abit '+ 2 * abit
                p.Line (threads(j).X, threads(j).Y)-(threads(j).X, pos)
                threads(j).Y = pos
                'Working code to enlarge the picturebox if required...
                If p.Height < (pos + 500) Then p.Height = pos + 500

                If events(k).Type = "Write" Then
                    p.Line (threads(j).X - box / 2, threads(j).Y)-(threads(j).X +
box / 2, threads(j).Y + box), , B
                    threads(j).Y = threads(j).Y + box
                    threads(j).Joined = False
                    threads(j).Forked = False
                    p.CurrentX = p.CurrentX + 50
                    p.CurrentY = p.CurrentY - 300
                    p.Print (events(k).Name)
                    p.Line (threads(j).X + box / 2, threads(j).Y - box / 2)-
(p.width - 200, threads(j).Y - box / 2)
                    'p.CurrentX = p.width - p.TextWidth(events(k).Channel & " : "
& events(k).Value) - 500
```

```
                    p.CurrentX = p.width - p.TextWidth(events(k).Value & " --> " &
events(k).Channel) - 500
                    p.CurrentY = p.CurrentY - 200
                    'p.Print events(k).Channel & " : " & events(k).Value
                    p.Print events(k).Value & " --> " & events(k).Channel
                End If
                If events(k).Type = "Read" Then
                    p.Line (threads(j).X - box / 2, threads(j).Y)-(threads(j).X +
box / 2, threads(j).Y + box), , BF
                    threads(j).Y = threads(j).Y + box
                    threads(j).Joined = False
                    threads(j).Forked = False
                    p.CurrentX = p.CurrentX + 50
                    p.CurrentY = p.CurrentY - 300
                    p.Print (events(k).Name)
                    p.Line (threads(j).X + box / 2, threads(j).Y - box / 2)-
(p.width - 200, threads(j).Y - box / 2)
                    'p.CurrentX = p.width - p.TextWidth(events(k).Channel & " : "
& events(k).Value) - 500
                    p.CurrentX = p.width - p.TextWidth(events(k).Value & " <-- " &
events(k).Channel) - 500
                    p.CurrentY = p.CurrentY - 200
                    'p.Print events(k).Channel & " : " & events(k).Value
                    p.Print events(k).Value & " <-- " & events(k).Channel
                End If
                If events(k).Type = "Create" Then
                    p.Line (threads(j).X - box / 2, threads(j).Y)-(threads(j).X +
box / 2, threads(j).Y + box)
                    p.Line (threads(j).X - box / 2, threads(j).Y + box)-
(threads(j).X + box / 2, threads(j).Y)
                    p.Line (threads(j).X - box / 2, threads(j).Y)-(threads(j).X +
box / 2, threads(j).Y + box), , B
                    threads(j).Y = threads(j).Y + box
                    threads(j).Joined = False
                    threads(j).Forked = False
                    p.CurrentX = p.CurrentX + 50
                    p.CurrentY = p.CurrentY - 300
                    p.Print (events(k).Name)
                    p.Line (threads(j).X + box / 2, threads(j).Y - box / 2)-
(p.width - 200, threads(j).Y - box / 2)
                    'p.CurrentX = p.width - p.TextWidth(events(k).Channel & " : "
& events(k).Value) - 500
                    p.CurrentX = p.width - p.TextWidth(events(k).Value & " --> " &
events(k).Channel) - 500
                    p.CurrentY = p.CurrentY - 200
                    'p.Print events(k).Channel & " : " & events(k).Value
                    p.Print events(k).Value & " --> " & events(k).Channel
               End If
                threads(j).State = events(k).After
                events(k).Drawn = True
                done = False
                Exit Do
            End If
            k = k + 1
            Loop
        End If
    j = j + 1
    Wend

'join threads
    j = 0
    While j < UBound(threads)
        If threads(j).InUse = True And threads(j).Joined = False Then 'consider a
join
            k = 0
            While k < UBound(states)
                If states(k).Name = threads(j).State Then
                    If Val(states(k).Afters) = 1 Then  'easy!
                        threads(j).Joined = True
                    Else 'See if there are enough others and do a join...

                        n = j '(only look at threads to our right)
                        c = 0
```

```
                        While n < UBound(threads)
                            If threads(n).State = threads(j).State Then c = c + 1
                            n = n + 1
                        Wend
                        If c = Val(states(k).Afters) Then 'have to do a join
                            pos = pos + abit
                            n = j + 1 '(only look at threads further along the
array)
                            While n < UBound(threads)
                                If threads(n).State = threads(j).State Then
                                    p.Line (threads(n).X, threads(n).Y)-
(threads(n).X, pos)
                                    threads(n).Y = pos
                                    p.Line (threads(n).X, pos)-(threads(j).X, pos)
                                    threads(n).InUse = False
                                End If
                                n = n + 1
                            Wend
                            threads(j).Joined = True
                            threads(j).Forked = False
                        End If
                    End If
                End If
                k = k + 1
            Wend
        End If
    j = j + 1
    Wend

'fork threads

    j = 0
    While j < UBound(threads)
        If threads(j).InUse = True And threads(j).Joined = True Then 'consider a
fork
            k = 0
            While k < UBound(states)
                If states(k).Name = threads(j).State And threads(j).Forked = False
Then
                'Working code to enlarge the picturebox if required...
                If p.Height < (pos + 1500) Then p.Height = pos + 1500

                    If Val(states(k).Befores) = 0 Then  'easy! - nothing to do
                        pos = pos + abit
                        p.Line (threads(j).X, threads(j).Y)-(threads(j).X, pos)
                        threads(j).Y = pos
                        p.Circle (threads(j).X, threads(j).Y + circ / 2), circ
                        p.CurrentX = p.CurrentX + 125
                        p.CurrentY = p.CurrentY - circ - 25
                        p.Print (threads(j).State)
                        threads(j).Y = threads(j).Y + circ + 5
                        threads(j).Forked = True
                        If threads(j).Y > pos Then pos = threads(j).Y
                    End If
                    If Val(states(k).Befores) = 1 Then  'still quite easy!
                        p.Line (threads(j).X, threads(j).Y)-(threads(j).X,
threads(j).Y + abit)
                        threads(j).Y = threads(j).Y + abit
                        p.Circle (threads(j).X, threads(j).Y + circ / 2 + 5), circ
                        p.CurrentX = p.CurrentX + 125
                        p.CurrentY = p.CurrentY - circ - 25
                        p.Print (threads(j).State)
                        threads(j).Y = threads(j).Y + circ + 5
                        threads(j).Forked = True
                        If threads(j).Y > pos Then pos = threads(j).Y
                    End If
                    If Val(states(k).Befores) > 1 Then
                        pos = pos + abit
                        p.Line (threads(j).X, threads(j).Y)-(threads(j).X, pos)
                        threads(j).Y = pos
                        p.Circle (threads(j).X, threads(j).Y + circ / 2), circ
                        p.CurrentX = p.CurrentX + 125
                        p.CurrentY = p.CurrentY - circ - 25
```

```
                        p.Print (threads(j).State)
                        threads(j).Y = threads(j).Y + circ + 5
                        threads(j).Forked = True
                        If threads(j).Y > pos Then pos = threads(j).Y
                        pos = pos + abit
                        p.Line (threads(j).X, threads(j).Y)-(threads(j).X, pos)
                        threads(j).Y = pos

                        n = 1
                        While n < states(k).Befores
                            'find a thread not already in use
                            c = 0
                            Do While c < UBound(threads)
                               If threads(c).InUse = False Then Exit Do
                               c = c + 1
                            Loop

                            If c = UBound(threads) Then
                                ReDim Preserve threads(UBound(threads) + 5)
                            End If
' >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
                            p.Line (300 + ((j) * width), pos)-(300 + (c * width),
pos)
                            p.Line (300 + (c * width), pos)-(300 + (c * width),
pos + abit)
                            threads(c).InUse = True
                            threads(c).State = threads(j).State
                            threads(c).X = 300 + (c * width)
                            threads(c).Y = pos + abit
                            threads(c).Forked = True
'                            pos = pos + abit  'Commented out 5/9/2001 to make
forks look nice
                            n = n + 1
' >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
                        Wend
                        If threads(j).Y > pos Then pos = threads(j).Y
                    End If
                End If
                k = k + 1
            Wend
        End If
        j = j + 1
    Wend
    breaker = breaker + 1
Wend

If p.Height < pos + 500 Then
    p.Height = pos + 1500
End If
End Sub
```

D.3.14RDTModule2.bas

```
'This module contains mostly stuff that draws models - instances and connections
Option Explicit

Private Type Chan
Name As String
X As Integer
Y As Integer
End Type

Private Type InstType
Name As String
Type As String
Channels() As Chan
Nch As Integer
End Type

Private el As IXMLDOMElement
Private el2 As IXMLDOMElement
```

```
Private nodes As IXMLDOMNodeList
Private nodes2 As IXMLDOMNodeList
Private node As IXMLDOMNode


Dim Instances() As InstType
Dim Types() As InstType


Public CurrentModel As String

Public Sub Draw(p As PictureBox)
Dim X As Integer
Dim Y As Integer
Dim i As Integer
Dim j As Integer
Dim k As Integer

Dim a As Integer
Dim b As Integer
Dim c As Integer
Dim d As Integer
Dim Xpos As Integer

'Clear off the old picture
p.Cls

'Draws a picture of the current "instance"
If CurrentModel = "" Then Exit Sub 'nothing to do

ReDim Types(20)
ReDim Instances(20)


'Read in the names of the types of processes and the names of the channels each
knows
Set nodes = Module1.doc.getElementsByTagName("Model")
Set el = nodes.Item(0)
Set nodes = el.getElementsByTagName("Process")

nodes.Reset
Set el = nodes.nextNode
i = 0
While i < nodes.length
    If i >= UBound(Types) Then ReDim Preserve Types(UBound(Types) + 10)
    Set nodes2 = el.getElementsByTagName("Event")
    Types(i).Name = el.getAttribute("Name")
    j = 0
    k = 0
    ReDim Types(i).Channels(20)
    Set el2 = nodes2.nextNode
    While k < nodes2.length
        If j >= UBound(Types(i).Channels) Then
            ReDim Preserve Types(i).Channels(UBound(Types(i).Channels) + 20)
        End If
        If InList(el2.getAttribute("Channel"), Types(i).Channels) = False Then
            Types(i).Channels(j).Name = el2.getAttribute("Channel")
            j = j + 1
        End If
        Set el2 = nodes2.nextNode
    k = k + 1
    Wend
    Types(i).Nch = j
    Set el = nodes.nextNode
    i = i + 1
Wend

'Read in the names of the instances of processes and add the names of their
channels
Set nodes = doc.getElementsByTagName("Model")
Set el = nodes.Item(0)
Set nodes = el.getElementsByTagName("Instance")
```

```
nodes.Reset
Set el = nodes.nextNode

'If Not nodes.length > 0 Then Exit Sub 'There is nothing to do

i = 0
While i < nodes.length And el.getAttribute("Name") <> CurrentModel
Set el = nodes.nextNode
    i = i + 1
Wend


ReDim Instances(20)
Set nodes2 = el.getElementsByTagName("ProcInstance")
nodes2.Reset
Set el = nodes2.nextNode
i = 0
While i < nodes2.length
    If i >= UBound(Instances) Then ReDim Preserve Instances(UBound(Instances) +
10)
    Instances(i).Name = el.getAttribute("Name")
    Instances(i).Type = el.getAttribute("Type")
    'Find the type and add the names of its channels
    j = 0
    While j < UBound(Types)
        If Types(j).Name = Instances(i).Type Then
            ReDim Instances(i).Channels(Types(j).Nch)
            Instances(i).Nch = Types(j).Nch
            k = 0
            While k < Instances(i).Nch
                Instances(i).Channels(k).Name = Types(j).Channels(k).Name
                k = k + 1
            Wend
        End If
        j = j + 1
    Wend
    Set el = nodes2.nextNode
    i = i + 1
Wend


'Draw the Instances
i = 0
Y = 200

'Make the box big enough to show processes with no connections 8/1/2002
p.width = 3000

While i < UBound(Instances) And Instances(i).Nch > 0
    'Make sure the picturebox is big enough to draw this instance
    If p.Height < (Y + Instances(i).Nch * 300) + 250 Then p.Height = (Y +
Instances(i).Nch * 300) + 500

    p.CurrentX = 350
    p.CurrentY = Y + 50
    p.Print (Instances(i).Name & " : " & Instances(i).Type)
    p.Line (300, Y)-(2000, Y + Instances(i).Nch * 300), , B
    'For each channel draw a stub and remember where it is
    j = 0
    While j < Instances(i).Nch
        p.CurrentX = 2050
        p.CurrentY = Y + j * 300 + 15
        p.Print (Instances(i).Channels(j).Name)

        p.Line (2000, Y + (j + 1) * 300 - 75)-(2750, Y + (j + 1) * 300 - 75)
        p.Line (2750, Y + (j + 1) * 300 - 75 - 25)-(2750 + 50, Y + (j + 1) * 300 -
75 + 25), , BF
        Instances(i).Channels(j).X = 2750 + 50
        Instances(i).Channels(j).Y = Y + (j + 1) * 300 - 75
        j = j + 1
    Wend
    Y = Y + Instances(i).Nch * 300 + 300
    i = i + 1
```

```
Wend


'Draw the connections
Xpos = 3000
Set nodes = doc.getElementsByTagName("Model")
Set el = nodes.Item(0)
Set nodes = el.getElementsByTagName("Instance")

nodes.Reset
Set el = nodes.nextNode
i = 0
While i < nodes.length And el.getAttribute("Name") <> CurrentModel
Set el = nodes.nextNode
    i = i + 1
Wend

Set nodes2 = el.getElementsByTagName("Connection")
nodes2.Reset
Set el2 = nodes2.nextNode
i = 0
While i < nodes2.length 'For each connection, find the ends and draw a line
    Set nodes = el2.getElementsByTagName("End")
    Set el = nodes.nextNode
    j = 0
    While j < UBound(Types)
        If el.getAttribute("ProcInstance") = Instances(j).Name Then
        k = 0
        While k < UBound(Instances(j).Channels)
            If Instances(j).Channels(k).Name = el.getAttribute("Channel") Then
                a = Instances(j).Channels(k).X
                b = Instances(j).Channels(k).Y
            End If
            k = k + 1
        Wend
        End If
    j = j + 1
    Wend

    Set el = nodes.nextNode
    j = 0
    While j < UBound(Types)
        If el.getAttribute("ProcInstance") = Instances(j).Name Then
        k = 0
        While k < UBound(Instances(j).Channels)
            If Instances(j).Channels(k).Name = el.getAttribute("Channel") Then
                c = Instances(j).Channels(k).X
                d = Instances(j).Channels(k).Y
            End If
            k = k + 1
        Wend
        End If
    j = j + 1
    Wend
    Xpos = Xpos + 250

    'Make sure the picturebox is big enough to draw this line
    If p.width < (Xpos + 100) Then p.width = Xpos + 200

    Call Join(p, a, b, c, d, Xpos, (i Mod 4) + 1)
    Set el2 = nodes2.nextNode
    i = i + 1
Wend
End Sub

Private Sub Join(p As PictureBox, X As Integer, Y As Integer, x2 As Integer, y2 As
Integer, XX As Integer, w As Integer)
'Draws a line from x,y to x2,y2 with the vertical part at yy
p.Line (X, Y)-(XX, Y)
p.DrawWidth = w
p.Line (XX, Y)-(XX, y2)
p.DrawWidth = 1
p.Line (XX, y2)-(x2, y2)
```

```
End Sub

Private Function InList(n As String, arr() As Chan) As Boolean
Dim i As Integer
i = 0

While i < UBound(arr)
    If n = arr(i).Name Then
        InList = True
        Exit Function
    End If
    i = i + 1
Wend
InList = False
End Function
```

References

Allen, R. J. and D. Garlan (1997). “A Formal Basis for Architectural Connection.” ACM Transactions on Software Engineering and Methodology 6(3): 213-249.

Aspray, W. (1991). John Von Neumann and the Origins of Modern Computing, MIT Press.

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

Beizer, B., N. Juristo and S. L. Pfleeger (1997). “Cleanroom process model: A critical examination.” IEEE Software: 114-118.

Box, D. (1998). Essential COM, Addison Wesley.

Butler, M. J. (1997). An Approach to the Design of Distributed Systems with B AMN. 10th International Conference of Z Users (ZUM'97), Reading, Springer-Verlag.

Carpenter, A. and N. Messer (1998). The use of VHDL+ in the Specification Level Modelling of an Embedded System. International Forum on Design Languages, Switzerland.

Chin, S.-K., J. Faust and J. Giordano (1995). Integrating Formal Methods Tools to Support System Design. First IEEE International Conference of Complex Systems (ICECCS'95), Fort Lauderdale, Florida, IEEE Computer Society Press.

Christensen, E., F. Curbera, G. Meredith and S. Weerawarana (2000). Web Services Description Language (WSDL) See: http://msdn.microsoft.

Church, A. (1936). “A Note on the Entscheidungsproblem.” Journal of Symbolic Logic 1(1): 40-41.

Clarke, E. M., J. R. Burch, O. Grumberg, D. E. Long and K. L. McMillan (1991). Automatic Verification of Sequential Circuit Designs. Mechanical Reasoning and Hardware Design, Royal Society Discussion Meeting.

Clarke, E. M., O. Grumberg and D. E. Long (1994). “Model Checking and Abstraction.” ACM Transactions on Programming Languages and Systems 16(5): 1512-1542.

Coffman, E. G., M. J. Elphick and A. Shoshani (1971). “System Deadlocks.” ACM Computing Surveys 3(2): 67-78.

Dickman, A. (1998). Designing Applications With Msmq : Message Queuing for Developers, Addison Wesley Publishing Company.

Edelman, A. (1997). “The Mathematics of the Pentium Division Bug.” SIAM Review 39(1): 54-67.

FDR2 User Manual(2000)., Formal Systems (Europe) Limited See: http://www.fsel.com/documentation/fdr2/html/index.html.

Flanagan, D. (1997). Java in a Nutshell, O'Reilly and Associates.

Fowler, M., K. Scott and G. Booch (1997). UML Distilled - Applying the standard Object Modelling language, Addison Wesley.

Garlan, D., R. Allen and J. Ockerbloom (1995). Architectural Mismatch, or, why it's hard to build systems out of existing parts. 17th International Conference on Software Engineering (ICSE-17), Seattle.

Gravell, A. M. and P. Henderson (1996). “Executing formal specifications need not be harmful.” IEE/BCS Software Engineering Journal 11(2).

Gray, D. N., J. Hotchkiss, S. LaForge, A. Shalit and T. Weinberg (1998). “Modern Languages and Microsoft's Component Object Model.” Communications of the ACM 41(5): 55-65.

Grumberg, O. and D. Long (1994). “Model Checking and Modular Verification.” ACM Transactions on Programming Languages and Systems 16(May): 843-871.

Hartel, P. H., M. J. Butler, A. J. Currie, P. Henderson, M. A. Leuschel, A. P. Martin, A. P. Smith, U. Ultes-Nitsche and R. J. Walters (1999). Questions and Answers About Ten Formal Methods. 4th International Workshop on Formal Methods for Industrial Critical Systems (FMICS99), Pisa, Italy, Star/CNR`.

Hashmi, M. M. K. (1998). Extending VHDL for Interface and System Specification. Fall VIUF - VHDL for Power Users.

Hashmi, M. M. K. (1998). VHDL+ Language Reference Manual, ICL.

Hashmi, M. M. K. and A. C. Bruce (1995). Design and Use of a System-Level Specification and Verification Methodology. IEEE European Design Automation Conference.

Hawley, G. (1999). “Selecting a Real-Time Operating System.” Embedded Systems Programming Europe(May): 23-29.

Henderson, P. Enact User Manual.

Henderson, P. (1993). Object-Oriented Specifiaction and Design with C++, McGraw-Hill Book Company.

Henderson, P. (1997). Formal Models of Process Components. International Workshop on Foundations of Component-Based Systems, Zurich.

Henderson, P. (1999). RolEnact, Available from: http://www.ecs.soton.ac.uk/~ph/RolEnact.

Henderson, P., Y. M. Howard and R. J. Walters (2001). "A tool for evaluation of the software development process." The Journal of Systems and Software 59(3): 355-362.

Henderson, P. and R. J. Walters (1999). Component Based systems as an Aid to Design Validation. 14th IEEE International Conference on Automated Software Engineering (ASE99), Cocoa Beach, Florida, IEEE Computer Society.

Henderson, P. and R. J. Walters (1999). System Design Validation Using Formal Methods. Tenth IEEE International Workshop on Rapid System Prototyping (RSP99), Clearwater, Florida, IEEE Computer Society.

Henderson, P. and R. J. Walters (2001). "Behavioural Analysis of Component-Based Systems." Information and Software Technology 43(3): 161-169.

Henderson, P., R. J. Walters and S. Crouch (2001). Inconsistency Tolerance across Enterprise Solutions. 8th IEEE Workshop in Future Trends of Distributed Computer Systems (FTDCS01), Bologna, Italy.

Henderson, P., R. J. Walters and S. Crouch (2002). RICES: Reasoning about Information Consistency across Enterprise Solutions. Systems Engineering for Business Process Change: New Directions. London, Springer-Verlag London Limited**:** 367-371.

Hoare, C. A. R. (1985). Communicating sequential processes, Prentice-Hall International.

Hoare, C. A. R. (1996). The role of formal techniques: past, current and future or how did software get so reliable without proof? 18th International Conference on Software Engineering (ICSE-18), Berlin, IEEE Computer Society Press.

Hodgson, S. and M. M. K. Hashmi (1997). "SuperVISE - System Specification and Design methodology." ICL Systems Journal 12(2): 233-250.

Holzmann, G. J. (1997). "The Model Checker SPIN." IEEE Transactions on Software Engineering 23(5): 279-295.

Hunter, D., C. Cagle, D. Gibbons, N. Ozu, J. Pinnock and P. Spencer (2000). Beginning XML, Wrox Press Inc.

IBM (2001). MQSeries Family See: http://www-4.ibm.com/software/ts/mqseries/.

IEEE (1994). IEEE Standard VHDL Language Reference Manual. New York, IEEE.

Ip, C. N. and D. L. Dill (1996). Verifying Systems with Replicated Components in Murphi. 8th International Conference on Computer Aided Verification (CAV'96), New Brunswick, NJ, Springer.

Jackson, D. (2002). Micromodels of Software: Lightweight Modelling and Analysis with Alloy, Software Design Group, MIT Lab for Comupter Science See:

JavaSoft (1996). Java RMI specification.

Jebson, A., C. Jones and H. Vosper (1993). "CHISLE: An Engineer's tool for hardware system design." ICL Technical Journal 8(3).

Kaveh, N. and W. Emmerich (2001). Deadlock Detection in Distributed Object Systems. 8th European Software Engineering Conference and the 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Vienna, ACM Press.

Luckham, D. C., J. L. Kenney, L. M. Augustin, J. Vera, D. Bryan and W. Mann (1995). "Specification and Analysis of System Architecture using Rapide." IEEE Transactions on Software Engineering 21(April): 336-355.

Luckham, D. C. and J. Vera (1995). "An Event-based Architecture Definition Language." IEEE Transactions on Software Engineering 21(September): 717-734.

Luckham, D. C., J. Vera and S. Meldal (1995). Three Concepts of System Architecture See: http://elib.stanford.edu/Dienst/UI/2.0/Describe/stanford.cs%2fCSL-TR-95-674.

Magee, J., N. Dulay, S. Eisenbach and J. Kramer (1995). Specifying Distributed Software Architectures. 5th European Software Engineering Conference (ESEC 95), Sitges, Spain.

Magee, J. and J. Kramer (1996). Dynamic Structure in Software Architecture. 4th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE4), San Francisco.

Magee, J. and J. Kramer (1999). Concurrency: State models and Java Programs, John Wiley and Sons.

Microsoft (2000). Legacy File Integration Using Microsoft® BizTalk Server 2000, Microsoft See: http://www.microsoft.com/biztalk/techinfo/LegacyFileIntegrationWP.doc.

Microsoft (2001). Microsoft Message Queuing Services, Microsoft See: http://www.microsoft.com/ntserver/appservice/techdetails/overview/msmqrevguide.asp.

Milner, R. (1989). Communication and Concurrency, Prentice Hall.

Milner, R. (1993). The Polyadic pi-Calculus: a Tutorial. Logic and Algebra of Specification. F. L. Hamer, W. Brauer and H. Schwichtenberg, Springer-Verlag.

Object Management Group Common Object Request Broker: Architecture Specification.

Ould, M. A. (1995). Business Processes - Modelling and Analysis for Re-engineering and Improvement, John Wiley and Sons.

Phalp, K. T., P. Henderson, G. Abeysinghe and R. J. Walters (1998). "RolEnact - Role Based Enactable Models of Business Processes." Information And Software Technology 40(3): 123-133.

Platt, D. S. (1999). Understanding COM+, Microsoft Press.

Sessions, R. (1998). COM and DCOM - Microsoft's Vision for Distributed Computing, Wiley Computer Publishing.

Siegmund, R., D. Muller, H. v. Sychowski and J. Lancaster (1998). Specification and Design of complex digital systems using VHDL+. Intellectual Property in Electronics Conference, Europe.

Sullivan, K., J. Socha and M. Marchukov (1997). Using Formal Methods to Reason about Architectural Standards. 19th International Conference on Software Engineering, Boston, IEEE Computer Press.

Sun Microsystems Enterprise Java Beans See: http://www,sun.com.

Szyperski, C. (1998). Component Software, Longman.

Thomas, A. (1998). Enterprise JavaBeans Technology, Patricia Seybold Group.

Turing, A. (1936). "On Computable Numbers with an Application to the Entscheidungsproblem." Proceedings of London Mathematics Society 2(42): 230-265.

Turner, D. N. (1995). The Polymorphic pi-calculus: Theory and Implementation. Edinburgh, University of Edinburgh.

Universal Description Discovery and Integration (UDDI)), Technical White Paper(2000). See: http://www.uddi.org.

Walters, R. J. (2002). A Graphically Based Language for Constructing, Executing and Analysing Models of Software Systems. 26th Annual International Computer Software and Applications Conference (COMPSAC 2002), Oxford, IEEE Computer Society.