Questions and answers about ten formal methods

Pieter Hartel, Michael Butler, Andrew Currie, Peter Henderson, Michael Leuschel, Andrew Martin, Adrian Smith, Ulrich Ultes-Nitsche, Bob Walters *

January 12, 1999

Abstract

An abstract model of a distributed data base application has been studied using process based, state based, and queueing theory based methods. The methods supported by graphical notations and/or integrated development environments were found to be easiest to work with. The methods supported by model checkers were the most successful in obtaining relevant information about the application.

Applying a number of different methods to study one particular model encourages a problem to be viewed from different angles. This gives complementary information about the model. We report on a variety of problems of the model found through various routes. Our main conclusion is that asking experts to apply different methods and tools at a sufficiently abstract level is effective.

1 Introduction

A given distributed data base system is used to serve customers such that each service can only be provided to one particular customer, and only once. The system has the following components:

- A single *Centre*, where most of the data is held;
- More than one *Office*, where copies of relevant parts of the data are held;
- Many *Customers*. Each customer has a home office which will normally hold the data pertaining to that customer. A foreign office should be able to provide a customer with the same services as the home office, after due consultation with the centre.

When presented with a description of the system and its requirements by our industrial partner, one of us created a rational reconstruction of the *model* underlying the system. Our objective here is to explore that model. We use a number of different formal methods and discuss the strengths and weaknesses of each in supporting our modelling activity. This is quite different from normal practice where one would construct (not re-construct) a model, validate it against the problem statement and requirements until sufficiently confident, and then refine the model through various design stages to an implementation.

Our methodology exploits the collective knowledge of a team of "champions" of different formal methods as follows:

1. One of us presented the rational reconstruction of the model to the team.
2. The team members created their own version(s) of the model. These efforts were documented in the form of an appendix to the present paper.
3. The team members presented their work to the rest of the team in a sequence of weekly meetings.
4. Each team member was interviewed, always by the same person, to answer a list of questions about the most important aspects of the method used.
5. The answers were collated and reviewed by the team, resulting in the present paper.

Our working method is different from that used in other, similar researches. For instance in the steam boiler control project [2] authors were sent a list of eight questions and asked to respond to the questions, using the refereeing process to improve the coherence in the responses. Because the contributors to our research are all from Southampton, we could afford to interview instead and thus achieve a higher degree of consistency right from the start. Our questions are partly based on those used in the steam boiler project, and partly inspired by discussions with our industrial partner. Our initial list of questions proved too long and was shortened during the process, leaving 18 questions in total. Of these roughly half coincide with the questions from the steam boiler control project.

The second important difference between our work and the work on the steam boiler controller is that we started on the basis of a solution, and then constructed rational

*Declarative Systems and Software Engineering group, University of Southampton

reconstructions (models) of the solution through abstraction. There was a significant degree of freedom in determining the direction and the degrees of refinement of the models.

Section 2 introduces the methods and tools that we use. The model of the distributed data base application is presented in Section 3. Section 4 discusses the modelling techniques used rather than the model itself. Section 5 concludes.

The main body of the paper has been written for a general audience, without specific knowledge of formal methods. We provide a number of appendices detailing the models in the notations of the various formal methods used. Those appendices do assume knowledge of each of the methods.

2 Methods

We first give a brief description of each of the methods. The first three are based on variations of the π-calculus, which is an algebra of communicating processes:

epi is an executable version [10] of the polyadic π-calculus [22]. Epi has been enriched with primitive data types, such as integers and sets, thus providing appropriate facilities for the description of our model.

latosπ is an executable specification of the operational semantics of the monadic π-calculus [23, Table 2]. The specification is written using Latos [9]. The π-calculus does not provide built in data types, nor does the latos specification of the π-calculus. The lack of data structures makes it difficult to scale the model up to more than a few customers and offices.

Prologπ is an executable specification of the operational semantics of the polyadic π-calculus [22], but without the full scope extrusion rules. The specification is written in Prolog, and it offers data structures by way of access to the underlying Prolog terms.

We use a method based on fine grained communicating processes:

SuperVISE uses a language called VHDL+ [16], which consists of the hardware description language VHDL [24], extended with a notation for specifying interfaces. SuperVISE differs from the other techniques presented here in that its development represents a move towards the abstract/theoretical from the concrete/practical. The other techniques might be more appropriately classified as a move from the theoretical to the practical.

We use a number of techniques based on communicating processes that support model checking:

Product-nets have been used in the specification and analysis of the example model. Product-Nets [4] are high-level Petri-nets [31, 33] with data-structures to all places and with predicates as guards to all places. Therefore, tokens have an internal structure, for instance to represent messages, queues, etc.

Spin is a model checker originally designed for validating communications protocols [13]. Its modelling language, Promela, supports dynamic creation of concurrent processes and both synchronous and asynchronous message passing.

Murϕ is another model checker, also originally designed for communications protocols [7]. A Murϕ description consists of a set of transition rules comprising a condition and an action; the execution model involves repeatedly making a nondeterministic selection of a rule whose guard is enabled and executing the corresponding action. All interactions take place by means of shared variables.

For our state based specifications we use two industry standard methods:

B notation [1] is based on set theory and the weakest pre-condition calculus.

Z is a specification technique based on first order logic and set theory [36].

The last method complements the others in that it focuses on non-functional behaviour (i.e., performance) rather than functional behaviour:

QNAP2 Queueing networks are a method for predicting the performance of a system from a wired model of queues and server stations [3].

All methods are provided with support tools. We give a brief account of the various tools that were used.

epi supports animation and state space exploration of behaviours of π-calculus descriptions. The animator enables the user to explore the state space exhaustively, to interactively step through computations, to roll computations back and to alter values operated upon by the description.

latosπ supports full state space exploration of relatively small π-calculus descriptions.

Prologπ is supplemented with a variety of tools—written in standard Prolog—for state space exploration and tools for the expression of safety properties.

SuperVISE differs from the other tools presented in that it does not provide simulation, verification or evaluation of models directly. Instead, SuperVISE generates standard VHDL code from VHDL+. The resulting code is then analysed using a standard simulation tool, although there is a some integration between SuperVISE and the most popular simulation tools. VHDL simulation tools provide comprehensive facilities for examining the behaviour of electronic devices which were the original target of the language. Models may be run manually, but it is left to the user to generate (i.e., specify in VHDL) an appropriate test environment if, for example, a model is to be exercised fully or an exhaustive search of its states is required.

Product-nets The Product Net Machine (PNM) [29] and the SH-Verification Tool (SHVT) [30] provide tools for the complete cycle from specification to (exhaustive) validation. The two programs comprise a graphical editor for specification, a simulator / complete analyser for computing the (dynamical) behaviour of the system, including abstractions of the behaviour, and a model-checking tool to check properties of the specification, also under fairness assumptions.

Spin can perform random or interactive simulations, and can perform an exhaustive state space verification of both safety and liveness properties. It also supports verification of linear-time temporal logic (LTL) constraints. Xspin [14] is a user-friendly graphical front-end to Spin, and allows animation of simulations and error traces.

Murϕ can perform simulation or exhaustive verification, although only deadlock and assertion violations are detected. Error reporting is by means of a textual trace of rules fired which must be interpreted by hand.

B toolkit provides a comprehensive package of tools for animation, proof, refinement and implementation of descriptions [25].

Z is one of the most mature specification notations. Many different tools have been developed for Z, including industrial strength type checkers, proof systems and animators.

QNAP2 is a language that supports the construction, simulation and analysis of queueing networks [34]. SWAP [8] is a tool built on top of a library of QNAP models, which focuses on system-level parameters relevant for modern servers, workstations and networks.

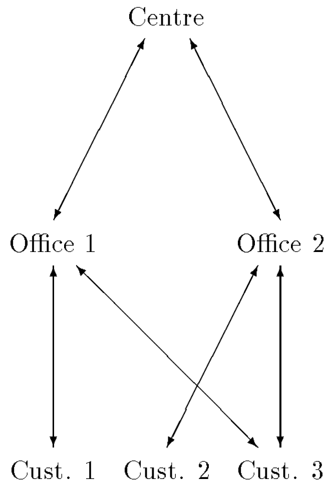

Figure 1: Graphical representation of the model, showing one centre, two offices and three customers. The edges show which parties may enter into communication.

3 Model

An informal, graphical description of the most basic model is provided in Figure 1. This identifies the centre, two offices and three customers with the following properties:

- The data for customer 1 is held by her home office 1; similarly for customer 2 and office 2. A foreign office is one that does not hold the customers data. Customer 3 does not have a home office, instead her data are held by the centre.
- All three customers will immediately attempt to obtain service from both offices. Customer 1 will succeed directly when contacting her home office 1 but will fail when contacting a foreign office 2. Similarly for Customer 2. An office that does not hold the appropriate data contacts the centre to obtain the data.
- The centre delivers the data for customer 3 to the requesting office, which will then serve customer 3. Communication between office and the centre will fail for customers 1 and 2.
- Once served, a customer is satisfied and does not request further service.

In a more refined model, a customer may be served also by a foreign office instead of her home office. In this case the foreign office contacts the centre, which in turn contacts the home office for the data. In an even more refined model, the customers might keep requesting services. We will discuss experiments where one or both of these refinements have been applied.

3.1 Abstract description

Some of the experiments reported below apply to only one version of the model. Others report on an abstract model and one or more refinements and/or variations. The questions are based on an experiment with an abstract model and a refinement.

Question 1
Is the abstract description of the model: – formal? – rigorous? – verbal?

epi,latosπ,Prologπ,Spin,Murϕ,B,Z The abstract description of the model(s) is formal.

SuperVISE attempts to fully describe the system, but its practical nature means it is unreasonable to claim that it is formal or rigorous.

Product-nets Because of its graphical representation the Product Net model helps to understand the flow of information in the system. The description is formal.

QNAP2 model is formal in the sense that it is based on a sound mathematical theory.

Some models are more abstract than others, in the sense that more or less detail is taken into account. It is easier to study more abstract models, but from a practical point of view it is more useful to study more detailed models.

Question 2
How abstract is the description of the model?

epi description is abstract because it represents the core of the real system, but omitting all detail. The description represents the centre, two offices and three customers as illustrated in Figure 1. Data bases are modelled using sets of tuples.

latosπ description is similar to the epi description. The main difference being that there is no proper model of the data base.

Prologπ description is detailed in the sense that it describes customers, offices and the centre with their communications and individual state. It is generic in the sense that the model supports an arbitrary number of processes, not just 6 as in several of the other models.

SuperVISE model is detailed and concrete rather than abstract. However, the working model is constructed by describing components which are then joined together. Part of the methodology adopted by SuperVISE users is to draw a block diagram showing how these components are connected.

Product-nets The model is abstract in the sense that it describes the system from the customers point of view, according to Figure 1, and in doing so allows the customer to be involved in one transaction only.

Spin has been used to create two abstract models of the system: one in which a customer ceases to exist once served, and a variation in which customers can continue to ask for services. These models are not abstract for two reasons: Firstly they take all aspects of the system into account, and secondly the lack of data abstraction (only arrays are provided) in Spin makes the descriptions fairly detailed.

Murϕ has been used to create two abstract models as described above under Spin. These models are even less abstract than the Spin models, in the sense that nondeterminism happens at global level and that all communication between processes is mediated through global variables.

B describes the service from the users viewpoint and is independent of the distributed nature of the implementation. This model is thus more abstract than that shown in Figure 1.

Z model is abstract in the sense that it describes the system solely from the customers point of view. At the abstract level there is not even the notion of an office.

QNAP2 model is abstract because it describes the system in terms of a single or small set of post office submodel, one submodel for each different post office configuration. It would be possible to model separate classes of customers but the need has not arisen to do so.

Animation is a technique that allows sample behaviours of a model to be traced and studied. The technique is useful to validate a model, by checking that the animations represent desired outcomes. Graphical animation is generally more attractive than textual animation but the latter has a useful role too.

Question 3
Has the abstract description been animated?

epi,latosπ All possible behaviours of the abstract description have been explored through exhaustive state space exploration. This is possible because the models are small and the state space is finite. The animations are textual.

Prologπ textual animations have been performed using three different search methods, executed with the standard Prolog backtracking mechanism:

- A simple depth first search with tracing;
- An interactive depth first with tracing;
- A breadth first search with tracing.

All these were easy to implement with little effort. There is scope for writing more sophisticated animators in a similar style.

SuperVISE model has been validated "informally" by manual examination of the model in action. SuperVISE does not provide proof or validation facilities directly.

Product-nets The model has not been animated graphically. Instead, the simulator provided with the product net system has been used to simulate all possible behaviours, which were then used as input to the model checker.

Spin The most useful animation is provided by a (graphical) message sequence chart, which focuses on the processes and communications. A message chart hides the internal details of processes. Detailed traces and the opportunity to interact with running processes are also provided.

Murϕ produces a detailed textual trace of the rules which are fired.

B animator has been used to explore some behaviours of the abstract description. The animations are textual.

Z The abstract model has been animated (textual) using PiZA [11].

QNAP2 model has been animated in two different ways:

- Various solvers have been used to calculate analytic solutions to the equations describing the queueing network. The solutions describes a steady state of the system.
- When an exact solver cannot be used, simulation forms the basis of an approximate solution.

3.2 Detailed description

After studying an abstract model, one may refine the model to a more detailed model. This should give more detailed insight into the internal behaviour of the system.

Question 4
Has a second, more detailed description of the model been given?

epi,latosπ,Prologπ,SuperVISE Only one model has been created.

Product-nets A second more detailed model has been created, which differs from the first abstract model in allowing the customers to remain in the system. A customer can now be involved in a sequence of transactions.

Spin In addition to the two abstract Spin models, a third model has been created, which in a sense is a refinement of one of the abstract models. The more detailed model uses asynchronous communication implemented by queues. The implementation of queues by buffered channels is provided as a standard Spin facility.

Murϕ has been used to investigate only the two abstract models. The liveness properties of the queuing model which were of most interest, but Murϕ does not support verification of these.

B A refinement of the model has been created, which introduces the notion of distributing the data base.

Z From the abstract Z model (level 0) two more detailed models have been derived:

- The first more detailed model (level 1) adds the notion of separate offices where authority to perform a transaction might reside.
- The second model (level 2) introduces further detail by adding the notions of message passing and the idea that network transactions might be involved in satisfying some requests for service.

QNAP2 A second more detailed model has been produced which which takes into account the performance of real components such as state of the art servers, work stations and network devices.

A detailed model should behave correctly whenever the abstract model does so. Some of the experiments use formal refinement which is correctness preserving. Other experiments use other verification techniques to establish the relationship between abstract and detailed models.

Question 5
Has the detailed description been verified against the abstract description? – formally? – rigorously? – verbally?

epi,latosπ,Prologπ,SuperVISE Only one model has been given.

Product-nets The more detailed model is not a refinement of the abstract model. It is formally related to the abstract model in the sense that both models share various Linear Temporal Logic (LTL) properties. These properties were proved by the model checker for both models.

Spin has been used to formally verify safety and liveness properties of the abstract and detailed models. Absence of deadlock is preserved by the refinement. The liveness properties are different in the two models. In the abstract model customers are prone to starvation, but not in the detailed model.

Murϕ has been used to verify that in the detailed model the same customer cannot be in two places at the same time.

B The detailed description has been validated against the abstract description using mathematical proof techniques (proof sketches).

Z The detailed descriptions have not been validated against the abstract model. This could be done without difficulty for the level 1 model. Formal validation for the level 2 model would be a lot of work because of the complexity of the model.

QNAP2 The detailed model has not been verified against the abstract model, although this would have been both possible and useful. A verification effort would have to check that in the steady state the statistics produced are consistent.

3.3 Producing the description(s)

Some of the experiments have taken much longer than the others. This should not be interpreted as a reflection on the tools/method used, but more as an indication of the level of skill of the experimenter.

Question 6
How much time has been spent on producing the model and any refinements?

epi About a day was needed to create the epi model. The epi tool had been built previously.

latosπ It took about 2 person months to learn the π-calculus, to create a deterministic version of the π-calculus semantics (from the published non-deterministic semantics [23]) and to experiment with the model. The time attributed to working on the model might be two days.

Prologπ It took about half a day to specify the π-calculus semantics. Constructing the model, and experimenting with it took about 2 days.

SuperVISE It took about 2 person months to learn the technique/tools and to apply this to the problem. Building the model itself might have taken about two days.

Product-nets One day was spent to create and study the two Product net models.

Spin Constructing and experimenting with the three Spin models took about a week.

Murϕ Constructing and experimenting with the three Murϕ models took just over a week.

B It took approximately two days to produce the specification and the refinement.

Z Two days were needed to construct the three Z models.

QNAP2 The abstract QNAP2 model was produced in half a day. It took about four days to produce the detailed system-level model.

3.4 Understanding the description(s)

There are two aspects to understanding and presenting the model. The first is an in depth presentation of the models itself, which often (but not always) requires more time than the second, which is a more general presentation, supported by showing animated behaviours. The next two questions asses the level of skill required for either of these, where we refer below an 'average programmer' as a a person with good programming skills in languages such as Java and C. No experience with formal methods, functional or logic languages is assumed.

Question 7
Is a detailed knowledge of the used formalism needed to understand the description(s) themselves?

epi,latosπ The π-calculus has an interesting semantics with its scope intrusion and extrusion rules. These need to be understood, which is not trivial. The average programmer will also have difficulty with for instance the λ notation. To explain the π-calculus model in sufficient detail might take a day.

Prologπ A detailed knowledge of Prolog is not needed to understand the description, but knowledge of the π-calculus would be needed.

SuperVISE Detailed knowledge is necessary to understand the description. Two days may be sufficient for the average programmer because VHDL looks like a programming language.

Product-nets The descriptions of the model are pictures but several other aspects of the methods, such as the LTL formulae are textual. A day would be sufficient for an average programmer to understand the description itself

Spin To understand the Spin description maybe half a day would suffice, because the Spin notation Promela has a C-like syntax.

Murϕ descriptions are fairly low level, and learning to understand them would take maybe a day. Murϕ descriptions have a Pascal-like syntax.

B Detailed knowledge of the formalism is not needed; with a few hours of explanation, the descriptions could be understood.

Z A training course of half a week would be needed before an average programmer is able to understand the Z models. Z descriptions look like Mathematics.

QNAP2 is a Fortran-like programming language. For an 'average programmer' to become sufficiently acquainted with the notation might take half a day.

To be able to present sample behaviours to an audience of 'average programmers', it is useful to know to what extent the audience must be trained before it would benefit from such presentations. We are assuming that in all cases the model needs to be explained as well, which is thought to take about 15 minutes.

Question 8
Is a detailed knowledge of the used formalism needed to understand the behaviour of the abstract or detailed models?

epi Half an hour is sufficient to explain the behaviour of the model as shown by the graphical user interface to the epi animation.

latosπ The lack of a graphical user-interface makes it as difficult to interpret the behaviours generated by the model as it is to understand the model itself. Some knowledge of the π-calculus is needed, which would take half a day to explain.

Prologπ A detailed knowledge of Prolog nor π-calculus is needed to understand the results of the animations, but a notion of processes and communication would suffice. This could be explained probably in one hour.

SuperVISE Detailed knowledge of VHDL+ and the code are not needed, but an appreciation of the components within the model is. This could be communicated using a block diagram. However without a working knowledge of the VHDL simulator being used and of the simulation/operation of electronic circuits an observer will have difficulty. The total time would be half a day.

Product-nets The pictorial nature of Product-nets makes it relatively easy to explain the behaviour of the model. An hour would be sufficient to achieve sufficient understanding.

Spin Detailed knowledge of the method is not needed as the graphical output of the message chart animator is readily understood. The total time for an explanation would 30 minutes.

Murϕ Detailed knowledge of Murϕ is required as the only tangible result of a Murϕ execution is a detailed textual trace of the rules that have been fired. This might take half a day.

B One would need to explain the basics of B's state variables and operations, which would take 15 minutes. In addition to that the model must be explained, making a total of 30 minutes.

Z Detailed knowledge is unnecessary to understand the results of the animations. A typical 2-day Z reading course as used in industry will be sufficient.

QNAP2 An understanding of the underlying mathematics is required to understand the results produced by the solvers. Appropriate training might take half a day.

4 Techniques and Tools

There is a wide variety in the nature of the various tools and techniques. Some of the techniques were developed purely for their theoretical interest, whereas others have been developed to satisfy practical needs, for example modelling distributed systems.

Question 9
How practical is the tool/technique?

epi provides an effective notation for capturing and understanding certain types of complex behaviour. It would be more difficult to describe large systems, and for many simple systems using epi would not be cost effective.

latosπ notation is useful only in the abstract due to lack of data structures. The technique would not be practical for more concrete or larger applications.

Prologπ The approach of using Prolog to specify the semantics of the π-calculus and then to specify the model in the π-calculus is practical because neither involve a lot of work once familiarity with both has been acquired. To be able to specify the semantics of the modelling language separately from the model specification brings with it a significant degree of flexibility. Prolog makes it straightforward to add sophisticated tracing facilities. The drawback is that only depth first searches are efficient, breadth first searches can be impractical due to large memory requirements. With a tabling implementation of Prolog, however, (i.e. one that avoids recomputation of a previously reached goal) the power of finite state model checking is easily available [32].

SuperVISE is not practical for the present purpose because:

- writing VHDL+ is alien to most programmers and it is too low level for the particular problem at hand. VHDL+ does not provide the right kind of abstraction.
- simulating VHDL+ is hard because the tools are not targeted for this sort of activity.

Product-nets are practical. Understanding the method and tools would help the programmer whilst implementing the system. Code cannot be derived automatically from the specification. The method and tools would be more useful to the designer for comparing and validating different designs. The product net tool is good at providing accurate information about specifications. Other tools exist that can handle larger specifications but do this at the cost of providing less accurate information.

Spin is practical because the notation is appropriate to describing concurrent systems. The tool provides useful information about the behaviour of the model, and does so in a graphical form.

Murϕ is not practical because the semantic gap between a mental model of a concurrent system and a rule based transition system is large. Bridging the semantic gap is difficult, error prone and gives rise to a loss of abstraction. Murϕ lacks the ability to reason about liveness properties.

B is practical because the method and toolkit are used in industry for commercial projects.

Z is practical and widely applied. Most of the tools that we have used (Z/EVES [35], ZTC [17] and PiZA [11]) are not merely academic tools. The show case of using Z is the specification of the IBM CICS system [15] (a distributed transaction processing system). All these observations indicate that Z with its tools is quite practical.

QNAP2 is practical and it is programmer friendly. The performance models are useful as part of the requirements. The results derived from the performance models are useful during system level design. SWAP is practical because it incorporates a visual representation using of the system using the latest GUI technology.

Some notations give rise to more intuitive descriptions than others, mainly through the use of graphical representations. Using familiar notation borrowed from programming languages can also be helpful.

Question 10
Did the technique allow/encourage giving clear and simple descriptions?

epi description is clear and simple, and is comparable in this respect to the Z and B notations. The Product-net notation is simpler.

latosπ description is less than clear and simple because of the lack of data structures in the pure π-calculus that we used. It can be quite difficult to understand the interactions of the processes because of the exponential growth of the number of interleavings between processes.

Prologπ is perhaps not the ideal formalism to specify distributed data base systems because the calculus lacks convenient facilities for specifying data structures. For bigger models this would be a problem.

SuperVISE VHDL+ is too low level to allow for a clear and simple description of the model.

Product-nets specifications are clear and simple. The notation makes hierarchical descriptions possible. It supports, but does not necessarily enforce, the use of good structuring methods.

Spin notation provides processes and channels, which allow for clear and simple descriptions of concurrent systems. The data structures provided are crude (arrays only).

Murϕ is less appropriate for modelling concurrent systems, because of its flat communication structure (processes interact at global level only) and the crude support of data structures (arrays only).

B description is small and clear and is independent of the distributed architecture of the design.

Z The abstract model is clear and simple, fitting comfortably on one slide whilst using meaningful names.

QNAP2 The queueing networks at the abstract level are expressed in a programming notation. This makes it too easy to develop overly complicated descriptions. The programming activity has to be supported by additional work to draw graphical representations of the queueing networks. At present this is done manually. The graphical representations are clear and simple.

4.1 Discoveries

The main purpose of building a model of a system is to gain a better understanding in the problem domain, which in our case is the behaviour of customers, offices and centres whilst managing a distributed database. Once a good understanding of the problem domain has been obtained, one might have specific questions about the domain, such as whether it is possible to obtain a particular service twice.

Question 11
Did the technique help you to explore the problem domain?

epi model was created precisely to explore the problem domain. Certain questions are sufficiently complex to make good use of the description of the model. For example can a service be provided twice or can that be ruled out?

latosπ state space exploration shows that:

- There is an exponential number of interactions between processes.
- All clients are eventually served.

Prologπ has helped us to explore two questions about the domain:

- A thief has been introduced along with the regular customer to show that the thief cannot obtain a service.
- Animations using depth first and breadth first searches have given confidence that a customer is eventually served. Model checking has yet to take place to prove this property.

SuperVISE does not really help to explore the problem domain, because of the need to draw pictures of processes and connections before beginning to specify. SuperVISE might have helped to find answers to questions about the domain if we had pursued this avenue beyond our current investment. We are currently investigating this issue using another case study.

Product-nets helped us to answer the following two questions:

- If a customer is provided with her service after a request for service is passed on to the centre, the service is provided only after the centre has approved it.
- If a customer goes to her home office, she is served eventually.

Spin exploration of the problem domain has helped us to discover and study two problems:

- The lack of certain liveness properties of the second model (where customers remain active in the system) prompted us to introduce the refinement with queues, which has all the required liveness properties. Queues would indeed be necessary in the implementation of the real system, and the Spin experiments have allowed us to find the minimum size of the queues between customers and offices.
- The second problem we found is that a 'stubborn' customer can always go to the wrong office and thus never obtain a service.

Murϕ The model is so simple that no insights in the problem domain have arisen.

B helps to explore the problem domain. The description is concise and independent of the distributed implementation and is only concerned with the problem from the point of view of the users of the system. This makes it easy to ask questions about the description without being encumbered by irrelevant detail. The abstract B description specifically helps to identify the nature of the service offered to customers from the customer viewpoint.

Z animation has made it possible to answer the following questions about the domain:

- The same service cannot be provided twice.
- If there is an authority for providing a service then it will be provided.

QNAP2 and SWAP suggest different ways of configuring systems e.g. types of scheduling, multi-server options. The gives insight into the potential sources of bottlenecks in a real system.

Any model will be an abstraction of reality; our model is particularly abstract. A model will thus leave unspecified various aspects. One needs to gain insight in such aspects before for example moving on to building a more detailed model.

Question 12
Did the technique help you to identify missing parts of the model?

epi Here are two examples of relevant questions, which have been explored using the epi description and the tools:

- What happens if two customers turn up simultaneously at two separate foreign offices?
- Is it possible for a customer to get two offices designated as her home office?

latosπ animation has identified two potential problems with the epi version of the model, on which the latosπ version is based:

- Hidden return channels are not used throughout, instead some of the return channels are public. Using hidden return channels would make the specification more robust.
- The description should have been encapsulated (using the restriction operator to hide all free names). This would prevent public channels from interacting with rogue agents.

Prologπ has helped us to uncover one missing feature of the initial specification: The model is liable to a deadlock in the situation when office 1 asks the centre for information which resides at office 2 and vice versa.

SuperVISE helped to discover the following problem: In an earlier version of the system, we had concentrated so much on authorising services at a foreign office that we had forgotten to revoke authorisation at the home office. This was brought to light by the simulation activity.

Product-nets,QNAP2 No missing parts were identified since the model was not intended to be detailed or even complete. In general, omissions, if any, would have been found.

Spin,Murϕ both detected immediately that, in the original description of the solution, if the centre does not know a customer, deadlock occurs.

B The question that came to light whilst working on the B description was when the relevant parts of the data base are distributed.

Z animation identified a situation where the state of a component was partially undefined.

A model might contain inconsistencies, or there may be discrepancies between an abstract and a more detailed version of a model. Such inconsistencies are harmful and need to be discovered.

Question 13
Did the technique help to identify inconsistencies in descriptions?

epi,latosπ,Prologπ,Murϕ,B did not help to discover inconsistencies.

SuperVISE has the capability to find inconsistencies in the model, but relies on the user to drive the simulations to recognise them should they occur.

Product-nets The first, abstract model contained a deliberate inconsistency in the sense that a customer disappears from the system after having been served. This inconsistency showed up after all customers in the system had been served.

Spin did not help to find inconsistencies, but one could imagine two offices having the records of a single customer. In this case it would be necessary to verify that the customers get served only once.

Z did not help to find inconsistencies in any of the three models. We have not tried to prove properties of the models, but if we had, any inconsistencies would have come to light.

QNAP2 has not helped us to discover inconsistencies but the SWAP tool checks that system-level configurations are valid with respect to the parameters of the library components.

4.2 Background

Here we present information about the background of the experiments, such as the source of information used whilst building the models, and the motivation for building the model.

Question 14
What is your description based on?

epi model is based on recollection of oral presentations and documents on the real system.

latosπ model is closely based on the epi description.

Prologπ,SuperVISE,B,Z,QNAP2 The description is based on recollection of oral presentation.

Product-nets The first, abstract model is based on oral presentation and documentation. The second more detailed model is based on independent development.

Spin model was based on recollection of an oral presentation. This turned out to be not entirely correct. Interestingly, the experimentation with Spin brought the misunderstanding to light. The incorrect interpretation assumed that the centre would sometimes communicate directly with the customer. However, a customer only communicates with an office, which then communicates with the centre on the customers behalf.

Murϕ model is based on recollection of oral presentation and the Spin model.

The methods that we have used are mostly either process based or state based, but we have also used some other approaches. It is interesting to see that more than anything else, the nature of the method determines the focus of the model, and also the way the components of the system are modelled.

Question 15
What is the main focus of the description?

epi model focuses on message passing. To describe this it was necessary to also provide a description of the internal state of the processes involved.

latosπ The centre, offices and customers are processes, communicating over channels with fixed names. All other objects are represented by names passed over the channels.

Prologπ The centre, offices and customers are processes, communicating over channels with fixed names. The data bases are facts in the Prolog system.

SuperVISE focuses on concurrent processes exchanging messages. The data bases are represented as local state owned by the processes.

Product-nets The focus of the model is provided by the customers. They are represented as tokens wandering through the network. Offices are also represented as tokens, but they remain at their positions. The centre is represented by a sub network. It can be viewed as an active component. Communication is represented by firing of transitions. The data base as maintained by the centre has the form of a multi set.

Spin model focuses on the processes and communications. The data base was modelled as a pair of arrays.

Murϕ is mainly state based. Therefore, modelling in murϕ focuses on breaking the life cycle of a process in phases. This is necessary to make explicit the points at which interaction between sections of code occurs and points at which non-deterministic choice occurs. In addition, one must ensure that an explicit handshake takes place for each synchronous communication.

B The service is represented as an abstract state machine with which users interact. Both the description and design are described as single state machines. The components are modelled implicitly by the operations and state variables that relate to them, e.g., the centre is modelled implicitly by the operations for distributing services and for dealing with non-local queries. The data is represented by appropriately typed state variables. Communication is modelled by operations.

Z The focus of the level 0 and 1 models is state, the focus of the more detailed model is state combined with communication. In all models the data bases are modelled by partial functions, customers are not explicitly modelled. The centre and the offices are modelled depending on the level of detail. In the most abstract model no centre or offices exist. At level 1 the centre and the offices are modelled by state, and at level 2 they are modelled by more detailed state and the ability to send and receive messages. At level 2 communications is modelled by a relation.

QNAP2 model focuses on the performance characteristics of a queueing network. The most important characteristics are the workload imposed by the customers, and the quality of service obtained by the customers.

Tool support is important, as studying a model involves many repetitive tasks that can successfully be automated. The user is then encouraged to concentrate on the intellectual challenges in the experiment.

Question 16
What tools did you use and why?

epi animator and associated state space search engine proved invaluable to explore the full state space.

latosπ translates the model, as well as the description of the operational semantics of the monadic π-calculus into a Miranda [37] program. This program explores the entire state space.

Prologπ Sicstus Prolog [6] has been used for the experiments. The LOGEN partial evaluator [18] has been used to compile the π-calculus semantics with the model into low level Prolog so as to allow for faster animation.

SuperVISE is the translator from VHDL+ to VHDL, and ModelSim is the VHDL simulator.

Product-nets The tools use are the Product net machine and the SH verification tool. They offer:

- a graphical editor to create the specification,
- a project administration tool to manage specifications consisting of multiple subnets,
- a simulation tool on the graphical level,
- a textual simulator with different operation modes (user driven/ randomly driven, stepwise simulation/ multi-step simulation),
- an exhaustive simulator (complete reachability analyser),
- an abstraction tool to decrease state-space sizes and to perform certain consistency checks,
- a temporal logic model checker including linear and approximate satisfaction (i.e., satisfaction under fairness) of properties.

The different parts of the tool are integrated by a common user interface that allows for a unique access to the different parts of the system.

Spin provides an integrated environment (Xspin) running under X-windows with a simulator, verifier, message animator, LTL manager, and finite state machine viewer of control graphs. The tools are nicely cross referenced, so that clicking on an object or event in one window highlights corresponding parts of other windows.

The Spin simulator can be driven interactively or by traces provided by the verifier. Spin was chose because of its maturity, the fact that it is widely available and free.

Murϕ has a Spartan interface consisting of the Murϕ compiler (which generates C++).

B toolkit was used to syntax-check and animate the description. It was also used to check the consistency of the description.

Z Three tools have been used to support the modelling in Z:

- The LaTeX document processing with appropriate styles for producing documents and slides.
- The Z/EVES system for type checking of the LaTeX source. Z/EVES also supports interactive theorem proving but we have not made use of this facility.
- The PiZA system animates the specification. Its input is a direct translation of the LaTeX sources.

The ability to work with one single source is an advantage.

QNAP2 is an established tool for calculating properties of queueing networks. A possible alternative would be to use a general purpose tool such as Mathematica. The Swap tool was used because of our involvement in its development.

Not all experiments have been performed with the same motivation. For example one might be inspired by the problem itself, or one might be interested in applying a particular tool. We do not believe that the motivation has actually influenced the outcomes because whatever the motivation, the experiment wants to be successful. Here success is measured in terms of the number of discoveries, either about the problem domain, the model or the method.

Question 17
Why did you write the description?

epi description was created to understand the principles of the real service.

latosπ was used to study an application of the π-calculus semantics.

Prologπ and the ECCE partial evaluation system [21] has been used to experiment with partial evaluation and abstract interpretation as a form of model checking. This is a fairly recent and promising idea [19, 20], which we are planning to pursue further.

SuperVISE was used to evaluate the tools for the purpose of modelling distributed systems.

Product-nets The Product net model was created to be able to compare the tools and the technique with the other tools.

Spin,Murϕ were used to show that model checking is a valid approach to studying distributed systems, and that model checking can provide useful insights.

B was used to understand and clarify the problem domain and to produce a provably-correct design.

Z models were created to help understand the problem and to show that Z is useful for describing models of distributed systems.

QNAP2 was used to show that even small performance models might be useful. Swap was used because we wished to exercise it.

5 Conclusion

The final question attempts to summarise the particular experiences. We should like to stress that our conclusions

.	Process based							State based		
#.Question	epi	latosπ	Prologπ	SuperVISE	Product-nets	Spin	Murϕ	B	Z	QNAP2
1.abstract description is formal (++=formal and graphical)	+	+	+	−	++	+	+	+	+	++
2.description is abstract (+=as in Figure 1, ++=more, −=less abstract)	+	+	+	−	+	+	+	++	++	++
3.abstract description animated (++=more than just manually driven, −=not animated)	++	++	++	+	++	++	+	+	+	++
4.there is a second detailed description (++=there is also a third refinement)	−	−	−	−	+	++	+	+	++	+
5.detailed has been verified against abstract (++=proof sketch, +=properties only)	n.a.	n.a.	n.a.	n.a.	+	−	−	++	−	−
6.days spent creating the descriptions (assuming that there are 20 working days per month)	1	40	2	40	1	5	7	2	2	5
7.days needed to understand descriptions	1	1	1	2	1	0.5	1	2	3	0.5
8.hours needed to understand animation	0.5	4	1	4	1	0.5	4	1	1	4
9.practicality of the tool/technique (++=scalable and/or graphical)	+	+	+	−	++	++	−	++	++	++
10.clarity of the descriptions (++=data abstractions possible)	+	−	−	−	++	+	+	++	++	+
11.helps to explore the problem domain (++=something *was* learned)	+	++	++	+	++	++	+	+	+	++
12.helps to identify missing parts of the specification (+=a missing feature *was* found)	++	++	++	++	+	++	++	++	++	+
13.ability to identify inconsistencies (+=an inconsistency *was* found)	+	+	+	+	++	+	+	+	+	++
14.basis of description (+=oral presentation ++=presentation and documentation)	++	++	+	+	+	+	+	+	+	+
15.main focus of description (C=communication, S=state, P=performance)	C	C	C	C	C	C	CS	S	S	P
16.number of separate tools used (1=IDE – Integrated environment)	2	3	2	2	2	1	1	1	3	2
17.motivation (+=to understand the problem −=to exercise technique/tool)	+	−	−	−	−	−	−	+	−	+
18.conclusion about technique/tool (++=scales up, checks safety and liveness properties, −=not user friendly)	+	−	++	−	++	++	−	++	++	++

Table 1: Summary of the answers to the questions. Please note that these answers apply to our way of using the technique/tool, and for a particular purpose.

apply to the particular models we built. Our conclusions do not necessarily carry over to other modelling activities.

Question 18
What is the main conclusion about using your technique/tool?

epi In a sense the epi model falls in between two stools. It is not easy enough to use for real programmers. The need for real data structures to model the internal state of the processes makes the model unnecessarily complicated from the point of view of the theory.

latosπ is of limited use in its present state because the tool is not user friendly.

Prologπ Prolog is a good tool to implement specification languages, and to experiment with specifications written in such languages. The promise of finite state and infinite state model checking provides a further incentive to use Prolog with partial evaluation.

SuperVISE in its present form is not the most appropriate tool. It requires a user to:

- write VHDL-like code which is unfriendly and unnatural to software engineers.
- use a VHDL simulator

However, SuperVISE has some interesting and powerful features which have not been exposed here, notably multi-level modelling and the power of the SuperVISE interface which enables the execution of models assembled from components described in differing levels of detail.

Product-nets are useful for the specification of systems at the architectural level, to compare different designs, to search for errors in high level designs. A typical specification would be far removed from an actual implementation and currently it is not possible to automatically generate code.

Spin,Murϕ Spin is more powerful than Murϕ. Its notation is appropriate for modelling concurrent systems, even though the data structuring facilities are primitive. Murϕ would be useful as an alternative if Spin were not available, although it is unable to deal with liveness properties.

B is appropriate to specify a distributed database problem. Some training is required but once the method is understood, the tools are easy to use.

Z Our experience with modelling in Z shows that the abstract description is clear, concise, and useful. The most detailed description is perhaps not best done in Z because of the lack of support for concurrency in Z. The tools were found to be surprisingly easy to use, because they interwork well.

QNAP2 and Swap have a sound theoretical basis in queueing theory. The method is well established and of practical value. The Swap tool is currently under development. The user of the method/tools has to be skilled in the interpretation of the statistical results. We believe that the design of any distributed system should be guided by its performance characteristics. It is important to keep the models up-to-date after a system has been delivered, as hardware and software configurations will change, thus affecting the performance characteristics.

Table 1 summarises the main findings, which indicate that:

- The π-calculus gives the most elegant model of the application. Product-nets are more verbose but easier to understand because they are expressed graphically.
- Model checkers based on Linear Temporal Logic (Product-nets and Spin) give the most comprehensive information about the model.
- The integrated tools (B-tool, Spin and Product nets) provide the best support.
- The B method and tools provides the most appropriate path from abstract specification to an implementation.

Acknowledgements

This research was partially supported by a grant from ICL and by EPSRC grant GR/J08928.

We are grateful in particular to Nic Holt of ICL for comments on an earlier draft of the paper.

References

[1] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

[2] J.-R. Abrial, E. Börger, and H. Langmaack (eds.). *Formal Methods for Industrial Applications – Specifying and Programming the Steam Boiler Control, LNCS 1165.* Springer-Verlag, Berlin, 1996.

[3] A. O. Allen. *Probability, Statistics and Queueing Theory with Computer Science Applications.* Academic Press, New York, second edition, 1990.

[4] H. J. Burkhardt, P. Ochsenschläger, and R. Prinoth. Product nets — a formal description technique for cooperating systems. GMD-Studien 165, Gesellschaft für Mathematik und Datenverarbeitung (GMD), Darmstadt, Sep 1989.

[5] M. J. Butler. csp2B: a practical approach to combining CSP and B. Declarative Systems & Software Engineering Technical Report in preparation, Univ. of Southampton, 1999.

[6] M. Carlsson, J. Widén, J. Andersson, S. Andersson, K. Boortz, H. Nilsson, and Th. Sjöland. *SICStus Prolog Users's Manual.* Swedish Institute of Comp. Sci, Kista, Sweden, Jan 1993.

[7] D. L. Dill, A. J. Dexler, A. J. Hu, and C. H. Yan. Protocol verification as a hardware design aid. In *Computer Design: VLSI in Computers and Processors*, pages 522–525. IEEE Computer Society Press, Los Alamitos, California, 1992.

[8] S. Dumas, D. Boudigue, and G. Gardarin. Performance evaluation of distributed object architectures. In *Performance Tools 98*, Palma, Mallorca, 1998.

[9] P. H. Hartel. LATOS – a lightweight animation tool for operational semantics. Technical report DSSE-TR-97-1, Dept. of Electr. and Comp. Sci, Univ. of Southampton, England, Oct 1997. `www.ecs.soton.ac.uk/ ~phh/ latos.html`.

[10] P. Henderson. From formal models to validated components in an evolving system. Declarative systems & software engineering technical reports, Univ. of Southampton, 1998.

[11] M. A. Hewitt, C. M. O'Halloran, and C. T. Sennett. Experiences with PiZa, a Z animator. In J. P. Bowen, M. G. Hinchley, and D. Till, editors, *10th Z Users Meeting : The Z fomal sepcification notation (ZUM), LNCS 1212*, pages 37–51, Reading, UK, Apr 1997. Springer-Verlag, Berlin. `www.cs.reading.ac.uk/ zum97/`.

[12] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

[13] G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[14] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on software engineering*, 23(5):279–295, 1997. `http:// cm.bell-labs.com/ cm/ cs/ who/ gerard/`.

[15] I. S. C. Houston and S. King. CICS project report: Experiences and results from the use of Z in IBM. In S. Prehn and W. J. Toetenel, editors, *Formal software development methods (VDM '91) LNCS 551*, pages 588–596, Noordwijk, The Netherlands, Oct 1991. Springer-Verlag, Berlin.

[16] ICL. *VHDL+ User Guide 3.0.* International Computers Limited 1998, Apr 1998.

[17] Xiaoping Jia. *ZTC: A Type Checker for Z – User's Guide.* Dept. of Comp. and Inf. Sci, DePaul University,Chicago, Illinois, May 1995. `ftp.comlab.ox.ac.uk/ pub/ Zforum/ ZTC-1.3/ guide.ps.Z`.

[18] J. Jørgensen and M. Leuschel. Efficiently generating efficient generating extensions in Prolog. In O. Danvy, R. Glück, and P. Thiemann, editors, *Dagstuhl Seminar on Partial Evaluation, (LNCS 1110)*, pages 238–262, Schloß Dagstuhl, Saarbrücken, 1996. Springer-Verlag, Berlin.

[19] M. Leuschel. The ECCE partial deduction system. In G. Puebla, editor, *Tools and Environments for (Constraint) Logic Programming (ILPS)*, Port Jefferson USA, Oct 1997. Universidad Politécnica de Madrid Tech. Rep. CLIP7/97.1.

[20] M. Leuschel. Program specialisation and abstract interpretation reconciled. In J. Jaffar, editor, *Joint Conf. and Symp. Logic Programming (JICSLP)*, pages 220–234, Manchester UK, 1998. MIT Press, Cambridge, Massachusetts.

[21] M. Leuschel, B. Martens, and D. De Schreye. Controlling generalisation and polyvariance in partial deduction of normal logic programs. *ACM transactions on programming languages and systems*, 20(1):208–258, 1998.

[22] R. Milner. The polyadic π-calculus: a tutorial. In F. L. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*, pages 203–246. Springer-Verlag, 1993.

[23] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes part 2. *Information and Computation*, 100(1):41–77, Sep 1992.

[24] Z. Navabi. *VHDL : analysis and modeling of digital systems.* McGraw-Hill, New York, second edition, 1998.

[25] D. S. Neilson and I. H. Sørensen. The B-technologies: a system for computer aided programming. In U. H. Engberg, K. G. Larsen, and P. D. Mosses, editors, *6th Nordic Workshop on Programming Theory*, pages 18–35. BRICS, Denmark, Oct 1994.

[26] U. Nitsche. Application of formal verification and behaviour abstraction to the service interaction problem in intelligent networks. *Journal of Systems and Software*, 40(3):227–248, Mar 1998.

[27] U. Nitsche and P. Ochsenschläger. Approximately satisfied properties of systems and simple language homomorphisms. *Information Processing Letters*, 60(201-206), 1996.

[28] U. Nitsche and P. Wolper. Relative liveness and behavior abstraction (extended abstract). In *16th SIGACT-SIGOPS Symp. on Principles of Distr. Computing (PODC)*, pages 45–52, Santa Barbara, California, 1997. ACM, New York.

[29] P. Ochsenschläger. Die produktnetzmaschine. *Petri Net Newsletter*, 39:11–31, Aug 1991.

[30] P. Ochsenschläger, J. Repp, R. Rieke, and U. Nitsche. The SH-Verification tool – abstraction-based verification of co-operating systems. *Formal Aspects of Computing*, page to appear, 1998.

[31] J. L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, Englewood Cliffs, N. J., first edition, 1981.

[32] Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. Swift, and D. S. Warren. Efficient model checking using tabled resolution. In *Computer-Aided Verification (CAV'97), LNCS 1254*, pages 143–154. Springer-Verlag, Berlin, 1997.

[33] W. Reisig. *Petri Nets—An Introduction, EATCS Monographs in Theoretical Computer Science, Vol 4*. Springer-Verlag, Berlin, 1985.

[34] Simulog SA. *QNAP Reference Manual*. Simulog, St. Quentin, Paris, France, 1996. `http://www.simulog.fr/`.

[35] M. Saaltink. The Z/EVES system. In J. Bowen, M. Hinchey, and D. Till, editors, *10th International Conference of Z Users (ZUM), LNCS 1212*, pages 72–85, Reading, UK, Apr 1997. Springer-Verlag, Berlin. `www.cs.reading.ac.uk/ zum97/`.

[36] J. M. Spivey. *The Z notation*. Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[37] D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In J.-P. Jouannaud, editor, *2nd Functional programming languages and computer architecture, LNCS 201*, pages 1–16, Nancy, France, Sep 1985. Springer-Verlag, Berlin.

A The polyadic π-calculus model animated by Epi

The centre (below) inputs (using the `in` operation) from channel `s` the name of a customer `b` and the name of a return channel `r`. The centre then outputs (`out` operation) on channel `r` the name of the customer, together with the service `pm.at(b)`. Here `pm` is an object representing a set of pairs mapping customers to their services. The method call `at(b)` selects the service due to customer `b`, or `nil` if the customer is not represented in the data base `pm`. The recursive call to the agent `centre` records the fact that customer `b` has now been served, by deleting the singleton set `set().add(b)` from the domain of `pm`.

```
centre(pm,s):=
 in(s,(b,r)::out(r,(b,pm.at(b)),
   centre(pm.ds(set().add(b)),s))).
```

At the first line of the agent below, an office inputs from channel `r` the name of a customer (`b`) and a return channel to the customer (`c`). At the second line of the agent, the method `member(b)` decides whether the customer is known to the office by checking if `b` is in the domain (`pm.dom()`) of the set of maplets `pm`. If the customer is known to the office, the latter provides the service directly (third line of the agent beginning with the keyword `then`). Otherwise the office consults with the centre (fourth line of the agent beginning with `else`).

```
office(s,pm,r):=
 in(r,(b,c)::pm.dom().member(b)
   then
    out(c,pm.at(b),office(s,pm.ds(set().add(b)),r))
   else
    out(s,(b,r),in(r,(b,m)::out(c,m,office(s,pm,r))))).
```

A customer (below) chooses an office from the set `offices`. The `alt` method binds the name `r` to the identity of the chosen office. The interaction with the chosen office consists of sending the office the identity of the customer (`b`), the name of the return channel (`c`) and then to receive the service (`x`) on the return channel. The customer expires (`skip()`) once served.

```
customer(b,c):=
  offices.alt(r::out(r,(b,c),in(c,x::skip()))).
```

The model is brought to life by creating six concurrent processes: the centre, two offices and three customers. The system is configured as shown in Figure 1. In the agent definition below, the set `pmc` represents the data base for the centre, and the sets `pm1` and `pm2` represent the data bases for the offices. Channel `0` is used for communications with the centre, channels `11` and `12` are used by both the customers and the centre to communicate with the offices, and channels `91` to `93` are the return channels for the customers.

```
system:=
  par(centre(pmc,0),
    par(
      par(office(0,pm1,11),office(0,pm2,12)),
      par(customer(1,91),
        par(customer(2,92),customer(3,93))))).
```

A.1 Epi animation results

The Epi GUI looks like a spreadsheet. Users enter expressions describing relevant information about the state of a model and Epi calculates and shows the corresponding values. To animate a specification the user requests statements to be executed that alter the state of the system. The GUI provides a button with each statement so that it may be activated. Neither the expressions nor the statements are fixed. The user is free to program appropriate expressions for evaluation, and/or statements for controlling the animation.

Figure 2 shows three screenshots, produced by tracing the progress of customer 1 through the system. Initially there is a menu of six possible choices, two choices for each of the three customers. The value of expression `p.menu().members()` shows the list of six choices. The list is displayed in a scrolling window, so that only the beginning is visible. Here `(11 (1 91))` means that the system is ready to transmit the message consisting of the tuple `(1 91)` on channel `11`. The expression `p.menu().size()` shows the length of the menu.

Activating the button labelled `p:=p.arbstep()` produces the state shown in the second screenshot. This makes an arbitrary choice out of the menu of possibilities. Apparently the system has allowed customer 1 to communicate with office 1. This reduces the number of choices to 3, because office 1 is now preparing to serve customer 1 (action `(91 45)`). Customers 2 and 3 may only communicate with office 2.

The next step is to activate the button labelled `p:=p.arbstepc(91)`. This makes an arbitrary choice of the events enabled on channel `91`. The selection causes office 1 to complete serving customer 1, so that now office 1 is available again for the remaining two customers. Since customer 1 leaves the system after being served, there are only four choices left.

Our example shows animations being driven 'forward', but they can also be rolled back. This is achieved by selecting the button labelled `p:=back()`. Rewinding animations is useful when alternative behaviours are explored.

The user is thus able to explore the model, and may do so in a rather convenient and familiar (spreadsheet-like) setting. Epi also offers search facilities but we will not discuss those here.

B The monadic π-calculus model animated by Latos

The centre (below) initially knows about one customer `u`, and listens (using the input operation `?`) on a channel `s` for requests from offices. The centre expects to receive the name of a private communication channel `q`. The centre then reads the name of a customer from channel `q` into `b` and the name of a return channel into `r`. The name matching operation `[b=u]` ensures that when `b` matches `u`, the 'name' `42` is sent (using the output operation `!`) over the return channel `r`. The recursive call `Centre(_,s)` restarts the centre, this time without knowledge of the customer. This models the fact that once served, a customer cannot be served again.

```
Centre(u,s)=
  s?q.q?b.q?r.[b=u]r!42.Centre(_,s);
```

An office initially knows about two customers `v` and `u`. An office listens to customers on channel `r` and it talks to the centre over channel `s`. An office expects to receive the name of a private communication channel `q` then reads the name of a customer into `b` and the name of a return channel into `c`. If `b` matches `v`, the 'name' `42` is sent over the return channel, and the office starts over again, this time without knowledge of the customer `v`, again, this models the fact that once served, a customer cannot be served again. Otherwise (`+`), if the customer matches `u`, the office embarks on a communication round with the centre as described above. Eventually the office either returns '42' to the centre, which then passes it on to the customer, or the center dies, blocking the office as well.

```
Office(s,v,u,r)=
  r?q.q?b.q?c.(
  [b=v]c!42.Office(s,_,u,r) +
  [b=u]((o)(q)s!q.q!b.q!o.o?m.c!m.Office(s,v,_,r)));
```

A customer knows its own identity `b` and it attempts to communicate with two offices `x` and `y`. The name `q` is a private channel used to communicate the identity of the customer to the office, whereas the private channel `c` is used to receive the service from the office. The last action of each of the two alternatives in the customer's agent specification is calling calling an agent `S`, with the identity of the customer and the name of the service provided by the office as parameters. The model does not provide a definition for this agent, so the customer processes deadlock once `S` has been called. This arrangement has been used to enable scenarios to be identified that represent a customer who has been provided with the requested service.

```
Customer(b,x,y)=
  ((c)(q)x!q.q!b.q!c.c?x.S(b,x))+
  ((c)(q)y!q.q!b.q!c.c?x.S(b,x));
```

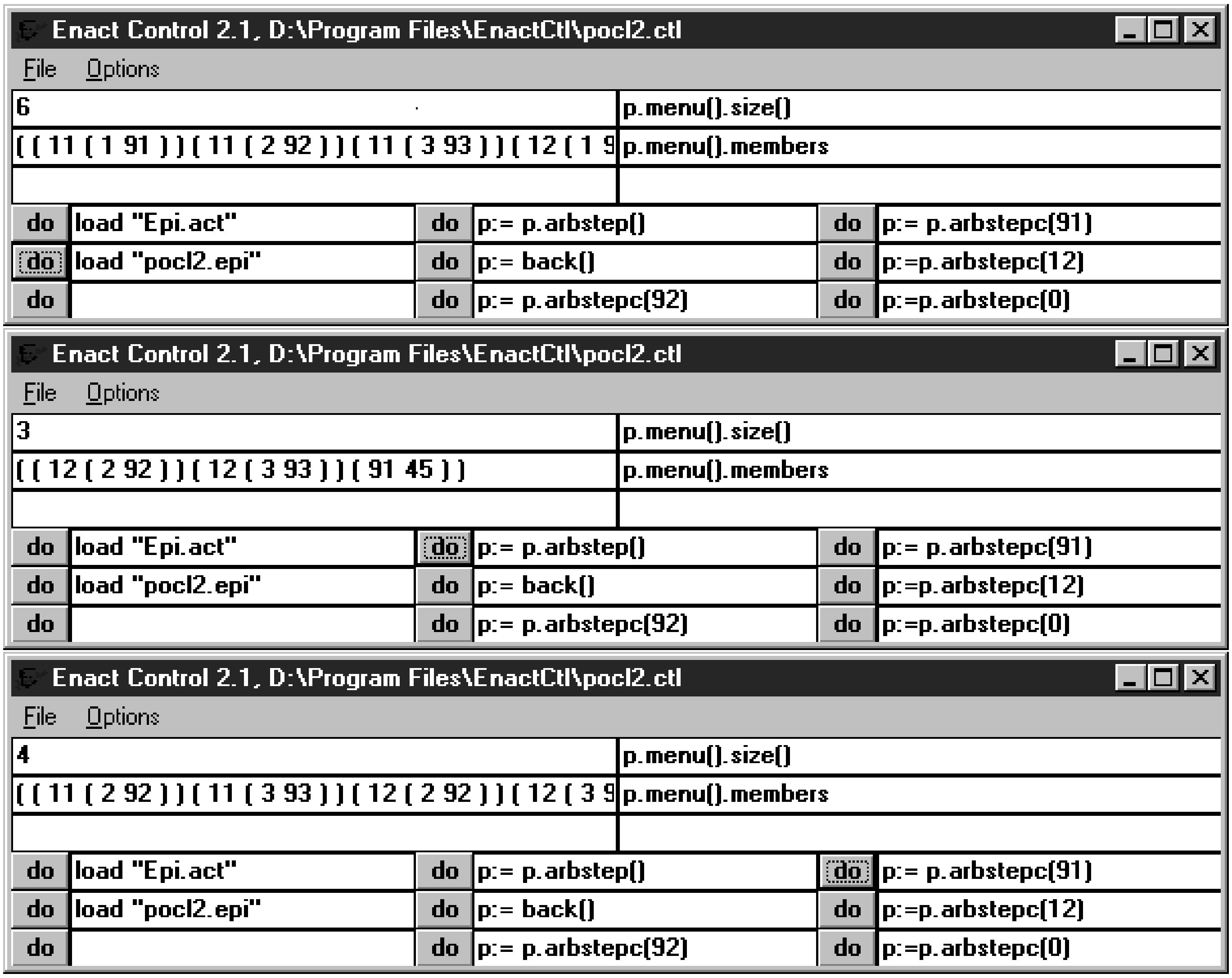

Figure 2: A sample Epi animation.

The `system` agent starts six concurrent ($|$) processes: the centre, the two offices and the three customers in a configuration as illustrated in Figure 1.

```
system=
  (c1)(c2)(c3)(o1)(o2)(s0)(Centre(c3,s0)|
  Office(s0,c1,c3,o1)|Office(s0,c2,c3,o2)|
  Customer(c1,o1,o2)|Customer(c2,o1,o2)|
  Customer(c3,o1,o2))
```

B.1 Animation results

Latos animations show the evolution of the model, by listing all the possible interactions that may take place at each step. This amounts to a breadth first search of the state space. The earliest possibility for customer $n \in [1, 2]$ to receive her service is at step 4. The individual steps involved are:

1. a private channel is established between customer n and her home office n;
2. customer n sends her identity over the private channel;
3. a private *return* channel is established between customer n and her home office n;
4. the service (`42`) is returned to the customer.

Both customers 1 and 2 have only one way of being served, but customer 3 has two ways of receiving service. However, she has to wait until step 8 for the earliest opportunity to be served because the offices have to consult with the centre. The latos animations show that customer 3 may receive her service via two routes, one taking four steps when Customer 3 and Office 1 communicate, and a further four steps to allow Office 1 and the Centre to communicate. The second route is analogous, but uses office 2 instead.

Up to 54 interleavings of these four scenarios are possible (considering as equivalent agent expressions that are α-convertible), taking up to $8 + 4 = 12$steps. This makes the latos animations verbose and time consuming to calculate (40 minutes on a SPARCstation-20 @ 60MHz, 128MB, Solaris 2.5). Latos is thus less than optimal for experimenting with larger models.

C The Prolog-π Model

The Prolog-π model of the distributed data base problem is expressed in the π-calculus. The novelty of the approach lies in implementing the π-calculus in Prolog, thereby making existing Prolog technology available for the animation, formal verification and validation. (The same can also be achieved for other formalisms such as CCS or Petri Nets.)

The constructs of the π-calculus we have used are as follows (for convenience we have added a "non-match" construct $[x \neq y]P$ to the standard π-calculus):

- "$\mathbf{0}$" *inaction* (a process that does nothing),
- "$P_1 + P_2$" *choice* between two processes P_1 and P_2,
- "$\bar{c}x.P$" *output* value x on channel c and continue with process P,
- "$c(x).P$" *input* value x on channel c and continue with process P (an input value x can synchronise with an output value x on the same channel, resulting in [for the outside world] the silent action τ),
- "$P_1 \mid P_2$" *parallel composition* of two processes P_1 and P_2,
- "$(x)P$" *restricting* process P from performing input or output on channel x (except for internal synchronisation τ),
- "$[x = y]P$" only enable P if the *match* $x = y$ succeeds,
- "$[x \neq y]P$" only enable P if the *match* $x = y$ fails, and
- "$A(y_1, \ldots, y_n)$" behaves like the process defined by an *agent defining equation* $A(x_1, \ldots, x_n) =_{def} P$ (this enables recursion).

An interpreter for this variant of the π-calculus can be written in Prolog, thanks to its built-in mechanisms for unification and non-determinism. The particular interpreter we have used is shown in Figure 4. Note that, for simplicity, we have not (yet) implemented the full scoping rules of the π-calculus, as they were not required for our particular problem.

The predicate `trans/3` in Figure 4 has as arguments a π-calculus expression, an action, and the π-calculus expression resulting from applying the action to the first expression. This predicate defines the behaviour of every π-calculus construct in a straightforward manner. For instance the first clause can be read as: "an expression X+Y can perform the action A resulting in the new expression X1 if the sub-expression X can perform the same action A leading to X1." Based on this predicate, Figure 4 also comprises a simple depth-first trace facility (`trace/2`) defined in just 2 clauses. We will show more elaborate tools below.

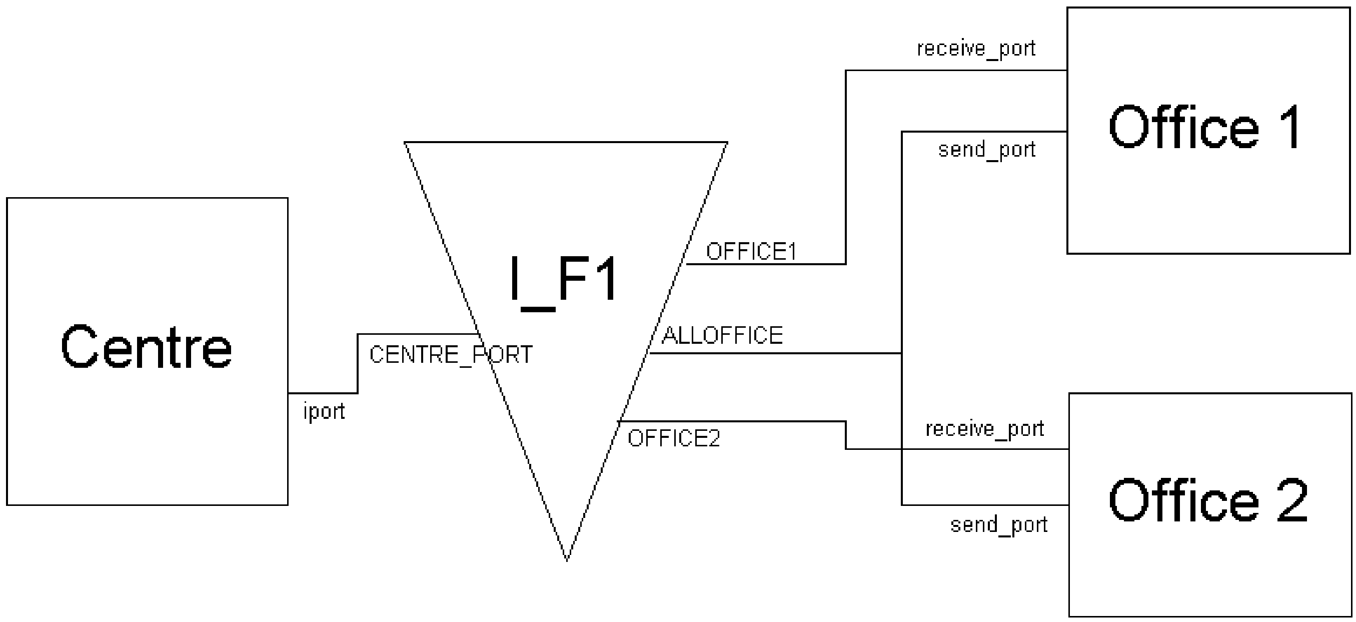

Figure 3: The architecture of the SuperVISE model.

```
trans(X + _Y,A,X1) :- trans(X,A,X1).
trans(_X + Y,A,Y1) :- trans(Y,A,Y1).

trans(eq(V1,V2,X),A,X1) :- V1=V2,trans(X,A,X1).
trans(neq(V1,V2,X),A,X1) :- \+(V1=V2),trans(X,A,X1).

trans(!(V,Ch,X),  V!Ch,X).
trans(?(V,Ch,X), V?Ch,X).

trans(X//Y, V<->Ch, X1//Y1) :-
     trans(X, V!Ch, X1), trans(Y, V?Ch, Y1).
trans(X//Y, V<->Ch, X1//Y1) :-
     trans(X,V?Ch,X1), trans(Y,V!Ch,Y1).
trans(X//Y, A, X1//Y ) :- trans(X,A,X1).
trans(X//Y, A, X//Y1 ) :- trans(Y,A,Y1).

trans(agent(X),A,X1) :- agent(X,AE),trans(AE,A,X1).

trans((Expr-CList), A, (X-CList) ) :-  /* restrict */
     trans(Expr,A,X),
     \+(hidden(A,CList)).

hidden(_V ? Ch, CList) :- member(Ch,CList).
hidden(_V ! Ch,  CList) :- member(Ch,CList).

trace(Expr,[A]) :- trans(Expr,A,_NewExpr).
trace(Expr,[A|Res]) :-
    trans(Expr,A,NewExpr),trace(NewExpr,Res).
```

Figure 4: An interpreter for the π-calculus (without scope rules)

C.1 Describing the Model

We will now present a first attempt at formalising the distributed data base problem in our version of the π-calculus. For this we will define the following types of agents:

1. Customers, which ask for a service and then wait for the service, (modelled here as a payment);
2. Thieves, which repeatedly ask and then wait for services;
3. Centres, which coordinate offices and do not handle service requests directly;
4. Offices, which have a small database of customers they know (actually only one customer they know) and which can handle service requests by customers and database requests by centres as well as requesting information about unknown customers from centres.

Our first attempt at modelling these agents was as follows:

1. $Customer(Id, O) =_{def}$ $\bar{O}\,service\_request(Id).O(Service).\mathbf{0}$
2. $Thief(Id, O) =_{def}$ $\bar{O}\,service\_request(Id).O(Service).Thief(Id, O)$
3. $Centre(P_1, P_2) =_{def}$ $P_1(db\_request(Id)).\bar{P}_2 db\_request(Id).$

$(P_2(ok(Id)).\bar{P}_1 ok(Id).Centre(P_1,P_2)+$
$P_2(ko(Id)).\bar{P}_1 ko(Id).Centre(P_1,P_2))$
$+\ P_2(db\_request(Id)).\bar{P}_1 db\_request(Id).$
$(P_1(ok(Id)).\bar{P}_2 ok(Id).Centre(P_1,P_2)+$
$P_1(ko(Id)).\bar{P}_2 ko(Id).Centre(P_1,P_2))$

4. $Office(C,P,DB) =_{def}$
$C(db\_request(Id)).([Id = DB]\bar{C}ok(Id).Office(C,P,\mathbf{0})+$
$[Id \neq DB]\bar{C}ko(Id).Office(C,P,DB))+$
$P(service\_request(Id)).$
$([Id = DB]\bar{P}money(Id).Office(C,P,\mathbf{0})+$
$[Id \neq DB]\bar{C}db\_request(Id).$
$(C(ok(Id)).\bar{P}money(Id).Office(C,P,DB)+$
$C(ko(Id)).\bar{P}zilch(Id).Office(C,P,DB))\ )$

The arguments of the agents are as follows. Customers and thieves carry an identifier *Id* as well as a channel *O* on which they will communicate with a particular office. Centres carry two channels P_1, P_2 via which they are connected to 2 offices. Offices carry two channels C, P for communicating with the centre and the public respectively, as well as a database DB. This database either contains a customer identifier, meaning the customer is eligible for a service, or the empty value 0.

The Prolog encoding of these agents is straightforward. For example, the encoding of customers and thieves is simply accomplished by defining the following Prolog facts (which will be used by the interpreter in Figure 4):

```
agent(customer(CustID,Office),
     !(service_request(CustID),Office,
       ?(_Service,Office,stop))).
agent(thief(CustID,Office),
     !(service_request(CustID),Office,
       ?(_Service,Office,
          agent(thief(CustID,Office)) ))).
```

Based on these basic agents we can now formalise a full-fledged system. For instance, the system depicted in Figure 5 can now be described simply by the following Prolog term:

```
agent(thief(666,p2)) //
agent(customer(702,p1)) // agent(office(c2,p2,702)) //
agent(office(c1,p1,701)) // agent(centre(c1,c2))
-[p1,p2,c1,c2]
```

C.2 Some More Advance Tools

Let us now exploit our Prolog implementation of the π-calculus and animate the system in Figure 5, as well as try to verify some properties of it. For instance, we might want to be sure that the thief 666 will never get money from any office. To do this we can define what it means for a state to be unsafe, by defining the predicate `unsafe/1` as follows:

```
unsafe(Expr) :-
  trans(Expr, money(666)<->_Channel, _NewExpr).
```

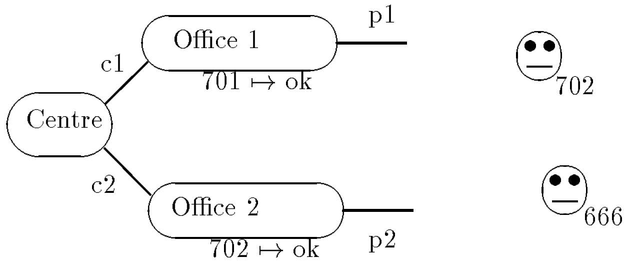

Figure 5: Distributed system, to be encoded in the π-calculus

We can now implement predicates which search for such unsafe states in a straightforward manner, thanks to Prolog's search facilities and support for non-determinism. Below are two tools which achieve this task. The first tool is an interactive tracer which allow the user to select among possible actions.

```
itrace(Expr) :-
   (unsafe(Expr) -> print(' UNSAFE !!') ; true),
   findall((B,NE),trans(Expr,B,NE),Options),
   print_options(Options),nl,
   print('=> '),
   prompt(_OldPrompt,''),
   read(Nr),
   member_nr((_,NewExpr),Options,Nr),
   itrace(NewExpr).

print_options(Os) :-
   print('Available Actions:'),nl,
   member_nr((Opt,_),Os,N),
   print('   '),print(N), print('. '),
   print(Opt),nl,fail.
print_options([]) :- !,print('   None !'),nl.
print_options(_).

member_nr(X,[X|_],1).
member_nr(X,[_|T],N) :-
   member_nr(X,T,TN), N is TN + 1.
```

The second tool performs a breadth-first search without user interaction. More sophisticated search and tracing facilities can be implemented with relatively little effort.

```
bf_trace(Expr,Trace) :-
   bf([(Expr,[])],Trace).

bf([(_Expr,TrSoFar)|_T],TrSoFar).
bf([(Expr,TrSoFar)|T],Trace) :-
   findall((NewExpr,NewTrace),
           (trans(Expr,A,NewExpr),
            append(TrSoFar,[A],NewTrace)),
           Ns),
   append(T,Ns,NewL),
   bf(NewL,Trace).
```

C.3 A Sample Trace leading to deadlock

In this section we present a sample trace of the earlier defined π-calculus model of Figure 5, exhibiting a fundamental error in our model. First we define the following auxiliary predicate, making use of our earlier defined interactive tracer:

```
iv :- PT = p2, PC = p1,
 itrace((agent(thief(666,PT)) //
 agent(customer(702,PC)) // agent(office(c2,p2,702)) //
 agent(office(c1,p1,701)) // agent(centre(c1,c2))
)-[p1,p2,c1,c2]).
```

The execution trace of the query `?-iv.` under Sicstus Prolog is now as follows::

```
| ?- iv.
Available Actions:
   1. service_request(666)<->p2
   2. service_request(702)<->p1

=> 1.
Available Actions:
   1. service_request(702)<->p1
   2. db_request(666)<->c2

=> 1.
Available Actions:
   1. db_request(666)<->c2
   2. db_request(702)<->c1

=> 1.
Available Actions:
   None !
```

In concrete terms, the above trace corresponds to the following scenario. In the first action, the thief has requested a service from office p2. Then, customer 702 has requested a service from office p1. At the third stage the office p1 has requested information from the centre about customer 666. We are now at a stage where the office p1 is waiting for a reply from the centre, and is thus unavailable for any other action. Also, the centre and the office p2 try to communicate, but fail to do so: the centre wants to request information about customer 666 from p2 while p2 wants to request information about customer 702 from the centre. No actions are available at this stage and we have a deadlock.

C.4 The improved agent definitions (no deadlock)

Having identified the deadlock, we can try to remedy the situation. Basically, we have to give offices the ability to handle customers and database requests (from the centre) at the same time. Similarly, a centre has to be able to handle multiple requests at the same time. The following adaptation of the above example achieves this:

```
agent(ioffice(Centre,Public,DB),
     agent(office_customer(Centre,Public,DB))
     //
     agent(office_dbserver(Centre,Public,DB))).
   /* 2 processes: to handle Centres + Customers */
agent(office_customer(Centre,Public,DB),
     ?(service_request(CustID),Public,
          eq(CustID,DB,!(money(CustID),Public,
          agent(office_customer(Centre,Public,0))))
          +
          neq(CustID,DB, !(db_request(CustID),Centre,
           ?(ok(CustID),Centre,
             !(money(CustID),Public,
        agent(office_customer(Centre,Public,DB))))
           +
           ?(ko(CustID),Centre,
             !(zilch(CustID),Public,
        agent(office_customer(Centre,Public,DB))))
             ))
        )
     ).
agent(office_dbserver(Centre,Public,DB),
     ?(db_request(CustID),Centre,
          eq(CustID,DB,!(ok(CustID),Centre,
          agent(office_dbserver(Centre,Public,0))))
          +
          neq(CustID,DB,!(ko(CustID),Centre,
          agent(office_dbserver(Centre,Public,DB))))
          )
      ).
agent(icentre(P1,P2),
     agent(centre_line(P1,P2))
     //
     agent(centre_line(P2,P1))).
   /* 2 processes: to handle 2 Centres */
agent(centre_line(P1,P2),
   ?(db_request(CustID),P1,!(db_request(CustID),P2,
        ?(ok(CustID),P2,!(ok(CustID),P1,
           agent(centre_line(P1,P2))))
     +
        ?(ko(CustID),P2,!(ko(CustID),P1,
           agent(centre_line(P1,P2))))
     ))).
```

C.5 Conclusion and Outlook

We have shown that Prolog is a good tool to implement specification languages, and one can experiment with specifications written in such languages with little effort. However, the standard execution mechanism of Prolog is actually too weak to easily implement more sophisticated verification tools such as model checkers. Fortunately, some more recent implementations of Prolog incorporate improvements such as loop checking and tabling. These allow one to easily perform *finite state model checking* in an efficient manner [32]. Furthermore, ongoing research (e.g., [20]) aims at achieving *infinite state model checking* by adapting and enhancing existing technology for the analysis and optimisation of Prolog programs. We believe that, once these tools are in place, Prolog will prove to be a very generic and flexible platform for the formal specification and verification of software designs.

D The SuperVISE model

The parts of the model are represented in SuperVISE and understood by the system as modules of an item of hardware.

These modules are connected by an 'interface'. The definition of a SuperVISE interface describes its ports and the messages which may pass through it. The description of a message defines which port it is received from and to which port it is sent. Other modules (including other interfaces) may be connected to the ports. A SuperVISE interface is able to do much more than simple passing of messages, but this is all that was used in this model. SuperVISE communications are broadcast to all modules connected to a port and non-synchronous.

The interface is a SuperVISE concept. It does not appear in the executable model as the SuperVISE compiler replaces interface with VHDL code in which compiler-generated modules communicating using signals reproduce the behaviour of the interface.

In the code, the model is described in two sections. The first describes the behaviour of the modules, the second describes how these modules are connected.

When running the model in a suitable simulator, all of the modules execute in parallel (as is normal in hardware).

The initial model of this problem comprised four pieces: a Centre, two Offices and an interface connecting them. The identity of the customer considered to be at an office is determined by setting the value of a variable in the Office module. For the initial model, the two offices were coded separately. When the 'go' signal is received by an Office, it first checks the identity of its Customer and looks in its internal data for that Customer.

If the Customer is known, the payment authority is revoked and a message is output indicating the Customer being paid.

If the Customer is not known, a message is sent to the Centre detailing the identity of the Office and that of the Customer. Because of the nature of message passing in SuperVISE, the Office resends the message periodically until an acknowledgement is received from the Centre. The Office then listens (on a private channel from the Centre) for a response from the Centre containing payment authority for the Customer. When this is received a message is output detailing the payment.

The Centre waits for a message from one of the Offices. When a message is received, it first sends an acknowledgement. Next it examines the message to identify the Customer and the Office. It then looks up that Customer in its data and sends a message on the appropriate Office's channel containing the amount to pay to the Customer and deletes the authority. The Centre then waits for another message from and Office. The Centre outputs a variety of messages so that the modeller may see the progress of the model as it executes.

E The Product Net Model

The Product net model of the considered scenario is a specification using high-level Petri nets (Petri nets including data types) that are called Product Nets [4]. In the subsequent description of the specification we assume a basic familiarity with Petri nets.

The specification is divided into 5 subnets that are combined by identifying commonly named places with one another. Since customers and offices occur in the application that we consider, we model a pool of customers and offices that occur in the specification. These pools are the places `customers` and `offices`. The first subnet of the specification (Figure 6) simply describes that customer `c` goes to office `o` by firing transition `go_to_office`.

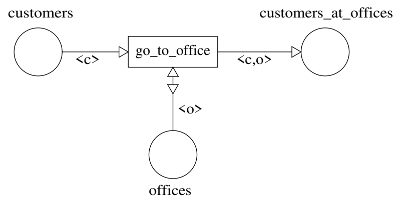

Figure 6: The first part of the specification: the customer goes to the office.

In Figure 7, a part of the offices' behaviour is modelled: If a customer is at an office (customer/office pair in place `customers_at_offices`) and the customer is known to the office (a corresponding customer/office pair in place `customer_is_known`) then she can be paid immediately by firing transition `pay_customer`. (For the purpose of this specification we assume that providing a service can be represented as paying an amount of money.)

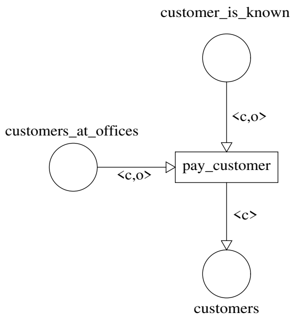

Figure 7: The second part of the specification: Payment at the home office of the customer.

If the customer is not known to the office (modelled by an inhibitor arc from place `customer_is_known` to transition `request_information` in Figure 8), a request for information about this customer is sent to the approving centre.

Figure 9 represents the centre's behaviour. If it receives

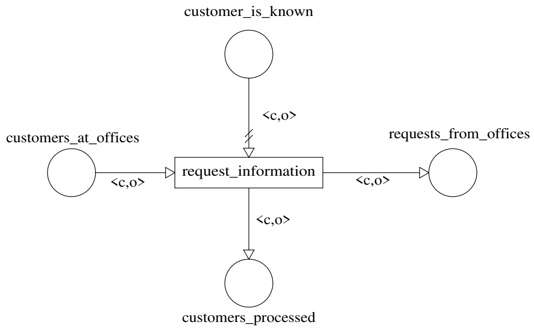

Figure 8: The third part of the specification: Customer request is referred to the centre.

a request from an office, it sends an `OK`-message or a `NO`-message depending on whether it could find information about the eligibility of the customer to receive requested payment.

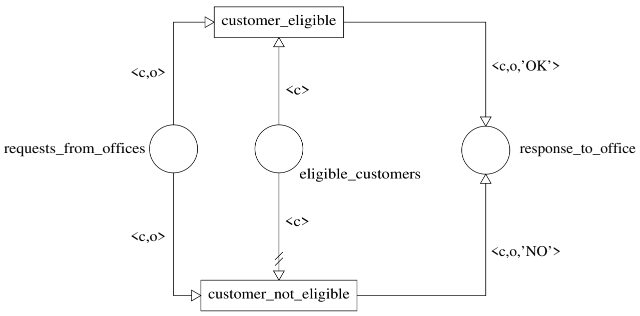

Figure 9: The fourth part of the specification. The centre decides whether or not to authorise payment to the customer.

Depending on the centre's response the customer may be paid at the office. This situation is modelled in Figure 10 by firing either transition `pay_customer_after_request` or `do_not_pay_customer`.

The specification has been analysed and validated using the integrated tool suite Product Net Machine / SH-Verification Tool [30]. First an exhaustive construction of the specification's state-space was done. The system has an initial state comprising of three customers and two offices. Two of the customers are known to one of the offices respectively and one customer is known to be eligible to receive payment by the centre. After an abstraction step [26] that only focussed on relevant actions of the specification with respect to customer service, the state-space was decreased from 2173 states to 9 states. In the abstraction step, events of the specification are either renamed or completely ignored. Renaming is used when different actions need not be distinguished and so get a common name. Actions are ignored when they are not relevant to the customer's point of view.

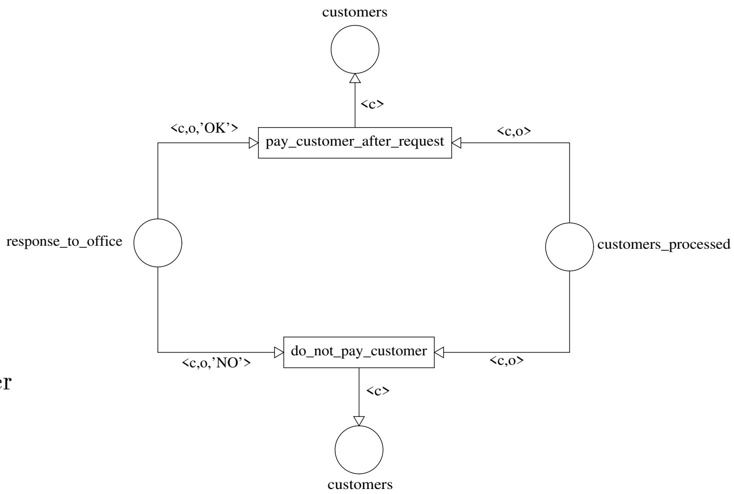

Figure 10: The fifth part of the specification: The centre responds to the office.

After changing the events, standard minimisation techniques for automata are applied to compute the minimal abstract state space.

Using verification techniques described in [27, 28] the specification was validated by model-checking temporal properties. The temporal property that a customer is only paid after a request from an office to the centre if the centre has given its approval, for instance, has been checked to hold for the specification. Another property, saying that eligible customers will be paid eventually, has also been checked to be true for the specification when considering fairness assumptions (*approximate satisfaction* of the property [27, 28]).

The fairness notion in the definition of approximately satisfied properties (also called relative liveness properties) states that all finite computations of the specification can be continued to an infinite computation that satisfies the property. It can be shown that this notion of fairness is close to the notion of strong fairness [28].

F The Spin Model

Below is the basic model written in Promela, the modelling language used by Spin. Firstly the database is modelled as a pair of arrays indexed by customer number - a boolean to record whether or not that customer's information is present, and an integer specifying the amount due. The variables `pmc`, `pm` and `ac` represent the centre's database, the offices' databases and the customers' personal accounts respectively. All are initialised to zero; the interesting values are set up by the `init` process (below).

```
#define NoOFCUSTOMERS 3
#define NoOFOFFICES   2
```

```
typedef Database {
      bool present[NoOFCUSTOMERS];
      byte amount[NoOFCUSTOMERS]
}

Database pmc = 0;
Database pm[NoOFOFFICES] = 0;
byte ac[NoOFCUSTOMERS] = 0;
```

F.1 The Centre

The centre is modelled as a server process `Centre` below, which sits in an infinite loop waiting for a request on its input channel `s`. This loop is marked with the label `end` to indicate that it is a valid end state; i.e. that it is not an error for the process to remain blocked at this point indefinitely.

The request (from one of the offices) consists of a customer number `b` and a reply channel `r`. If information about customer `b` is present in the centre's database, the amount due to that customer is sent on the reply channel and the customer is removed from the database. The second alternative of the if statement specifies that if the customer is not known to the centre the value zero is returned; however this alternative has been commented out, reflecting the missing functionality in the original π-calculus model. As a result, if the centre receives a query about a customer it does not know about, it will block and deadlock will occur.

```
proctype Centre (chan s)
{     byte b;  chan r;
 end: do
      ::    s?b,r;
            if
            ::    pmc.present[b]  ->
                        r!pmc.amount[b];
                        pmc.present[b] = false;
      /*    ::    !pmc.present[b]  ->
                        r!0   */
            fi
      od;
}
```

F.2 The Offices

Each office is modelled by a server process (`Office` below) in the same way as the centre. An individual office is distinguished by its identifier `n`, and uses the channels `r` and `s` to receive requests from customers and to send queries to the centre, respectively. The request consists of a customer number `b` and a reply channel `c`. If the office knows about customer `b`, it sends the specified amount on the reply channel and removes the customer from its database; otherwise it queries the centre, and passes on the reply from the centre to the customer, using an unbuffered channel `t` to communicate with the centre.

```
proctype Office (byte n; chan s, r)
{      byte b, x;  chan c;
       chan t = [0] of {byte};
 end:  do
       ::    r?b,c;
             if
             ::    pm[n].present[b]  ->
                         c!pm[n].amount[b];
                         pm[n].present[b] = false;
             ::    !pm[n].present[b]  ->
                         s!b,t;  t?x;  c!x;
             fi
       od;
}
```

F.3 The Customers

Each customer is modelled by a process (`Customer` below) whose parameters are an identification number `n` and two channels to allow it to communicate with either of the two offices. The customer makes a nondeterministic choice between the offices, engages in a transaction with the chosen office by sending its customer number `n` and the name of a reply channel `c` on the appropriate channel, and waits for a reply. The amount received is stored in the customer's account (simply to make it globally visible during simulation), and the customer process then terminates.

```
proctype Customer (byte n; chan t0, t1)
{     chan  c = [0] of {byte};
      byte  x = 0;
      if
      ::    true  ->  t0!n,c;  c?x
      ::    true  ->  t1!n,c;  c?x
      fi;
      ac[n] = x;
}
```

F.4 Initialisation of the processes

The `init` process is the first one to be run, and sets up the model. First it initialises the databases so that office 0 knows about customer 0, office 1 knows about customer 1, and the centre knows about customer 2. It then starts six concurrent processes representing the centre, two offices and three customers in the configuration shown in Figure 1.

```
init
{     chan  s = [0] of {byte, chan};
      chan  t0 = [0] of {byte, chan};
      chan  t1 = [0] of {byte, chan};

      atomic {
          pmc.present[2]   = true;
```

```
        pmc.amount[2]    = 42;
        pm[0].present[0] = true;
        pm[0].amount[0]  = 42;
        pm[1].present[1] = true;
        pm[1].amount[1]  = 42;

        run Centre (s);
        run Office (0, s, t0);
        run Office (1, s, t1);
        run Customer (0, t0, t1);
        run Customer (1, t0, t1);
        run Customer (2, t0, t1);
    }
}
```

F.5 Analysis of the Spin model

This model was analysed using Xspin running under Linux/X-Windows on a Pentium. After performing a syntax check, a check was made for violations of safety properties. This uncovered the possibility of deadlock owing to the missing functionality in the centre, and generated an error trace, which could then be used to guide a simulation to reconstruct the execution sequence that led to the error. Figure 11 shows a screen dump from Xspin part way through this guided simulation. On the left of the screen, the top window is the main control window and displays the Promela source; below it is a window showing the results of the verification: "invalid endstate". The three windows on the right of the screen show the guided simulation in progress. The top one shows the detailed execution trace, while the lower ones show the sequence of messages sent between processes and the values of the global variables. All three of these are continuously updated as the simulation progresses.

Figure 12 shows the complete message sequence chart from the above simulation, which clearly illustrates where the error occurs. (On the screen, the boxes on the message sequence chart can be cross-referenced to the corresponding lines of source code in the control window simply by positioning the cursor over them.)

Spin proved to be highly effective in exploring the distributed database/client-server problem, because the nature of the problem maps well onto the modelling facilities provided by Promela, and because of the powerful and user-friendly interface provided by Xspin. A number of refinements of the basic model were investigated, and liveness as well as safety properties established using Xspin's linear-time temporal logic property manager. Spin is clearly an effective and appropriate tool for this kind of modelling, and could usefully be applied to larger problems (although as with any model checker, some care in modelling is needed to keep the time and space requirements within manageable bounds).

G The Murϕ model

Below is the basic model in Murϕ. The description consists of a series of constant, type and variable declarations, followed by a set of transition rules which define the behaviour of the system. Each rule is a guarded command, consisting of a condition (a Boolean expression on the global variables) and an action. The execution model is to repeatedly evaluate the conditions, choose (nondeterministically) one rule whose condition evaluates to true, and execute the corresponding action. The execution of the actions is atomic - all interleaving of executions is specified explicitly by breaking a process down into a set of rules.

The first section declares some constants, types and variables. The database is modelled as an array of records, where each record has two fields: a boolean to record whether or not that customer's information is present, and an integer specifying the amount due. The variables `pmc` and `pm` represent the centre's database and the offices' databases respectively. `CustStat`, `OffStat` and `CentreStat` are state variables for the customers, offices and centre respectively. The remaining variables are used to model the interactions between customers and offices, and between the offices and the centre.

```
const NoOfCustomers:  3;
      NoOfOffices:    2;

      NoOne:          0;

type  CustRng:      NoOne..NoOfCustomers;
      RealCustRng:  1..NoOfCustomers;
      OffRng:       1..NoOfOffices;

      CustType:   enum {Pennyless, Waiting, Done};
      CentreType: enum {Ready, Busy};
      OfficeType: enum {Free, Serving,
                        Querying, Awaiting};

      Data:       record
                     present:  boolean;
                     amount:   0 .. 255;
                  end;
      Database:   array [CustRng] of Data;

var   pmc:  Database;
      pm:   array [OffRng] of Database;

      CustStat:   array [RealCustRng] of CustType;
      OffStat:    array [OffRng] of OfficeType;
      CentreStat: CentreType;
      AtOffice:   array [OffRng] of CustRng;
      MoneyRecd:  Database;
      CentreReq:  CustRng;
      CentreResp: Data;

      CustAck:    array [CustRng] of boolean;
```

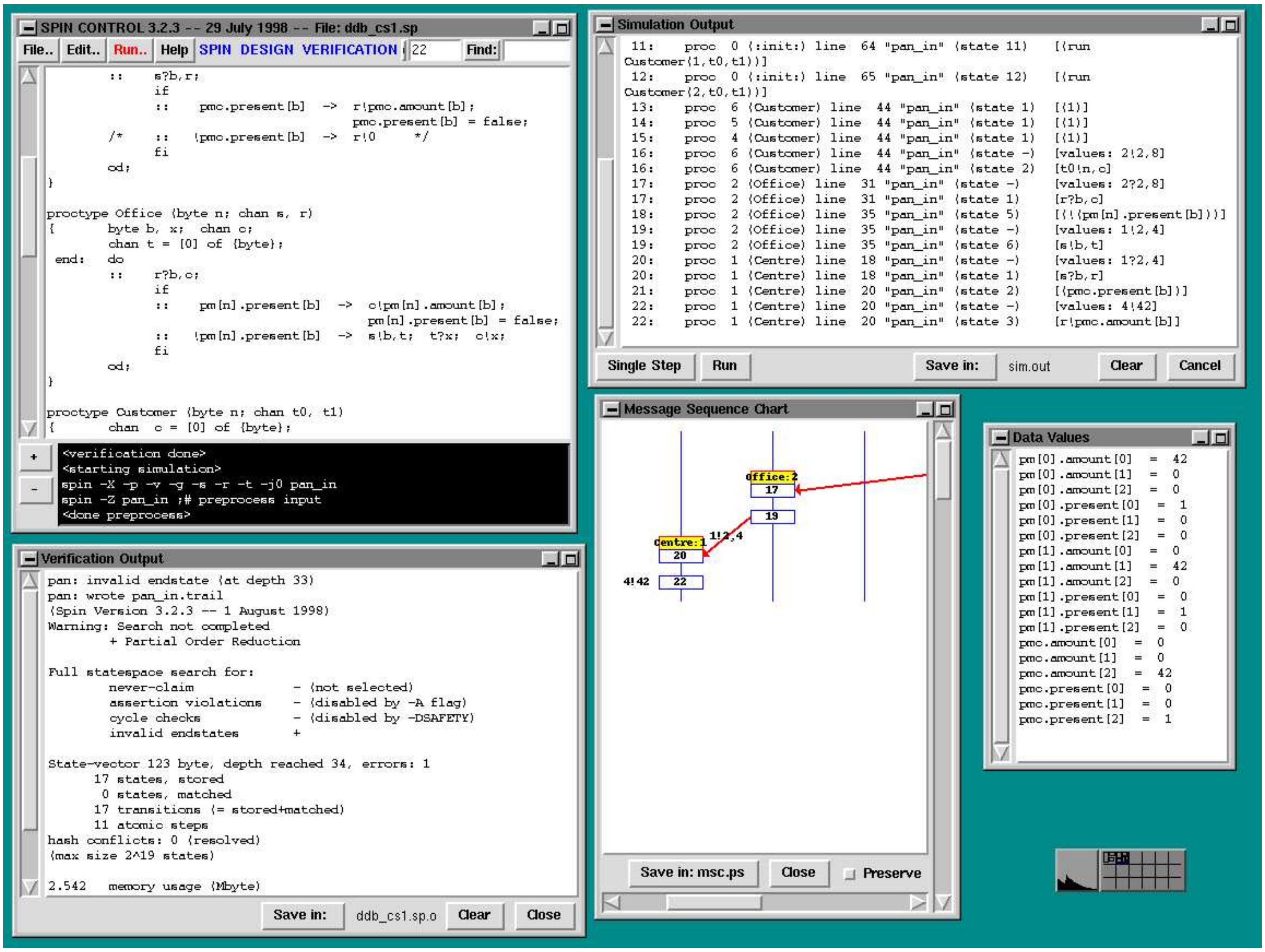

Figure 11: The SPIN simulation

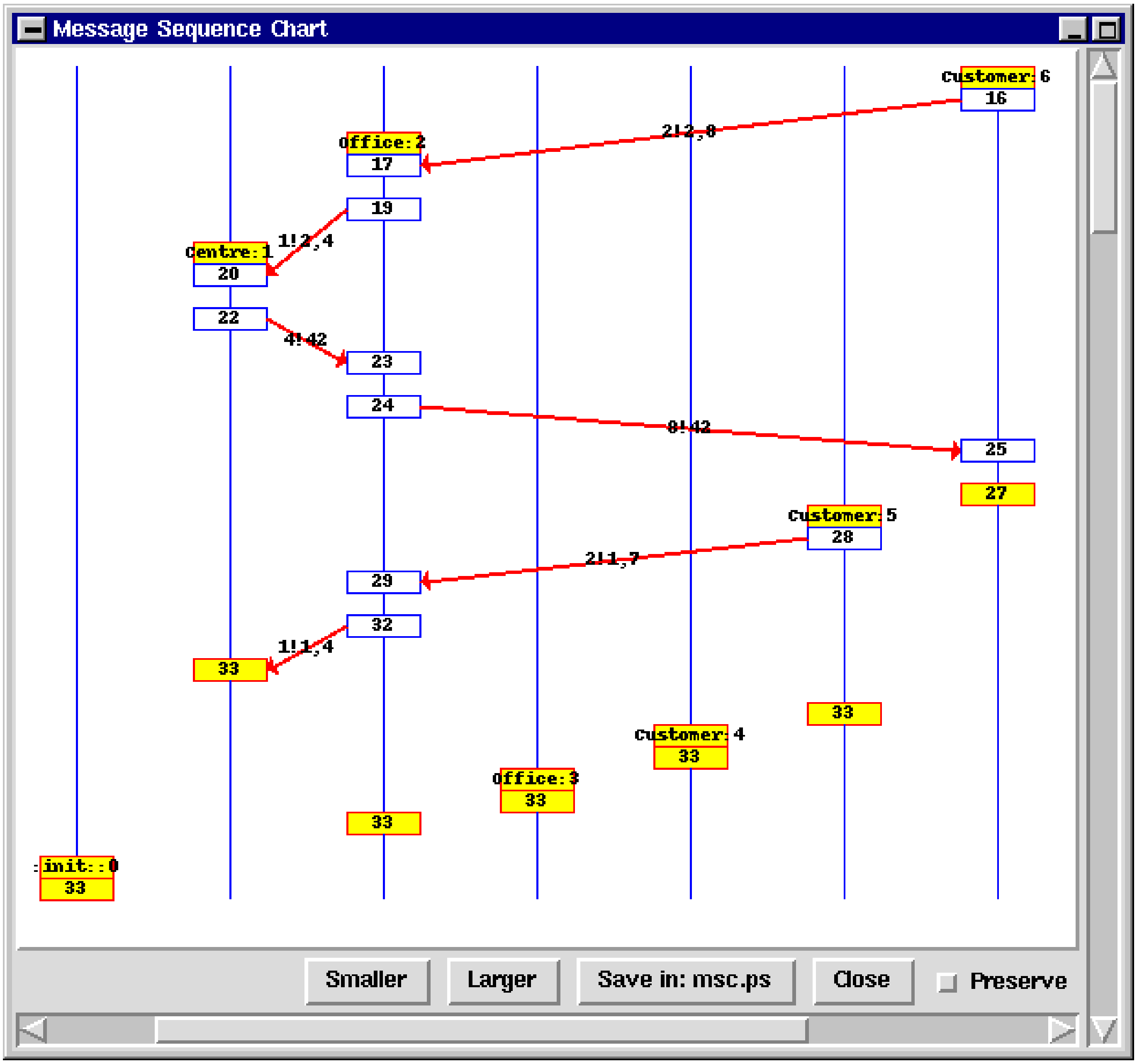

Figure 12: The SPIN message sequence chart

```
    CentreAck:  boolean;

    dummy:      boolean;
```

G.1 The Centre

The centre is modelled (below) by a pair of rules and a two-valued state variable: `CentreStat` is either `Ready` (to accept a query from an office) or `Busy`. If it is `Ready` and there is a request pending, the rule `"Accept query"` is enabled; the corresponding action acknowledges the request, changes the status to `Busy` and responds appropriately. If the status is `Busy`, the centre must wait until the whole transaction has been acknowledged by the office which requested it; it the centre then return to `Ready` status. (If the else part of the conditional statement in the first rule is commented out, corresponding to the missing functionality error in the original specification, deadlock can occur, and this is detected and reported by the Murϕ verifier.)

```
rule "Accept query"
    CentreStat = Ready &
    CentreReq != NoOne
==>
    CentreAck  := true;
    CentreStat := Busy;
    if pmc[CentreReq].present then
        CentreResp             := pmc[CentreReq];
        pmc[CentreReq].present := false;
    else
        CentreResp.amount  := 0;
        CentreResp.present := true;
    end;
end;

rule "Wait for Office Acks"
    CentreStat = Busy &
    !CentreResp.present &
    !CentreAck
==>
    CentreReq  := NoOne;
    CentreStat := Ready;
end;
```

G.2 The Offices

Each office is modelled (below) by a set of four rules and a four-valued state variable: `status` is either `Free`, `Serving` (engaged in a transaction with a customer), `Querying` (the centre), or `Awaiting` (a reply from the centre). The `ruleset` construction replicates the set of rules for each possible value of the parameter `off` (in this case 1 or 2).

```
ruleset off: OffRng do
  alias c: AtOffice[off];  status: OffStat[off] do

    rule "Accept Customer"
        status = Free &
        c != NoOne
    ==>
        if pm[off][c].present then
            status             := Serving;
            MoneyRecd[c]       := pm[off][c];
            pm[off][c].present := false;
        else
            status             := Querying;
        end;
    end;

    rule "Await customer ack"
        status = Serving &
        CustAck[c]
    ==>
        CustAck[c]    := false;
        AtOffice[off] := NoOne;
        status        := Free;
    end;

    rule "Contact Centre"
        status = Querying &
        CentreReq = NoOne
    ==>
        CentreReq := c;
        status    := Awaiting;
    end;

    rule "Await centre's reply"
        status = Awaiting &
        CentreAck &
        CentreResp.present
    ==>
        CentreAck          := false;
        MoneyRecd[c]       := CentreResp;
        CentreResp.present := false;
        CentreResp.amount  := 0;
        status             := Serving;
    end;

  end;
end;
```

G.3 The Customers

Each customer is modelled by a pair of rules and a three-valued state variable: `CustStat[c]` is either `Pennyless` (the initial value), `Waiting` (engaged in a transaction with an office) or `Done` (having completed a transaction). Once a customer's status becomes `Done` it can no longer engage in any transactions (i.e. it effectively terminates).

```
ruleset c: RealCustRng do

  ruleset off: OffRng do
    rule "Go to an office"
        CustStat[c]   = Pennyless &
        AtOffice[off] = NoOne
    ==>
```

```
        AtOffice[off] := c;
        CustStat[c]   := Waiting;
      end;
  end;

  rule "Complete transaction"
      CustStat[c] = Waiting &
      MoneyRecd[c].present
  ==>
      CustAck[c]  := true;
      CustStat[c] := Done;
  end;

end;
```

G.4 Detection of valid end states and avoiding deadlocks

Murϕ assumes that system execution is infinite; it does not distinguish between valid and invalid end states. It therefore always reports an error in a client/server model where all the clients have terminated, and is unable to differentiate this case from a genuine deadlock. To circumvent this problem, we introduce an additional rule which detects the valid end state, and avoids deadlock by repeatedly changing the value of a dummy variable. (Murϕ's definition of deadlock is that the current state has no successor other than itself.) This allows genuine deadlocks to be detected and reported correctly. We also use this rule to introduce an assertion regarding a property of the system - customer 3, whose details are held at the centre, will successfully complete his transaction regardless of which office he goes to.

```
rule "Reached valid end state"
  forall c: RealCustRng do
      CustStat[c] = Done & !CustAck[c]
  end  &
  forall off: OffRng do
      OffStat[off] = Free & AtOffice[off] = NoOne
  end  &
  CentreStat = Ready & CentreReq = NoOne &
  !CentreAck & !CentreResp.present
==>
  assert
    MoneyRecd[3].amount = 42
    "Cust 3 completed transaction successfully";

  dummy := !dummy;  --  Avoid deadlock
end;
```

G.5 The initial state of the system

The startstate describes the initial state of the system. All the variables are initialised by the `clear` command to the lowest values of their type, and the databases are set up so that office 1 knows about customer 1, office 2 knows about customer 2, and the centre knows about customer 3.

```
startstate

  clear pmc;  clear pm;
  pmc[3].present   := true; pmc[3].amount   := 42;
  pm[1][1].present := true; pm[1][1].amount := 42;
  pm[2][2].present := true; pm[2][2].amount := 42;

  clear CustStat;
  clear OffStat;
  clear CentreStat;
  clear AtOffice;
  clear MoneyRecd;
  clear CentreReq;
  clear CentreResp;
  clear CustAck;
  clear CentreAck;
  clear dummy;
end
```

Running the Murϕ verifier on the above model shows that deadlock does not occur, the valid end state is reached, and the assertion is never violated. If we introduce the missing centre functionality error as described above, an error trace is produced, allowing the user to determine (albeit with some difficulty) where the error occurred. The trace consists of the sequence of rules that were fired to reach the state in which the error was detected. For each rule, the values of the state variables that changed as a consequence of executing that rule are printed, except for the initial and final states, for which the values of all the state variables are printed.

Below is report produced for the erroneous version of the model, abbreviated slightly by omitting some uninteresting variables from the first and final states. Careful analysis of the values in the final state reveals that office 2 is waiting for a response from the centre (`CentreResp.present`), while the centre is waiting for office 2 to acknowledge its (non-existent) response (`!CentreAck`). By working backwards through the trace the sequence of events which led to this situation can be determined, and hence the nature of the error can be deduced.

```
The following is the error trace for the error:

Deadlocked state found.

Startstate Startstate 0 fired.
pmc[0].present:false
pmc[0].amount:0
      .
      .     (20 lines omitted)
      .
pm[2][3].present:false
pm[2][3].amount:0
```

```
CustStat[1]:Pennyless
CustStat[2]:Pennyless
CustStat[3]:Pennyless
OffStat[1]:Free
OffStat[2]:Free
CentreStat:Ready
AtOffice[1]:0
AtOffice[2]:0
MoneyRecd[0].present:false
MoneyRecd[0].amount:0
MoneyRecd[1].present:false
MoneyRecd[1].amount:0
MoneyRecd[2].present:false
MoneyRecd[2].amount:0
MoneyRecd[3].present:false
MoneyRecd[3].amount:0
CentreReq:0
CentreResp.present:false
CentreResp.amount:0
CustAck[0]:false
CustAck[1]:false
CustAck[2]:false
CustAck[3]:false
CentreAck:false
dummy:false
----------

Rule Go to an office, c:2, off:1 fired.
CustStat[2]:Waiting
AtOffice[1]:2
----------

Rule Go to an office, c:1, off:2 fired.
CustStat[1]:Waiting
AtOffice[2]:1
----------

Rule Accept Customer, off:1 fired.
OffStat[1]:Querying
----------

Rule Contact Centre, off:1 fired.
OffStat[1]:Awaiting
CentreReq:2
----------

Rule Accept Customer, off:2 fired.
OffStat[2]:Querying
----------

Rule Accept query fired.
The last state of the trace (in full) is:
pmc[0].present:false
pmc[0].amount:0
      .
      .    (20 lines omitted)
      .
pm[2][3].present:false
pm[2][3].amount:0
CustStat[1]:Waiting
CustStat[2]:Waiting
CustStat[3]:Pennyless
OffStat[1]:Awaiting
OffStat[2]:Querying
CentreStat:Busy
AtOffice[1]:2
AtOffice[2]:1
MoneyRecd[0].present:false
MoneyRecd[0].amount:0
MoneyRecd[1].present:false
MoneyRecd[1].amount:0
MoneyRecd[2].present:false
MoneyRecd[2].amount:0
MoneyRecd[3].present:false
MoneyRecd[3].amount:0
CentreReq:2
CentreResp.present:false
CentreResp.amount:0
CustAck[0]:false
CustAck[1]:false
CustAck[2]:false
CustAck[3]:false
CentreAck:true
dummy:false
----------

End of the error trace.

====================================================
```

Although this method of debugging is perfectly feasible, it is difficult and time-consuming, especially in more complex cases. The fundamental problem in this case is that the distributed database/client-server problem does not map well onto the modelling paradigm provided by Murϕ. In contrast to Spin, the Murϕ model is far removed from the user's mental model of the problem; every sort of structure in the mental model has to be unravelled in order to express it in Murϕ, and reported errors have first to be tracked down in terms of the model, then related back to the original problem. Although Murϕ has been used successfully in debugging communications protocols (the application for which it was designed), it is less suitable than Spin for modelling at a higher level of abstraction.

H The model using B

For the B specification, it was assumed that customers collect *tokens* from an office. The B specification consists of two parts, a state-oriented part and an event-oriented part. The state part is shown in Figure 13. This describes a B machine with a single state variable *tokens* that maps customers to a number of tokens available to that customer. The machine contains operations for adding and removing customers, for allocating extra tokens to a customer and an operation allowing a customer to collect

```
MACHINE  Tokens

VARIABLES  tokens

INVARIANT  tokens : CUST +-> NAT

INITIALISATION  tokens := {}

OPERATIONS

AddCust(cc) = tokens := tokens \/ { cc |-> 0 } ;

RemCust(cc) = tokens := {cc} <<| tokens ;

AllocToken(cc) = tokens(cc) := tokens(cc)+1 ;

toks <-- CollToken(pp,cc) =
    IF tokens(cc)=0
    THEN toks := 0
    ELSE
        ANY nn WHERE  nn : (1..tokens(cc))
        THEN  toks:=nn ||
              tokens(cc) := tokens(cc) - nn
        END
    END
```

Figure 13: State-oriented specification

tokens. Note that the *CollToken* operation returns a non-deterministic number of tokens. This is an abstraction of the policy used in the refinement whereby, if there are tokens available for a customer at the office, then only these are given to the customer and other tokens that may be at the centre or the home office are ignored.

The event-oriented part of the specification is shown in Figure 14. This describes order constraints on the operations written in a CSP-like notation [12]. The first process, *Customers*, constrains operation execution from a customer point of view saying that before a customer is active, it must be added using the *AddCust* operation. The second process, *Servers*, constrains operations from an office point of view saying that before collecting tokens, they must have been requested. Note that the notation of Figure 14 is not standard B, but it can be translated automatically to standard B using a custom-built tool [5]. In this CSP-like notation, `Ev -> P` represents event prefixing, `P [] Q` represents external choice between `P` and `Q`, while `||| x. P[x]` represents the interleaving of many instances of process `P` parameterised by `x`, where `x` ranges over some set (in the case of Figure 14, `InitCust` is parameterised by `cc` which ranges over `CUST`). Any operation mentioned in the event-oriented part but not in the state-oriented part is implicitly a `skip` operation.

The refinement of the system is also presented in two parts. In the state-oriented part in Figure 15, the abstract

```
PROCESS  Customers = ||| cc . InitCust[cc]  WHERE

InitCust[cc] = AddCust.cc -> ActiveCust[cc]

ActiveCust[cc] = RemCust.cc ->  STOP
                 [] AllocToken.cc -> ActiveCust[cc]
                 [] ReqToken.cc ->   ActiveCust[cc]
                 [] CollToken.cc ->  ActiveCust[cc]

PROCESS  Servers = ||| pp . Server[pp]  WHERE

Server[pp] = ReqToken.pp?cc -> CollToken.pp.cc ->
                 Server[pp]
```

Figure 14: Event-oriented specification

variable *tokens* is replaced by two variables *ctokens* and *ltokens* representing respectively the tokens stored at the centre and at a local office. This partitioning of the tokens models the distributed nature of the implementation. The machine also contains a variable mapping customers to their home office. As well as concrete versions of the operations of Figure 13, the refined machine also contains extra operations for distributing tokens from the centre (*DistToken* and *ReplyFromCentre*) and for transferring tokens from a local office to the centre.

Execution of these extra operations are constrained by the event-oriented description of Figure 16. This describes the protocol followed when a customer requests a token. After a request has been made, the way it is dealt with depends on whether there are tokens available for the customer locally. If the centre has to be queried, it will either respond immediately or will query the home office of the customer.

The refinement is verified using a invariant relating the abstract with the concrete state as follows:

```
! c. (c:CUST =>
 tokens(c) =
  ctokens(c) +
  SIGMA(p).(p|->c : dom(ltokens) |
             ltokens(p|->c))
)
```

Verification of the refinement uses the standard rules of B to show that each operation of the abstract system is refined by its counterpart in the concrete system under the invariant and that each extra operation introduced in the refinement is a refinement of `skip` under the invariant.

I The model using Z

After some preliminaries, we present three models. The first is a simple model of the action of the whole system.

```
REFINEMENT  TokensRef  REFINES Tokens

VARIABLES  home, ctokens, ltokens

INVARIANT
    home : CUST --> OFFICE        &
    ctokens : CUST +-> NAT        &
    ltokens : (OFFICE * CUST) +-> NAT

INITIALISATION
    ctokens := {}  ||  ltokens := {}

OPERATIONS

AddCust(cc) = ctokens := ctokens \/ { cc |-> 0 } ;

RemCust(cc) = ctokens := {cc} <<| ctokens ;

AllocToken(cc) = ctokens(cc) := ctokens(cc) + 1 ;

DistToken(pp, cc) =
    SELECT
        (cc |-> pp) : home  &  ctokens(cc) > 0
    THEN
        Inc(ltokens, (pp |-> cc), ctokens(cc))  ||
        ctokens(cc) := 0
    END ;

toks <-- CollToken(pp, cc) =
    toks := ltokens(pp |-> cc)  ||
    ltokens(pp |-> cc) := 0  ;

ReplyFromCentre(pp,cc) =
    Inc(ltokens, (pp |-> cc), ctokens(cc))  ||
    ctokens(cc) := 0  ;

ReplyToCentre(pp,cc) =
    IF (home(cc) |-> cc):dom(ltokens)
    THEN ctokens(cc) := ctokens(cc) +
            ltokens(home(cc) |-> cc)  ||
         ltokens(home(cc) |-> cc) := 0
    END

DEFINITIONS

Inc(f,x,y) == IF x:dom(f) THEN f(x):=f(x)+y
                          ELSE f(x):=y
```

Figure 15: State-oriented refinement

```
PROCESS  Office = ||| pp . AwaitCust[pp]  WHERE

AwaitCust[pp] = ReqToken.pp?cc -> DealCust[pp](cc)

DealCust[pp](cust:CUST)  =
    IF  ((pp|->cust):dom(ltokens))  &
        (ltokens(pp |-> cust) > 0)
    THEN  CollToken.pp.cust -> AwaitCust[pp]
    ELSE  QueryCentre.pp.cust -> Centre[pp](cust)

Centre[pp](cust)  =
    IF  (ctokens(cust) > 0) or (pp=home(cust))
    THEN  ReplyFromCentre.pp.cust -> Collect[pp](cust)
    ELSE  QueryHome.pp.cust -> ReplyToCentre.pp.cust ->
              ReplyFromCentre.pp.cust -> Collect[pp](cust)

Collect[pp](cust) =
    CollToken.pp.cust ->  AwaitCust[pp]
```

Figure 16: Event-oriented refinement

No individual offices are identified, and no communication is included. The second moves to a model where the offices are modelled, to the extent of having a number of separate databases. A third model describes full distribution, complete with a message-passing scheme.

I.1 Preliminaries

Z provides a rich mathematical toolkit. For our present purposes, a number of domain-specific items must be identified. Firstly, we need a set to describe the service, which for the purpose of this specification is assumed to represent a certain amount of money to be paid out to the customer. There are no debts, so only positive amounts of money are needed.

$$Money == \mathbb{N}$$

Secondly, there is a set of people.

$$[Person]$$

For ease of discussion, we name a number of people.

$$p_1, p_2, p_3 : Person$$
$$\text{disjoint}\langle\{p_1\}, \{p_2\}, \{p_3\}\rangle$$

Finally, we will also need a set of offices.

$$[Office]$$

and we will identify a couple of distinct offices.

$$o_1, o_2 : Office$$
$$o_1 \neq o_2$$

I.2 A simple model

The simplest model ignores all the distributed aspects. It is, in some sense, the view the customer (or counter clerk) has of the system.

The system knows about how much money is due to a certain collection of people. This is modelled as a partial function. In greater generality, it might be a relation, because if someone didn't claim their money at some stage, it might be held over for a time.

$$
\begin{array}{|l}
\textit{WholePO} \\
\hline
pm : Person \mathrel{+\!\!\!\!\rightarrow} Money \\
\hline
\end{array}
$$

Initially, for our purposes, the system knows about three people, and how much money each is due.

$$
\begin{array}{|l}
\textit{InitPO} \\
\hline
\textit{WholePO}' \\
\hline
pm' = \{p_1 \mapsto 45, p_2 \mapsto 45, p_3 \mapsto 45\} \\
\hline
\end{array}
$$

The only operation is when a customer asks for some money. The precondition says that the customer must be due some.

$$
\begin{array}{|l}
\textit{MoneyReq} \\
\hline
\Delta \textit{WholePO} \\
customer? : Person \\
cash! : Money \\
\hline
customer? \in \operatorname{dom} pm \\
pm' = \{customer?\} \mathbin{\lhd\!\!\!\!-} pm \\
cash! = pm\ customer? \\
\hline
\end{array}
$$

We may animate this specification using the PiZA tool. The script first runs the initialisation schema (`cs` = call schema), and then a sequence of money requests.

```
specification_file( "zmodel" ).
begin.
set reporting_level = 3;
{cs InitPO}
{cs MoneyReq customer?  = p_1 }
{cs MoneyReq customer?  = p_3 }
{cs MoneyReq customer?  = p_2 }

...
```

The following output comes from executing the script. We see that customers p_1 and p_3 are given their money, but a second request from p_1 fails.

```
% creating schema calls
1 ================================================
Results of calling schema InitPO
pm' = { ( p_1, 45 ), ( p_2, 45 ), ( p_3, 45 ) }
==================================================
2 ================================================
Results of calling schema MoneyReq
cash!  = 45
customer?  = p_1
pm = { ( p_1, 45 ), ( p_2, 45 ), ( p_3, 45 ) }
pm' = { ( p_2, 45 ), ( p_3, 45 ) }
==================================================
3 ================================================
Results of calling schema MoneyReq
cash!  = 45
customer?  = p_3
pm = { ( p_2, 45 ), ( p_3, 45 ) }
pm' = { ( p_2, 45 ) }
==================================================
Schema call to MoneyReq failed.
```

Failure in the animation is not acceptable—it may correspond to a system crash in the implementation. So far, the specification says nothing about how to behave if the precondition fails to hold. We add a specification of the failure case.

$$\textit{Report} ::= \textit{Ok} \mid \textit{NoAuthority}$$

$$
\begin{array}{|l}
\textit{TransOk} \\
\hline
report! : Report \\
\hline
report! = \textit{Ok} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\textit{NoMoney} \\
\hline
\Xi \textit{WholePO} \\
customer? : Person \\
cash! : Money \\
report! : Report \\
\hline
customer? \notin \operatorname{dom} pm \\
cash! = 0 \\
report! = \textit{NoAuthority} \\
\hline
\end{array}
$$

$$\textit{Request} == (\textit{MoneyReq} \land \textit{TransOk}) \lor \textit{NoMoney}$$

Request is a total operation. Specifying the failure case in this way is important for a robust specification, but it will be omitted in the sequel, because it is typically uninteresting.

The animation of *Request* follows a different course. At the second request, the customer p_1 receives no cash, and is told why.

```
6 =================================================
Results of calling schema Request
cash!  = 45
customer?  = p_1
pm = { ( p_1, 45 ), ( p_2, 45 ), ( p_3, 45 ) }
pm' = { ( p_2, 45 ), ( p_3, 45 ) }
report!  = Ok
=================================================
7 =================================================
Results of calling schema Request

...
=================================================
8 =================================================
Results of calling schema Request
cash!  = 0
customer?  = p_1
pm = { ( p_2, 45 ) }
pm' = { ( p_2, 45 ) }
report!  = NoAuthority
=================================================
```

I.3 A model with some office structure

A model which admits that the database is in fact distributed might be described as follows. pmc is the database held at the centre; for an office o, the database for that office is $pmo\ o$.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{StructuredPO}} \\
pmc : Person \pfun Money \\
pmo : Office \rightarrowtail\!\!\!\!\!\!\rightarrow (Person \pfun Money) \\
\hline
\forall p : \mathrm{dom}\, pmc \bullet p \notin \mathrm{dom}(\bigcup(\mathrm{ran}\, pmo)) \\
\forall pm_1, pm_2 : \mathrm{ran}\, pmo \mid pm_1 \neq pm_2 \bullet \\
\quad (\forall p : \mathrm{dom}\, pm_1 \bullet p \notin \mathrm{dom}\, pm_2)
\end{array}
$$

The predicates (and the fact that pmo is an injection) ensure that no person is authorised in more than one place at any instant.

In our initial state, some data is at the centre, other data is already at the offices.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{InitStructuredPO}} \\
StructuredPO' \\
\hline
pmc' = \{p_3 \mapsto 45\} \\
pmo' = \{o_1 \mapsto \{p_1 \mapsto 45\}, o_2 \mapsto \{p_2 \mapsto 45\}\}
\end{array}
$$

When a customer requests some money, the money is paid out provided the customer is in either the central database, or in the database of the office to which the request is made. The customer is removed from the data base of the office.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{StructMoneyReq}} \\
\Delta StructuredPO \\
customer? : Person \\
office? : Office \\
cash! : Money \\
\hline
(customer? \in \mathrm{dom}\, pmc \\
\quad \wedge pmc' = \{customer?\} \ndres pmc \\
\quad \wedge cash! = pmc\ customer?) \\
\vee \\
(customer? \in \mathrm{dom}(pmo\ office?) \\
\quad \wedge pmo' = pmo \oplus \\
\qquad \{office? \mapsto (\{customer?\} \ndres pmo\ office?)\} \\
\quad \wedge cash! = pmo\ office?\ customer?)
\end{array}
$$

Again, we could improve the specification by covering error cases. This model is informally a refinement of the previous model, but having omitted offices from the abstract model, it is not possible (using the normal Z notion of refinement) to produce a proof of this. To prove that the model is a refinement of the previous, we would have to hide the offices in the current model.

I.4 Distributed model

There are several items of state in our distributed system. The centre has a database, as before.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{CentreState}} \\
pm : Person \pfun Money
\end{array}
$$

The individual offices also have a database. They also have, in this model, a notion of their own identity. For any given office, this is constant, so we give a special definition of $\Delta OfficeState$.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{OfficeState}} \\
id : Office \\
pm : Person \pfun Money
\end{array}
$$

$$\Delta OfficeState == [OfficeState;\ OfficeState' \mid id = id']$$

Our distributed components will communicate using messages. We describe a special datatype for messages.

$$
\begin{array}{ll}
Msg ::= & Pers\langle\!\langle Person \times Office \rangle\!\rangle \\
& \mid Req\langle\!\langle Person \rangle\!\rangle \\
& \mid Amount\langle\!\langle Money \rangle\!\rangle \\
& \mid Authorisation\langle\!\langle Person \times Money \rangle\!\rangle
\end{array}
$$

Messages have a destination, and this might be a person, one of the offices, or the centre.

$$Id ::= P\langle\!\langle Person \rangle\!\rangle \mid O\langle\!\langle Office \rangle\!\rangle \mid C$$

Our model for the sending of messages is to launch them into the ether, marked for a recipient. At any one time there may be several messages in the ether for the same recipient. We assume nothing about order of delivery.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Ether}} \\
\hline
\mathit{messages} : \mathit{Id} \leftrightarrow \mathit{Msg} \\
\hline
\end{array}$$

Whilst this model describes a broadcast system, there is nothing to stop it being implemented using a more careful delivery technique.

If an office receives a request for cash from an individual, it will either deliver cash (a cash message!) or ask the centre for authorisation.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{OfficeRequestLocal}} \\
\hline
\Delta \mathit{OfficeState} \\
\Delta \mathit{Ether} \\
\hline
\exists\, p : \mathit{Person} \bullet \mathbf{let}\ \mathit{msg} == O\ \mathit{id} \mapsto \mathit{Req}\ p \bullet \\
\quad \mathit{msg} \in \mathit{messages} \land \\
\quad p \in \operatorname{dom} \mathit{pm} \land \\
\quad \mathit{messages}' = \mathit{messages} \setminus \{\mathit{msg}\} \\
\qquad \cup \{P\ p \mapsto \mathit{Amount}\ (\mathit{pm}\ p)\} \land \\
\quad \mathit{pm}' = \{p\} \ntriangleleft \mathit{pm} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{OfficeRequestRemote}} \\
\hline
\Xi \mathit{OfficeState} \\
\Delta \mathit{Ether} \\
\hline
\exists\, p : \mathit{Person} \bullet \mathbf{let}\ \mathit{msg} == O\ \mathit{id} \mapsto \mathit{Req}\ p \bullet \\
\quad \mathit{msg} \in \mathit{messages} \land \\
\quad p \notin \operatorname{dom} \mathit{pm} \land \\
\quad \mathit{messages}' = \mathit{messages} \setminus \{\mathit{msg}\} \\
\qquad \cup \{C \mapsto \mathit{Pers}\ (p, \mathit{id})\} \\
\hline
\end{array}$$

Upon receiving an authorisation from the centre, the office will forward cash to the requesting individual.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{OfficeResponse}} \\
\hline
\Xi \mathit{OfficeState} \\
\Delta \mathit{Ether} \\
\hline
\exists\, p : \mathit{Person};\ m : \mathit{Money} \bullet \\
\quad \mathbf{let}\ \mathit{msg} == O\ \mathit{id} \mapsto \mathit{Authorisation}(p, m) \bullet \\
\qquad \mathit{msg} \in \mathit{messages} \land \\
\qquad \mathit{messages}' = \mathit{messages} \setminus \\
\qquad\quad \{\mathit{msg}\} \cup \\
\qquad\quad \{P\ p \mapsto \mathit{Amount}\ m\} \\
\hline
\end{array}$$

When a request is received at the centre, it returns an authorisation if the person is listed as being due a payment.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{CentreOp}} \\
\hline
\Delta \mathit{CentreState} \\
\Delta \mathit{Ether} \\
\hline
\exists\, p : \mathit{Person};\ o : \mathit{Office} \bullet \\
\quad \mathbf{let}\ \mathit{msg} == C \mapsto \mathit{Pers}(p, o) \bullet \\
\qquad \mathit{msg} \in \mathit{messages} \land \\
\qquad p \in \operatorname{dom} \mathit{pm} \land \\
\qquad \mathit{pm}' = \{p\} \ntriangleleft \mathit{pm} \land \\
\qquad \mathit{messages}' = \mathit{messages} \setminus \{\mathit{msg}\} \\
\qquad\quad \cup \{O\ o \mapsto \mathit{Authorisation}(p, \mathit{pm}\ p)\} \\
\hline
\end{array}$$

Of course, we need to deal with the error cases again. These operation schemas have much in common, and they are a little different from the usual Z style. Some derived operators are probably needed.

In order to set up a refinement relation between this specification and the previous one, we must create a means whereby the operations may have the same interface as the corresponding operations in the previous model. These 'front office' operations use the input/output parameters to construct messages.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{FrontOfficeReq}} \\
\hline
\Delta \mathit{Ether} \\
\mathit{customer}? : \mathit{Person} \\
\mathit{office}? : \mathit{Office} \\
\hline
\mathit{messages}' = \mathit{messages} \cup \\
\quad \{O\ \mathit{office}? \mapsto \mathit{Req}\ \mathit{customer}?\} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{FrontOfficeResp}} \\
\hline
\Delta \mathit{Ether} \\
\mathit{cash}! : \mathit{Money} \\
\mathit{customer}? : \mathit{Person} \\
\hline
\mathbf{let}\ \mathit{msg} == P\ \mathit{customer}? \mapsto \mathit{Amount}\ \mathit{cash}! \bullet \\
\quad \mathit{msg} \in \mathit{messages} \land \\
\quad \mathit{messages}' = \mathit{messages} \setminus \{\mathit{msg}\} \\
\hline
\end{array}$$

The whole operation then is

$$\begin{array}{l}
\mathit{OfficeOp} == \mathit{FrontOfficeReq} \fatsemi \\
\quad (\mathit{OfficeRequestLocal} \lor \\
\qquad \mathit{OfficeRequestRemote} \fatsemi \\
\qquad \mathit{CentreOp} \fatsemi \mathit{OfficeResponse}) \fatsemi \\
\quad \mathit{FrontOfficeResp}
\end{array}$$

So far, we have described separately the individual offices, the centre, and the ether by which they communicate. To complete the description, we must describe the whole state of the system.

$$
\begin{array}{|l}
\hline
\textit{WholeState} \\
\textit{CentreState} \\
\textit{Ether} \\
\textit{Offices} : \textit{Office} \nrightarrow \textit{OfficeState} \\
\hline
\forall o : \mathrm{dom}\ \textit{Offices} \bullet o = (\textit{Offices}\ o).id \\
\hline
\end{array}
$$

The *OfficeOp* operation involves a particular office, the centre, and the ether. In order to make it an operation on the *WholeState*, we use the Z technique called *promotion*. The whole system involves a number of offices; the office participating in the *OfficeOp* is identified with the correct one.

$$
\begin{array}{|l}
\hline
\textit{PromoteToWholeState} \\
\Delta\, \textit{WholeState} \\
\Delta\, \textit{OfficeState} \\
\textit{office}? : \textit{Office} \\
\hline
\textit{office}? \in \mathrm{dom}\ \textit{Offices} \\
\theta\, \textit{OfficeState} = \textit{Offices}\ \textit{office}? \\
\theta\, \textit{OfficeState} = \textit{Offices}'\ \textit{office}? \\
\{\textit{office}?\} \lhd \textit{Offices} = \{\textit{office}?\} \lhd \textit{Offices}' \\
\hline
\end{array}
$$

$$
\textit{OfficeOpPromoted} == (\exists\, \textit{OfficeState};\ \textit{OfficeState}' \bullet \\ \textit{PromoteToWholeState} \land \textit{OfficeOp})
$$

OfficeOpPromoted is intended to be a refinement of *StructMoneyReq* in the previous model. The signature is the same, and the expected behaviour is the same—though much additional behind-the-scenes communication has now been specified. A formal proof of the refinement is outside the scope of this exercise.

I.5 Discussion

The Z models increase in complexity. At each level there is the opportunity for clear expression of *what* is to happen, without necessarily describing *how*—though more and more information is added as we descend towards a detailed design. Z provides the opportunity to prove that the design steps are valid, but we have not exploited that ability. We have, however, used available tools to validate the specification by animation.

J The QNAP2 and SWAP tool models

In this appendix, the abstract and concrete QNAP2 models are described as well as the SWAP tool model. Values of the model parameters (service rates, routing etc.) are assumed to be reasonable; they are not based on expert knowledge of the application domain.

J.1 Abstract QNAP2 model

The QNAP2 specification of the abstract model represents the centre and its customers as follows:

```
/DECLARE/ QUEUE CENTRE, OFFICE;
          INTEGER NUMCUSTS;

/STATION/ NAME = CENTRE;
          SERVICE = CST(5.0);
          TRANSIT = OFFICE;

/STATION/ NAME = OFFICE;
          SERVICE = ERLANG(5.0, 2);
          TRANSIT = CENTRE, 0.2,
                    OFFICE;
          INIT = NUMCUSTS;

/CONTROL/ OPTION = NRESULT;
/EXEC/    BEGIN
          FOR NUMCUSTS := 100, 200, 300 DO
              BEGIN
                  SOLVE;
                  PRINT("Number of customers = ",
                        NUMCUSTS,
                        "Office response time = ",
                        MRESPONSE(OFFICE));
              END;
          END;
/END/
```

Customers repeatedly enter the office to obtain a service. It his assumed that 80% of the jobs are performed by the office. For the remaining 20%, the job requires the service of the centre. This information is specified by the `TRANSIT` statement. The model's notion of service by the centre represents the actual download of a customer's data from the centre to the office.

Both the centre and the office contain a queue and a service station. In the centre, each request is assumed to take exactly 5.0 seconds (`CST(5.0)`) to be processed and all serviced jobs flow back to the office. In the office, service time is given by a random number. The Gamma distribution, a.k.a. the Erlang distribution, is commonly used to model the time taken to complete a task of the form 'customer service'. The parameters to the `ERLANG` function are its mean and an order.

We assume that there are a fixed number of customers in the abstract model as a whole and that they all start by queueing for service at the office. The model is solved (exactly) for 100, 200 and 300 customers. The mean mean response time at the office is chosen as a suitable measure of performance because it represents an aggregated quality of service as seen by the customers. Results show that the mean response time for 100, 200 and 300 customers is 498.9, 998.9 and 1499.0 seconds respectively. There appears to be a linear relationship between customer numbers and office response time. We conclude that the centre is not helping to increase system performance as perceived by the customers at the office.

J.2 Concrete QNAP2 model

Although it only has one office, the abstract model can represent a one-office viewpoint of a system that has a multitude of offices linked to a single centre. The major intention of concrete model is to represent system that has two types of office. The model is as follows:

```
/DECLARE/ INTEGER NUMOFFS = 2;
          QUEUE OFFICE(NUMOFFS), CENTRE, FORK;
          INTEGER NUMCUSTS = 600;
          CLASS LOWJOB, HIGHJOB;

/STATION/ NAME = CENTRE;
          SERVICE = CST(5.0);
          TRANSIT = FORK;

/STATION/ NAME = OFFICE(1);
          SERVICE = ERLANG(5.0, 2);
          TRANSIT = CENTRE, 0.2, FORK;

/STATION/ NAME = FORK;
          TYPE = INFINITE;
          SERVICE = EXP(5.0);
          TRANSIT = OFFICE(1), 0.5, OFFICE(2);
          INIT = NUMCUSTS;

/STATION/ NAME = OFFICE(2);
          TYPE = MULTIPLE(2);
          SERVICE = UNIFORM(1.0, 5.0);
          TRANSIT = CENTRE, 0.05, FORK;
          SCHED = FIFO, PRIOR;
          CAPACITY = 50;
          REJECT(LOWJOB, HIGHJOB)
              = BEGIN
                    TRANSIT(FORK);
                END;

/CONTROL/ TMAX = 1000000;
/EXEC/    SIMUL;
/END/
```

An array of queues is used for the offices and different classes of customer allow for priority scheduling in (one of) the stations. To model the choice a customer has about which (type of) office to attend, an explicit station representing a fork in the network is required. (The network is shown in figure 17.)

The centre and the first office of the concrete model are identical to the centre and the office of the abstract model except that the jobs start at the fork and once serviced jobs return to the fork.

The fork station is an infinite server station, which means that the customers are only delayed by a service

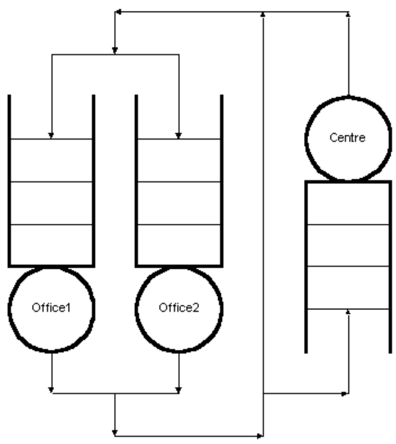

Figure 17: The concrete queueing network

time. The delay is a exponential random number. After service, customers are equally likely to choose either office.

In the second office, there are two processes serving the queue, service times are uniform random variables between 1.0 and 5.0 seconds and there is a 5% referral rate to the centre. Customers are ordered according to their class (`PRIOR`), and customers with equal priority are ordered according to the time of their arrival `FIFO`. The office can only hold 50 jobs. If a job arrives when the queue is full it is sent back to the fork so that the customer has the chance of opting for the first office again.

The concrete model simulates a period of 1000000 seconds (actual execution time is less than one minute). The results are shown in the table below, where the rows correspond with the stations `OFFICE 1`, `OFFICE 2`, `CENTRE`, and `FORK`. The columns are the average service time at a station, the percentage of time that the station is busy, the average number of customers queued at the station, the average response time and the total number of customers served by the station.

SERV	BUSY%	CUST#	RESP	SERV#
5.011	1.000	1197	5979	199575
3.002	0.299	0.638	3.199	199318
5.000	0.252	0.283	5.607	50389
5.001	0.000	2.001	5.001	401204

The results indicate that the first office is by far the bottleneck (station) for the model.

J.3 The SWAP model

SWAP is a tool that constructs a large model of a computer system from accurate models of hardware elements and software elements. It uses a sophisticated user interface that includes an image graph in which each image represents an element of the system. A screenshot of our database model in the SWAP tool is shown in figure 18.

The image graph is used to construct a large queueing network. A workload (i.e. jobs) for the network are generated from the tree structure that occurs in the SWAP interface. The workload models the system's programs and in a database system its elements are scripts, transactions and queries.

In SWAP, the 13 resource elements of our model are displayed in the image graph, which represents the system's configuration. An office workstation is connected through a WAN and a LAN to a server (machine) at the centre. The server has two disks on which tables of the system's database are distributed and a terminal. The DBMS runs on the server and logs database accesses on the second disk.

There are two scripts, one for the office's workstation `SOffice` and one for the centre's terminal `SOffice`. The workstation's script consists of a transaction to enter new customers into the database (`TNewCustomer`) and a transaction to record a (policy) payment to a customer (`TPayment`). For each execution of the script, there is a 10% probability of execution of the first transaction, which means that the probability of execution of the second transaction is 90%. These relative percentages are not shown in the figure. Transactions are expressed in SQL extended with explicit delays and so for example, the `TPayment` transaction used by the workstation is as follows:

```
select * from PAYMENT where CustomerID = 12345678;
think time;
insert into PAYMENT values
  ('25th Nov 1998', 42.00, 12345678);
commit;
```

(These 4 statements are represented by the children of the `TPayment` node that occurs in workload tree displayed in the figure.)

Simulation of the model will produce responses for each of the resource elements in the configuration. No explicit results are presented here because we are still in the process of completing the SWAP model.

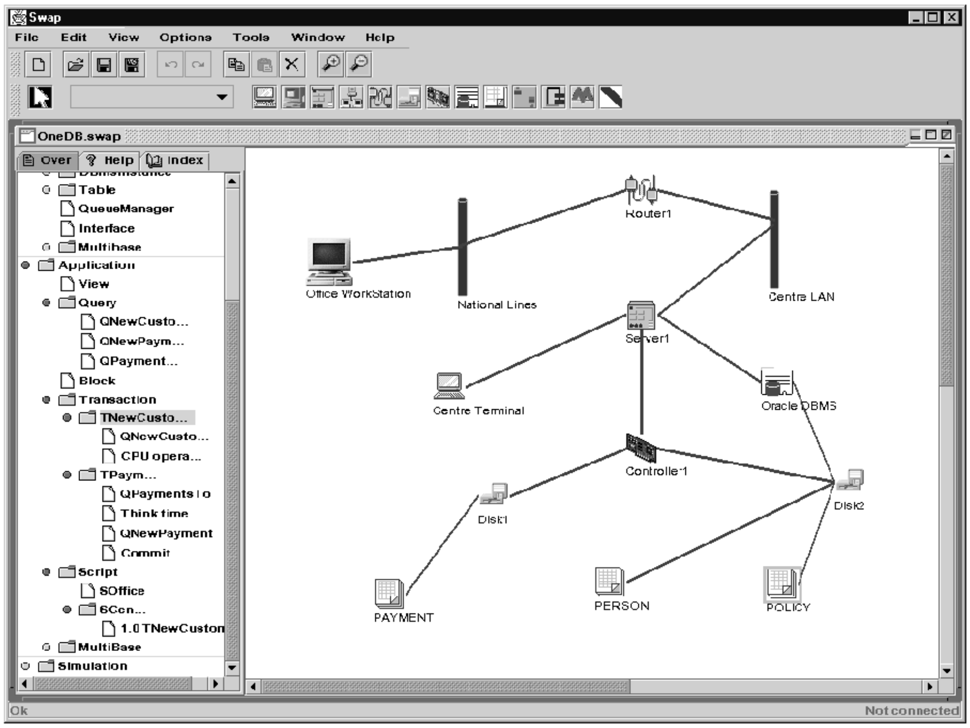

Figure 18: The SWAP model