FORMAL MODELS AND ANALYSIS FOR SOFTWARE COMPONENT INTERACTION TESTING

by

Wayne Liu

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2000

0-612-53503-7

ABSTRACT

An important problem in component-based software development is testing for correct interactions of components. A promising approach is to use formal state-space models and analysis techniques for effective and automated testing. However, existing techniques are not feasible as models and test cases are difficult enough for people to create, or alternatively, too complex for automatic tools to analyze.

To overcome these two problems, this thesis presents an object-oriented modelling language that is easy to use for software design, but it also has formal semantics to allows models to be re-used for interaction testing. Formal coverage criteria and test requirements can be defined for the models, and test cases can be generated using a model-checking approach. The thesis also presents new algorithms created to contain the state explosion that occurs in test generation from large models of software components. The algorithms allow the analysis of much larger models than previously possible. The feasibility of using these techniques for interaction testing is demonstrated by an experiment on the design of an example software system.

ACKNOWLEDGMENTS

I wish to thank my wife, Lily, for believing in me and supporting me, for all the hard work we put into this as a team. I wish to thank my supervisor, Prof. P. Dasiewicz, for his patience when I did not know what I was doing, as well as when I hurried to finish. I wish to thank all the members of my committee: Prof. J.M. Atlee, Prof. G.v. Bochman, Prof. R. Seviora, and Prof. A. Singh. Finally, I wish to acknowledge my Jesus Christ, my Saviour. "Trust in the Lord with all your heart and lean not on your own understanding; in all your ways acknowledge him, and he will make your paths straight."

TABLE OF CONTENTS

LIST OF TABLES

LIST OF ILLUSTRATIONS

Chapter 1

INTRODUCTION

Component-based software development is expected to be an effective and widely used method of creating software. However, a major problem is to ensure that components that were developed separately work properly together. Little research and very few techniques exist for testing component interactions, compared to testing individual components or whole systems. The effectiveness of the proposed techniques is largely unknown, and the techniques may involve significant manual effort.

Formal models of component interactions can simplify testing of component interactions by formally defining testing requirements for interactions, and automatically generating required test cases using model-checking technology. The approach allows more rigorous testing, since formal models and test selection criteria do not require individual interpretation by testers. Thus, the effectiveness of the test selection techniques can be evaluated objectively and various techniques compared. In addition, the techniques can be applied automatically, to ensure thorough testing and reduce cost at the same time. Formal models also benefit other areas of development such as debugging designs, or regression test generation. Automated generation of regression tests can ensure that design information is kept up-to-date with changes in code.

Several researchers have proposed the use of model-checking technology to generate test cases from formal models. The technique has been applied to hardware[41], protocols[25][29], and specifications of simple control systems[3][43][20], but has never been applied to more complex models of software components. Previous applications have been necessarily simple as state explosion severely limits the size of models that can be model-checked.

Another obstacle in applying formal models is that they are seen as being difficult to create and understand, compared to informal models. The problem is more pronounced when modelling software components, as compared to hardware, protocols, or simple software specifications. The usual formal description techniques used for test generation tend to be hard to use, lack a friendly notation, or lack features for modelling software designs. Conversely, more friendly languages tend to be informal, or very difficult to model-check.

Thus, current methods are not feasible for testing component interactions. They either require too much manual effort for modelling and testing, or create models that are much too complex for analysis. To be feasible, two major obstacles must be overcome: the state explosion problem and the cost of creating formal models of software components. The goal of this thesis is to provide techniques to overcome these two obstacles and make component interaction testing from formal models more feasible. This goal is very ambitious as software design models are far beyond the ability of current model-checking technology.

1.1 Contributions

This dissertation presents results of research to make interaction test generation feasible.

First, it presents a formal, object-oriented modelling language, called ObjectState, which balances efficiency of analysis with ease of modelling. The language is based on the popular, but informal, object-oriented modelling notation UML for Real-Time[52], combined with a Pascal-like programming language to model local component data. A formal semantics is given for the language in terms of labelled transition systems (LTSs). LTS is a formalism used to study correctness of concurrent systems, and captures the essential characteristics of component interactions.

In addition, the language is extended to model formal test requirements and a formal definition is given for the event-flow coverage criteria[50] for selecting interaction test cases.

Unfortunately, ObjectState models tend to be too large for even advanced model-checking techniques to handle. Thus, this thesis also presents new algorithms to reduce state explosion for test generation. The new algorithms are specifically designed for test generation, and can achieve much greater reduction than previous algorithms for verification. The new algorithms remove many types of redundancy that cause state explosion.

To evaluate the techniques in this thesis, tools were created to compile and analyze ObjectState descriptions. A large example software model was modelled in ObjectState, the call-processing software of a private branch exchange (PBX), and was analyzed using the tools. The results showed that the model was reasonably easy to create, and that the algorithms allow useful subsets of the model to be analyzed, where previous techniques fail.

1.2 Overview

The next chapter reviews existing techniques of interaction testing coverage criteria; formal semantics for design modelling languages; and reduction algorithms for model-checking. Chapter 3 introduces the syntax and semantics of the ObjectState language, and compares it to other modelling techniques. It also presents the formal semantics for the Event-flow coverage criteria for interaction testing. Chapter 4 presents two algorithms to enable scalable model checking for test generation. Proofs of correctness are included, as well as comparisons with other reduction algorithms. Chapter 5 evaluates the effectiveness of the modelling language and the algorithms using an example model of a realistic software design. Chapter 6 presents conclusions and future work.

Chapter 2

BACKGROUND AND RELATED WORK

2.1 Chapter overview

This chapter starts with an overview of the models and test coverage in component interaction testing, and their advantages and limitations. It then describes previous techniques that use formal models to generate test cases, how they apply to component interaction testing, and their advantages and limitations. In particular, it discusses methods to handle state explosion. Finally, it describes the approach taken in this thesis, which attempts to adopt the advantages of previous techniques and address the limitations.

2.2 Testing concepts

Testing evaluates the quality of a program by running the program with actual input stimuli and examining the actual output behaviour. Testing is an appropriate technique for evaluating many quality factors, such as correctness, reliability, robustness, usability, performance, etc. This thesis focuses on correctness testing, which aims to detect the presence of defects in the program. In contrast, the related technique of reliability testing aims to evaluate the probability of failure of the program.

A program is defined to be correct if its output conforms to its specification for all input. Since it is impractical to run any significant fraction of the possible inputs of a program, correctness testing focuses on testing for the presence of likely defects of the program. A defect is a part of the program which does not satisfy its postcondition given its preconditions, or which requires preconditions that are not satisfied by the program. Usually these preconditions and postconditions are informal assumptions in the mind of the developer(s).

The key concepts in correctness testing are program models and coverage criteria. The program model represents some aspects of a program that affects correctness of the program. The coverage criteria selects tests based on the program model that are likely to reveal defects.

For example, path testing is a test selection technique based on the control flow graph, a representation of possible sequences of events in a program's execution. One coverage criteria is that each node in the control flow graph must be covered, which corresponds to selecting tests that exercise each statement in the program. Another technique, domain testing, is based on a partition of the possible values of a parameter to a program. The coverage criteria results in selecting tests that use an input from each class in the partition.

Program models and coverage criteria are formal if there is a mathematical relation between executions of the program and coverage of the model. For example, path testing is formal. In sequential, imperative programming languages, it is possible to define the situations under which statements in the language (i.e. nodes in the control flow graph) are executed. Domain testing is also formal, as it is possible to define the situations under which a parameter value is in a class of the partition. In contrast, for informal descriptions of a program, such as an informal flowchart, there is no mathematical relation between the test execution and when a statement has been covered. (Beizer's book[5] thoroughly covers these and other testing models and coverage criteria.)

Testing with formal models and coverage allows better control of quality because it is more independent of individual testers, and easier to quantify. Thus, the effectiveness of different coverage criteria can be experimentally validated and compared. At the same time, formal methods allow greater automation, and reduce cost.

2.3 Component interaction testing

This section discusses models and coverage that are relevant to testing interactions.

While many techniques exist for testing a whole program or individual program components, few models and coverage criteria exist for testing interactions between program components. The main obstacle has been obtaining models that capture information about interactions between components.

Unit and system testing models are simply not adequate. By definition, interaction errors cannot be found by looking at components in isolation. These errors occur when one component's precondition on the behaviour of the rest of the system is not satisfied, or its post-condition does not satisfy other components' assumptions.

For example, consider a part of a program that allocates memory but does not free it. This is not an error, necessarily, as another part of the program may be responsible for freeing it. In that case, it would be an error to free the memory twice. However, it often happens that no part of the program eventually frees the memory, resulting in memory leaks.

Thus, it can be seen that interaction errors cannot be found by unit testing, and are very difficult to find by system testing as they involve problems with internal actions. These actions are very difficult to observe and control from the system level.

Hence, the first requirement for systematically testing component interactions is a model that captures interactions, which in turn, requires a definition of component interactions.

2.3.1 Code-based interaction models

One view of interactions comes from the perspective of the source code.

- Call-graphs – Components are procedures, and an interaction is calling another procedure. Coverage requires each procedure call to be exercised.
- Interprocedural dataflow – Components are procedures, and an interaction is the definition of a variable in one procedure and the use of the variable in another procedure. Coverage requires each definition of a variable in one procedure and its subsequent use in another procedure to be covered.
- Module coupling – Components are modules. An interaction is calling a procedure in another module, passing values, data structures, references, or defining and using exported module variables. Offutt[56] defines coverage based on various levels of coupling. Module coupling coverage includes coverage from call-graphs (control-coupling) or interprocedural data-flow (data-coupling). It also covers, for example, "tramp coupling paths": paths from a definition of a variable in module A, through a call to B, to its use in C.

Other types of interactions can be concurrent object communication, such as Java synchronized methods and blocks, and Ada rendezvous.

Code-based models have the advantage that they are formal, and can be derived automatically from the code. One disadvantage is that many kinds of interactions cannot be seen from the source code. For example, when components are processes and interactions are inter-process communication, such as file I/O, message passing, remote procedure calls, and so on, these interactions are not distinguished from internal procedures. Another disadvantage is that the control-flow and data-flow information can be too detailed for testing interactions, and leads to too many test cases that overlap unit testing. At the same time, testing focuses on use and definition of variables, but not the actual values of variables. Values of variables also affect the component's interactions, and hence problems can be missed by the code-based techniques.

2.3.2 Design-based interaction models

In general, it is better to view interactions from the perspective of the object-oriented design models.

Design models capture both the code-level interactions and interactions not visible in the code. In an object-oriented design model, an interaction is a "set of messages exchanged among a set of objects to accomplish a particular purpose"[58]. They are abstract concepts to be designated by the designer. Testing from design models targets the interactions that exist in the mind of the designer—a more appropriate level of abstraction than source code or requirement specifications.

Many popular object-oriented modelling notations use informal, extended finite-state machines (FSMs) to specify behaviour of components. The notation defines messages and events, which cause the state machine to make a transition, or are caused by a transition of the state machine.

FSM models are convenient for defining informal interaction coverage criteria. One example is every transition from every state of a state machine must be exercised. Intuitively, this allows the testing of other components against all the "different cases" of behaviour of a component.

The MM-path approach[45] is more thorough than testing each transition. It requires the tester to identify MM-paths, which are sequences of transitions that are 'basic units' of interaction. Various heuristic rules are given to allow testers to identify MM-paths. MM-paths may overlap (have identical sub-sequences), and, presumably, every interaction must belong to an MM-path. Thus, this technique covers not only every transition, but also the uses of the transition in different cases.

Because the informal state machine models lead to informal test requirements, there is no mathematical relation between test cases and coverage. Thus, the tester must interpret the model and the test selection criteria, manually determine the tests to run, and subjectively evaluate coverage. For example, it is not possible to formally define when a transition has been "exercised", nor to generate a test case to exercise a transition. To allow automation, a formal modelling language is needed. With a formal model, the techniques can also be experimentally compared and validated objectively (independent of individual testers).

2.3.3 Formal interaction models

To overcome the limitation of testing from informal models various techniques have been proposed for testing from formal models.

In the Flattened REgular Expression (FREE) approach[7], the tester is required to derive formal state machine models suitable for generating executable test cases from the implementation. The tester creates the FREE state model by defining states as sets of values for component variables. That is, the component is in a state if the values of its variables are an element of the set of values of the state. Interactions are designated by the modeller, including function call and return, or interrupt and return. Transitions are triggered by interactions, and the destination states of the transitions are derived from the values of the variables after the interaction.

To test interactions among components, the local state models are combined into a FREE mode machine, with internal interactions hidden. From the mode machine, test selection is based on Chow's method[16]: breadth-first generation of a tree of transitions from the initial state, and exercising every branch in the tree.

This method is impractical, firstly, because it is very difficult to create the FREE state machine. Identifying appropriate partitions of variable values, and transitions between them is very labour-intensive. Worse still, the construction of the mode machine causes state explosion (Section 2.5), and makes test selection using Chow's method intractable.

2.4 Testing from formal models

The approach of testing from formal models has been explored many times. Much work has focussed on the theory of testing, formally relating coverage and test generation to some concept of correctness. Much of the work in protocol testing[8] has been in this category. Other formal techniques, such as ASTOOT[17], targeted system testing from formal specifications of individual classes, or modules, or systems as a whole. These works did not address the following issues that are basic requirements for practical interaction testing:

- creating complex interaction models,
- deriving formal test criteria and generating test cases,
- and analyzing large interaction models.

This thesis adopts the same basic ideas for test selection and test generation from state space models as previous studies, but focuses on efficiency of creating and analyzing models. Finally, there has been practical work on handling non-deterministic responses from the system, and generating executable test cases. While these issues are not addressed in this thesis, but would be important in actual applications to testing.

2.4.1 Formal modelling languages

The first problem is to create a formal model suitable for testing as easily as an informal design model. Rather than directly creating the formal state space models, as in the FREE method, they would be automatically generated from an object-oriented design model with a formal notation. However, many formal notations for state space models are not suitable for this task. Their mathematical concepts and syntax are unfamiliar and difficult, and they lack facilities for simple modelling of components and local component data. In addition, many notations generate state spaces that are too large to analyze for testing, or even infinite.

Previous formal testing techniques used notation designed for modelling hardware, communication protocols, software specifications, such as LOTOS[13], Esterel[43], SDL[28][20], and SCR[3]. Esterel and SCR focus on the control aspect and ignore the data aspect of systems. Thus, they are difficult to use for modelling software components interaction, which depend on complex data types. However, they do have the advantage of keeping models small and facilitating analysis. SDL and LOTOS simplify specifications by not setting *a priori* bounds on model elements (such as number of processes or communication buffer sizes), leading to huge or infinite models. They can support data types through the ACT ONE algebraic specification language, which also causes large or infinite models. As a result, only very simple systems can be modelled and analyzed. Of these, SDL has notation and features that are most suitable for object-oriented design models, but it produces the state spaces that are hardest to analyze due to its use of infinite buffers.

Researchers have also defined formal languages for object-oriented modelling for purposes of simulation, code generation, or verification of architecture, rather than test generation. Examples are vPromela[46], BDL[64], Wright[1], ROOM[62], and Rapide[53].

These languages are similar in that they simplify modelling of components by emulating features of popular, informal object-oriented modelling languages. These languages generally deal with three aspects:

- An architectural description language (ADL), with a graphical representation of components and connections
- A behavioural language, with a graphical state-machine representation of the behaviour of each component
- A data manipulation language, for detailed annotations of the effect of transitions on local component data. Annotations can be in informal text, a programming language, or a formal modelling language.

However, most of these languages still suffer from the problems of complex mathematical notation, lack of support for data, and/or generation of huge state spaces.

BDL[64] and Wright[1] focus on sequences of events and lack facilities to handle data structures. BDL extends IDL with specifications of allowable sequences of interactions. BDL semantics is based on partial-ordering of events, and has four different translations to LTS. Once in LTS form, the models can be used for test generation, but no applications for test generation have been given so far. Wright uses annotations in CSP[33], where the semantics are similar to LTS and are appropriate for test generation.

ROOM[62] and vPromela[46] facilitate handling of data by using imperative programming constructs. ROOM models can be annotated with C++, which allows code generation, but are hard to analyze for test generation. The vPromela language adds UML modelling features to Promela, the C-like modelling language for SPIN. Promela is designed to be efficient for analysis. However, Promela lacks local scopes for procedures and variables, making it unnecessarily difficult to model data and data types.

Rapide[53] also allows the definition of complex data types in a manner similar to algebraic specifications. However, its notation and semantics are based on constraints on data values and partial ordering of event sets. Thus far, the models have only been applied to simulation, and not model-checking or test generation.

2.4.2 Test selection from formal models

Once formal models have been obtained, there are various approaches to deriving test cases from them.

One class of methods follows the protocol conformance testing approach. These methods both select and generate test cases automatically from the state-based model. The FREE mode machine testing technique falls into this category. These methods assume the implementation is a formal FSM or LTS like the specification. They can generate tests that ensure a relation between the specification and implementation, such as the implementation has the same states and transitions as the specification. Direct application of these methods to software is not practical for realistic software models since they attempt to check all states and transitions of the model. The state explosion problem implies an explosion also in the number (or length) of tests. An overview of the methods in this area is given by Bochman and Petrenko[8].

Another class of methods does not attempt to blindly cover all states and transitions, but require 'other information' to partition behaviour into classes. Each class is called a test requirement since at least one test is required for each class. The 'other information' is given in several ways:

- User-specified pairs of events, for which sequencing constraints are derived from the model[13];
- Temporal logical requirement specifications, for which partitions of sequences are derived from the model [43];
- Mutation operations on the formalism used to create the model[3];
- User-specified sets of related events, for which event-flow sequences are derived from the model[50];
- Explicit test requirements in a formal notation such as MSC[28], extended TTCN[51], or Promela 'never claims'[20].

These techniques may be appropriate for generating test requirements for interaction tests if they are applied to models of component interactions, and the 'other information' focuses on interactions. The MM-path method can be applied using this approach, by writing each MM-path as a formal test requirement. Similarly, event-flow can be used for interactions by selecting related events to be interactions between components. More investigation is needed to determine the effectiveness of these methods for interaction testing.

2.4.3 Generating test cases for formal test requirements

Once the formal test requirements are created, the next step is to generate test cases using model-checking. Model-checking is a popular method for verifying models of hardware, communication protocols, or software requirements. Model-checking algorithms determine whether a temporal logic property holds for all sequences of transitions in a state space description. If the property does not hold, most algorithms can output a sequence of transitions of the state space that does not have the required property.

The same model-checking algorithms can be used to generate test cases by finding sequences of transitions to satisfy the test requirement, and then outputting the sequence. The sequence of transitions is the basis of the test case, since the transitions include information on inputs to the system, and outputs of the system (and also internal events in the system). For infinite models, such as for models in SDL, bounded search is used.

2.5 State explosion problem

Test generation using model-checking algorithms has complexity that is linear in the size of the state space. In general, the number of global states grows exponentially with the number of components and variables. This problem is known as state explosion. State explosion severely limits the size of models that can be analyzed, since adding a single variable to an analyzable model can make it impossible to analyze. Advances in computer hardware alone will have no significant effect on making larger models analyzable, and better algorithms are needed.

Using test requirements to generate test cases may alleviate the state explosion problem, since only states that belong to sequences that satisfy the test requirement need to be considered by the algorithm. However, test requirements may also exacerbate the problem if they do not sufficiently constrain the number of states, while requiring long sequences to satisfy.

In general, previous formal testing techniques have relied on standard model-checking tools and algorithms for verification. They have not attempted to create more efficient algorithms tailored for test generation. As a result, the applications have been necessarily simple, such as hardware (cache coherence protocol[41]), protocols (DREX[25], sliding window, etc.[13]), and specifications of simple control systems (cruise control[3], Automatic Protection Switching (APS) system[43], Intelligent Network[20]). Bounded search was applied to protocols (INRES[28] and SCCOP[29]).

2.5.1 State space reduction algorithms

Many algorithms have been proposed to deal with the state explosion problem. They are based on the fact that not all the information in the formal model is necessary for deciding whether some property holds for the state space.

A basic reduction technique is to generate the state space 'on the fly', that is, as required by the model-checking algorithm. The SPIN model-checker is a well-known implementation of this technique[34]. If the property can be checked without constructing the entire state space, the state space construction can stop early. In addition, search algorithms for model-checking only require that the states that have been explored be stored, but not the transitions between them. In the worst case, however, the technique only reduces time and space required by a constant factor.

Valmari[66] categorizes advanced reduction algorithms into four basic strategies:

- Model pre-processing
- Packed state space
- Process-algebraic compositionality
- Commutativity methods

These are described in more detail in the following sections.

2.5.1.1 Model pre-processing

Pre-processing techniques work during the construction of the state space from the model written in some modelling notation. They create a smaller state space by removing irrelevant information contained in the model.

One pre-processing technique is to eliminate remnant variable values. For example, an advanced implementation of this is included in SPIN[38]. Remnant variable values are values of variables that lead to the same possible future behaviours. For example, when the value of a variable is not used until it is eventually overwritten, it cannot affect the future behaviour of the system. Thus, instead of exploring states with a different value of the unused variable, this technique removes all the different remnant variable values by setting the unused variable to an "uninitialized" value.

Another pre-processing technique is coarsening of atomicity, where a sequence of transitions of a component's model is merged into a single transition (e.g. in SPIN[38]). This technique works correctly when the merged sequence does not contain multiple critical references, that is, interacts with other components multiple times (through shared variables, or other communication mechanisms).

These pre-processing techniques are subsumed by more powerful techniques using compositionality or commutativity, but pre-processing is useful, as it is easy to implement.

Pre-processing techniques for data-independent systems can reduce the number of data values used for a variable. Many systems such as protocols or cache memory are independent of the actual data being transferred or cached. Theories exist to find the minimum number of data values that are needed for verification[59].

2.5.1.2 Packed state space

Packed state space methods efficiently represent the states and transitions, using an implicit representation.

The supertrace method uses lossy compression on the bit vector representation of the states, using a hashing technique. Using lossy compression implies a possibility of incorrect answers, and there is only a constant reduction in the memory requirements. Data structures for lossless compression of states include GE-sets[30] and deterministic finite automata[57]. However, with lossless compression, there is always a chance that the size of states is actually increased.

The binary decision diagram (BDD) represents a set of bit-vectors efficiently by sharing sub-BDDs. BDDs can successfully compress systems with huge numbers of states. However, they rely on a good ordering of the bit-variables representing the bit vector, where the values of variables that are far apart in the order should not be dependent on each other. Hu[39] showed that they are unsuitable for high-level designs of systems, since different parts of the system depend on many other parts in a network. As a result, there are no good variable orders, and the BDD size explodes exponentially. Hu suggests the inter-dependency can be removed by replacing the dependent variables with explicit functions of the independent variables, but the task is very difficult and impractical for complex dependencies.

Symmetry reduction depends on bijections that preserve the structure of the state space (and the property being checked). That is, the bijection mappings preserve transitions between mapped states. By knowing the set of bijections, only one representative of an equivalence class of states needs to be explored and stored. The set of bijections can be computed dynamically, resulting in long computation times, or manually encoded into the model source code (such as in the Murphi[42] language), which may miss possible reductions. The amount of reduction depends greatly on the available symmetry of the system. For example, a system with a ring topology can have a constant reduction, while a fully connected topology can have an exponential reduction.

Other efficient representations have been proposed for systems with hierarchical structure[2] or reversible rules[40]. In these cases, some states can be derived easily from others, and hence only 'representative' states need to be stored.

2.5.1.3 Process-algebraic compositionality

Process-algebraic models of state spaces define the concepts of behaviourally equivalent models and composition of components. Compositional methods construct the global state space hierarchically, by replacing the state space of each component with a smaller, equivalent state space. The reduced components are composed and each composition, in turn, is replaced with a smaller, equivalent state space. At the end, the methods produce a reduced global state space equivalent to the full composition. The reduced model is equivalent in the sense that it can be used to check a predefined set of properties, and give the same answers as the original state space.

These methods reduce effort if the composition of subsets of components is smaller than composition of the whole system. However, that is not always the case as "a subsystem that is isolated from its proper context may exhibit lots of 'spurious' behaviour; that it does not have when it is a part of the system as a whole"[66]. To alleviate this problem, interface processes are introduced during the composition of subsystems, to restrict the number of spurious behaviours[63][15].

However, constructing an appropriate interface process for each composition is very difficult. The best interface process is simply the rest of the system, but that causes state explosion in the interface process. Proposed techniques allow user-defined interface processes. To prevent the possibility of user-defined processes restricting too many behaviours of the composition, and thus yielding incorrect answers in the model-checking, the correctness of the interface processes are also checked while checking the model property. If the interface processes are found to be incorrect, the user must try another one.

2.5.1.4 Commutativity

A major source of state explosion is the concurrent execution of a set of actions by concurrent components. The final effect of the set of actions is independent of the order in which the actions are taken, but different orderings result in different intermediate states. Commutativity-based algorithms[67][27] (also called partial-order algorithms) select actions that need not be explored from a state, using information about which actions that are guaranteed to be enabled later. The actions must not affect the property being checked. The algorithms depend on deriving guarantees from the modelling notation about enabled actions in future states. The better the information on guaranteed actions, the more reduction is possible. Nevertheless, it is not possible to obtain optimal reductions, as there is not enough information at a state to determine which actions should be explored immediately, and which should be deferred to later states. Overall, these algorithms eliminate an important source of redundancy.

2.5.2 Structural vs. semantic algorithms

Reduction algorithms can also be distinguished based on how much information they use about the system and the property being checked. For example, on-the-fly methods search the states of a system in the same order regardless of the meaning of the states of the system or the property that is been checked. The only information it cares about is the current state and possible successor states. On the other hand, commutativity-based reduction requires information about actions that are guaranteed to be enabled in the future. Compositionality-based reduction requires information about the hierarchical structure of the system, and perhaps abstractions of subsystems (as interface processes).

Taken to the extreme, the best technique is to manually abstract a model specifically for the property to be checked. Holzmann[37] showed it is possible to create very small models to check useful properties—less than 100 states. Obviously, manual abstraction is very difficult, but the important lesson is that the more specific the reduction algorithm, the more reduction is possible. For test generation from component interaction models, algorithms should take advantage of characteristics of component interaction models, such as models of many components with loose coupling, as well as characteristics of test generation, such as the ability to incrementally generate test cases.

2.6 Approach of thesis

Effective testing of component interactions requires formal models of interactions and formal test criteria. The most promising techniques use formal state space models. Formal state space models should not be created by hand, but should be automatically generated from object-oriented design models created in a friendly notation. In addition, the large size of object-oriented design models requires new algorithms specifically designed for generating test cases from them.

Chapter 3

FORMAL OBJECT-ORIENTED MODELLING

3.1 Chapter overview

In order to facilitate interaction testing, a formal model of the components' interactions must be created. The model can be obtained cost-effectively by reusing design models created for object-oriented analysis and design. Popular modelling techniques, like UML[60], are informal, so are not suitable for automatic test selection and test generation. Alternatively, most of the formal languages suffer from the problems of complex mathematical notation, lack of support for data types, and/or generation of huge state spaces.

The ObjectState modelling language was developed to show the feasibility of creating formal design models for interactions test generation. ObjectState balances the needs of a friendly notation for object-oriented design, formal state-space semantics for test selection, and an efficient resulting state space for test generation.

This chapter presents the syntax and formal semantics of ObjectState. The formal semantics is given as a translation of language feature to labelled transition systems (LTS). To support test generation, ObjectState notation and semantics are extended to capture formal test requirements, and event-flow coverage criteria is formally defined for ObjectState models. Finally, the language is compared to related work in modelling languages.

3.2 Formal models of interactions

The semantics of ObjectState models will be defined using labelled transition systems (LTSs). The LTS is a very popular mathematical model for correctness of concurrent systems. It is a simple, intuitive formalism that models distributed state, interfaces between components, sequences of interactions, and state-based interactions. It has mature theoretical foundations, including many research tools for analysis. Hence, it is very suitable for interaction testing.

3.2.1 Definition of labelled transition systems

The following definition of LTS follows Fernandez and Mounier[22]. An LTS is defined as a tuple (Q, A, T, q_0) where

- Q is a set of states
- A is a set of labels (representing actions or events)
- T is a set of transitions, $q_1 \xrightarrow{a} q_2$, where $q_1, q_2 \in Q$ and $a \in A$
- $q_0 \in Q$ is the initial state

A special label, τ, is used to represent actions that are "hidden". It is used to simplify the behaviour of an LTS model. The weak transition relation, written as $q_1 \overset{a}{\Longrightarrow} q_n$ where there exists a sequence of transitions

$$q_1 \xrightarrow{\tau} q_2 \xrightarrow{\tau} \ldots q_{n-1} \xrightarrow{\tau} q_n \xrightarrow{a} q_{n+1} \xrightarrow{\tau} \ldots q_{n+m} \text{ where } n \geq 1,\ m \geq 0)$$

Using the weak transition relation, the observable behaviour of the system is unaffected by any number of hidden actions. .

A system S in which only labels in a set L are visible, and the rest are hidden, is denoted $S{<}L{>}$.

3.2.2 Behaviour and equivalence of models

In an LTS, we are interested only in the sequences of actions that the model can have, and the actions that are enabled or disabled at each state. An action a is enabled at a state q if there is a transition $q \overset{a}{\Longrightarrow} q_2$ for some q_2.

In particular, the identities of the states are not important. In fact, two states have the same behaviour if they have the same set of enabled events, and the destination states of all transitions are considered equivalent; thus, they can be considered equivalent.

Formally, we define a family of relations between states:

- $R_0 \equiv Q \times Q$,

- $R_{k+1} \equiv \{(p_1, p_2) \mid \forall a \in A$
  $\forall p_1'(p_1 \overset{a}{\Longrightarrow} p_1' \Rightarrow \exists p_2'(p_2 \overset{a}{\Longrightarrow} p_2' \wedge (p_1', p_2') \in R_k) \wedge$
  $\forall p_2'(p_2 \overset{a}{\Longrightarrow} p_2' \Rightarrow \exists p_1'(p_1 \overset{a}{\Longrightarrow} p_1' \wedge (p_1', p_2') \in R_k)\}$

The (observation) equivalence relation is defined as $\sim \equiv \bigcap_{k=0}^{\infty} R_k$. Thus, $p_1 \sim p_2$ if $(p_1, p_2) \in R_k$ for all k.

Two LTS models are defined to be equivalent if their initial states are equivalent: i.e. the models always have the same behaviour, starting from the initial state.

For example, in the top LTS of Figure 3-1, the two states shaded grey are equivalent: both can perform action 'c' and nothing afterwards. Thus, the LTS is equivalent to the bottom one in Figure 3-1.

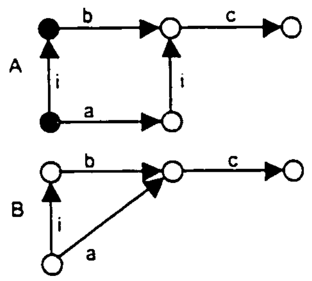

Figure 3-1 Observationally equivalent LTS

Note the two states shaded black are not equivalent, as one can perform 'a' and 'b', while the other can only perform 'b'. Thus, one state can perform all the behaviour of the other.

Many notions of equivalence have been defined for LTS, such as safety equivalence, trace equivalence, branching bisimulation equivalence, and so on. Because the purpose of this thesis is simply to use equivalence for deriving algorithms for model reduction, observation equivalence is sufficient: It preserves more than enough properties of the model for testing, so that it does not limit the kind of analysis that may be done after reduction. For example, using observation equivalence to reduce the model does not imply the implementation is required to be observation equivalent to the model. (In the language of protocol testing, the "implementation relation" does not have to be observation equivalence.) It is more suitable than the other equivalences because it is simpler to understand, and simpler for deriving algorithms.

3.2.3 Example: list structure

To illustrate how the definition of LTS can be used to model software, consider the example of a simple list data structure, with elements restricted to $\{1, 2, 3\}$. The list has the following operations:

1. new( ) --- make the list into a new (empty) list
2. append(x) --- append x to the current list
3. x=get(i) --- get the ith element of the list and return it (x) or return ERROR if there are less than i items

This example list omits the remove operation, since the example is already complicated without it.

The initial state is the empty list (q_0). The following transitions are possible from q_0:

- for get operations: as there are no elements in the empty list, all will return ERROR; the state of the data structure does not change (remains at q_0)

$$q_0 \xrightarrow{ERROR=get\,1} q_0$$

$$q_0 \xrightarrow{ERROR=get\,2} q_0$$

$$q_0 \xrightarrow{ERROR=get\,3} q_0$$

- the new also does not change the state since q_0 is already the empty list

$$q_0 \xrightarrow{new} q_0$$

- appending an element moves the data structure to a new state, depending on the element inserted

$$q_0 \xrightarrow{append\,1} q_1$$

$$q_0 \xrightarrow{append\,2} q_2$$

$$q_0 \text{ —}append\,3\text{→ } q_3$$

From q_1 we have transitions such as

$$q_1 \text{ —}1{=}get\,1\text{→ } q_1$$

$$q_1 \text{ —}ERROR{=}\mathrm{get}\ 2\text{→ } q_1$$

$$q_1 \text{ —}new\text{→ } q_0$$

$$q_1 \text{ —}append\,1\text{→ } q_4$$

The graph in Figure 3-2 shows the possible states of a list with 2 elements or less:

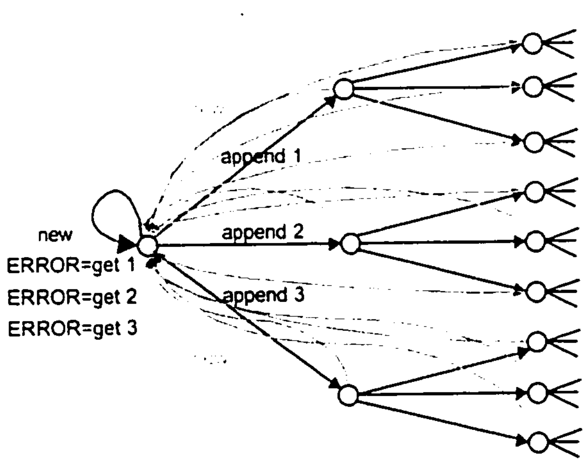

Figure 3-2 LTS of a list data structure with 2 elements or less. Grey transitions are labelled with 'new'.

This simple example is already very complicated. A remove operation would further complicate the diagram, by adding transitions to a previous level. Obviously, it is not convenient to create a LTS model directly. Instead, the LTS is usually obtained using a modelling notation.

So far, the example does not specify a maximum length of the list. If we assume a maximum length, say 3, then we can define the model of the list using an array (say of size 3) to store the elements of the list, and a variable to store the length of the list. We can label the states with the values of the variables. Thus, the state with 0 elements, and 1 in each element of the array is labelled as q_{0111}. Operations that change the value of the variables make a transition to the state with the new label. An example of a transition is

$$q_{0111} \xrightarrow{append2} q_{1211}$$

Note however, that there are more states in this model than the previous one. For example, there are many possible representations of the empty list: q_{0111} or q_{0123} or q_{0333}, etc. However, all the states are (observation) equivalent to each other. There are 84 states in the second representation, compared to 40 in the first, even though the two models have the same behaviour. The size of a system greatly affects the ability to analyze the system.

3.2.4 Model reduction

Models can be reduced by merging all equivalent states. Given a classification of the states of S, where $[p]$ denotes the class of p, the *quotient* of S, is the LTS $[S] = ([Q], A, [T], [q_0])$ where

- $[Q] = \{[q]\}$ for all $q \in Q$
- $[T] = \{[q_1] \xrightarrow{a} [q_2] \mid q_1 \xrightarrow{a} q_2 \in T\}$

If all the states in a class are equivalent, $[p_1] = [p_2]$ implies $p_1 \sim p_2$, then the quotient LTS is equivalent to the original LTS, $S \sim [S]$, and usually has fewer states.

For the example of the list, we can put all states representing the empty list (q_{0111} or q_{0123} or q_{0333}, etc) into one class, all states representing a list of the single element, 1, into one class, and so on. The quotient, with respect to this classification, is the original representation of the list.

3.2.5 Composition of models

LTSs have the ability to model concurrent processes and their interaction using composition.

The composition of two LTSs, $S = (Q_1, A_1, T_1, q_{10})$ and $T = (Q_2, A_2, T_2, q_{20})$, is the LTS $S \,||\, T$, defined as (Q, A, T, q_0) where

- $A = A_1 \cup A_2$,
- $Q = Q_1 \times Q_2$,

- T is the set of transitions of the form $(q_{11}, q_{21}) \xrightarrow{a} (q_{12}, q_{22})$ defined by one of the following rules
  - if $a \notin A_1 \cap A_2$ and $q_{11} \xrightarrow{a} q_{12}$, then $(q_{11}, q_{21}) \xrightarrow{a} (q_{12}, q_{21})$ for any q_{21}
  - if $a \notin A_1 \cap A_2$ and $q_{21} \xrightarrow{a} q_{22}$, then $(q_{11}, q_{21}) \xrightarrow{a} (q_{11}, q_{22})$ for any q_{11}
  - and if $a \in A_1 \cap A_2$ and $q_{11} \xrightarrow{a} q_{12}$ and $q_{21} \xrightarrow{a} q_{22}$, then $(q_{11}, q_{21}) \xrightarrow{a} (q_{12}, q_{22})$
- the initial state is $q_0 = (q_{10}, q_{20})$

In the composition $S \mid\mid T$, each component S and T can perform any actions not in $A_1 \cap A_2$ independently of the other. Otherwise, both S and T must perform the action in $A_1 \cap A_2$ simultaneously. Thus, composition models synchronous communications.

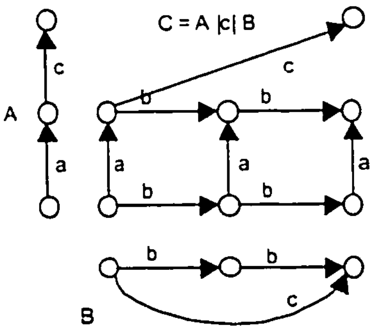

Figure 3-3 Composition of two LTSs

From the definition, the composition operation is commutative, $S \mid\mid T = T \mid\mid S$, and associative, $(S \mid\mid T) \mid\mid U = S \mid\mid (T \mid\mid U)$. Hence, we write multi-way composition as $S_1 \mid\mid S_2 \mid\mid \ldots \mid\mid S_r$. Note that the rules of composition imply that all components in a multi-way composition that share an action label will synchronize on that action (all components execute it at the same time, or not at all).

3.3 Formal interaction models with ObjectState

The ObjectState modelling language combines the architecture description features of UML for Real-Time[52], a hierarchical state-machine model for high-level behaviour of components, with a Pascal-like language for detail-level modelling of component data.

In the following, each feature of ObjectState is explained, its syntax is given in a version of BNF, and its mapping to LTS is described informally. An example is shown with the corresponding UML for Real-Time description.

In the syntax description, the notation '<<item>>' indicates a non-terminal symbol, '|' indicates a choice, while all other symbols are terminal symbols. White-space (including tabs or end-of-line) is only important as separators between symbols.

3.3.1 Component Configuration

The architectural description part of ObjectState is based on UML for Real-Time, the real-time extension of UML. UML lacks the notion of connections between components, but UML for Real-Time employs a more complete architectural description notation.

The collaboration diagram of UML for Real-Time describes the possible pattern of communications between a set of capsules (i.e. components) that execute concurrently. Capsules have no knowledge of the existence of other capsules outside themselves. Capsules

```
                <<system>> ::=  CONFIG {
                                  <<capsule instantiations>>
                                  <<port connections>>
                                };
  <<capsule instantiation>> ::=  COMP <<capsule-name>> => <<capsule-class-name>>;
          <<capsule class>> ::=  DEFINE <<capsule-class-name>> => PROCTYPE {
                                  <<port declarations>>
                                  <<proctype behaviour>>
                                };
       <<port declaration>> ::=  PORT <<port-array-name>> =>
                                      <<protocol-role-name>>;
          <<protocol role>> ::=  DEFINE <<protocol-role-name>> => PORTTYPE {
                                      <<message definitions>>
                                };
     <<message definition>> ::=  <<message-name>> => [ <<message params>> ]
          <<message param>> ::=  <<param-name>> => <<param type>>
        <<port connection>> ::=  CONNECT(<<capsule-port-index-1-name>> =>
                                           <<capsule-port-index-2-name>>);
                              |  EXTERNAL(<<capsule-port-index-names>>);
```

Figure 3-4 Syntax of component configurations

communicate with the rest of the system or with the environment through their ports only. In particular, there are no global data variables. Connections between ports determine which capsules communicate with which capsules. Ports are associated with a protocol that defines the messages that may be exchanged via the ports. Capsules can contain other capsules hierarchically in UML/RT, but not in ObjectState.

An ObjectState system is defined by instantiating capsules and connecting their ports (Figure 3-4). A capsule instantiation declares a capsule of a given name to be of a given capsule class. The ports and the behaviour of a capsule are defined in its capsule class. A port declaration declares a port of a given name and multiplicity to have a given protocol role. The protocol role defines the messages that can be sent on a port.

The configuration diagram maps to its LTS semantics in a very simple way: A system maps to a composition of LTS models, capsules to simple LTS models, a protocol role maps to a set of event labels, and connections map to renaming of event labels. The renaming function replaces the port identifier in the labels with the connection identifier, so that the same messages sent along the same connection are given the same labels, but the same messages sent along different connections are given different labels. Ports of compatible protocol roles are connected using `CONNECT`. If a port is not connected to another capsule, but it is visible to the environment of the system, then it is declared `EXTERNAL` Otherwise, a port is "disconnected" and no messages will be exchanged at that port.

3.3.1.1 Example

This example shows how the following system is written in ObjectState, and represented as an LTS. The UML for Real-Time collaboration diagram in Figure Figure 3-5 specifies `User` is a capsule of class `UserClass` and `Counter` is capsule of class `counterClass`. The multiplicity of `User` is 2, hence there are two capsules, called `User[1]` and `User[2]`. `User[1]` has a port `User[1].p`, and `User[2]` has `User[2].p` of the same protocol role (porttype), `counterProtocol`. The `Counter` has two ports, `Counter.p[1]` and `Counter.p[2]`, whose protocol role is also `counterProtocol`.

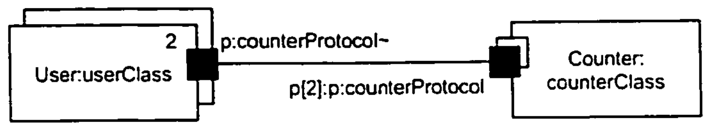

Figure 3-5 Configuration diagram of example system

There is ambiguity in the connection line drawn: It can represent two pairs of connections, either

`User[1].p` connected to `Counter.p[1]` and `User[2].p` connected to `Counter.p[2]`

or

`User[2].p` connected to `Counter.p[1]` and `User[1].p` connected to `Counter.p[2]`.

The following ObjectState description assumes the first pair of connections is desired:

```
DEFINE 'userClass' => PROCTYPE {
  PORT 'p' => 'counterProtocol';
  ...
};
DEFINE 'counterClass' => PROCTYPE {
  PORT 'p[1..2]' => 'counterProtocol';
  ...
};
CONFIG {
  COMP 'User[1]' => 'userClass';
  COMP 'User[2]' => 'userClass';
  COMP 'Counter' => 'counterClass';
  CONNECT('User[1].p' => 'Counter.p[1]');
  CONNECT('User[2].p' => 'Counter.p[2]');
};
```

The configuration diagram did not show the messages in `counterProtocol`. Assume the protocol is defined as follows:

```
counterProtocol = PORTTYPE {
  'on' => [],
  'off' => [],
  'incr' => ['lock' => 'boolean'],
        'decr' => [],
        'val' => ['value' => 'integer'],
    };
```

The protocol defines message names `on`, `off`, `incr`, `decr`, and `val`. The `incr` message has a parameter `lock`, of type `boolean`, and the `val` message has a parameter `value`, of type `integer`.

The formal semantics of the model consists of three LTS models, `User[1]`, `User[2]`, and `Counter`. Transitions in the LTS models are originally labelled as `port.message`. For example, in `User[2]`, the label `p.incr(false)` means send or receive `incr(false)` via the port `User[2].p`.

To allow communication through connections, messages are renamed on both components to the same name. For example, suppose the identifier for connection between `User[2].p` and `Counter.p[2]` is `Conn2`. Then, for the components to communicate through the signal `incr(false)`, rename `p.incr(false)` in `User[2]` to the label `Conn2.incr(false)` and rename `p[2].incr(false)` in `Counter` also to `Conn2.incr(false)`.

3.3.2 Component Behaviour

Component behaviours are modelled at a high level by hierarchical finite state machines (FSM).

States have names, transitions, and substates. States may define exit actions, entry actions, and signal handlers. All transitions of superstates automatically become transitions of all substates.

Transitions and actions can consist of several steps. The syntax and semantics of steps are similar to Promela[36], which in turn is based on CCS and CSP. Steps are composed of alternative sequences of atomic actions of the component, called basic steps. Note that this implies transitions of the FSM model are not atomic. Only basic steps are atomic.

A transition waits for conditions that allow the execution of the first basic steps (called triggering conditions) in each alternative sequence. Once a transition has begun executing, it will not be interrupted even if the triggering condition for other transitions become true; this is the so-called run-to-completion semantics[52]. By default, a transition returns to the same

```
<<proctype behaviour>> ::=  <<data declarations>>
                            <<states>>
           <<state>> ::=    DEFINE 'state-name' => STATE {
                              <<substates>>;
                              <<transitions>>
                            };
        <<substate>> ::=    SUBSTATE <<state-name>>;
      <<transition>> ::=    WAIT {<<steps>> };
                       |    INIT {<<steps>> };
                       |    ONEXIT {<<steps>> };
                       |    WHEN <<signal-name>> => HANDLE {<<steps>>};
                       |    DEFINE <<signal-name>> => SEQUENCE {<<steps>>};
            <<step>> ::=    <<basic-step>>
                       |    ENTER <<state-name>>;
                       |    CALL <<signal-name>>;
                       |    IF {<<steps>>} OR {<<steps>>} ... OR {<<steps>>};
                       |    DO {<<steps>>} OR {<<steps>>} ... OR {<<steps>>};
                       |    GOTO <<label-name>>;
                       |    LABEL <<label-name>>;
      <<basic-step>> ::=    OP <<condition>>, <<send|receive>>, <<action>>;
                       |    SELECT [<<var-name>> => <<var-type>>]
                              <<condition>>, <<send|receive>>, <<action>>;
```

Figure 3-6 Syntax for component behaviour

state, unless a new destination state is specified with ENTER. If the transition enters a new state, the exit and entry actions are executed in the order of states exited and entered.

Transitions in substates can raise signals to be handled by superstates, similar to exception handling in many programming languages. The CALL operation causes the handler defined in a superstate for '`signal-name`' to be executed. If no superstate defines a handler for this signal, it is simply ignored. If several superstates define a handler for the same signal, they are executed in order from the innermost superstate to the outermost. Control is returned to step after the call operation, unless the signal handler enters a new state.

Subroutines similar to those in procedural programming languages are also supported. They can be defined using SEQUENCE, and can be called in any state of the component using CALL.

Each basic step can have any combination of the following parts:

- a condition on local variables, and message parameters if receiving a message
- a message send or receive
- operations on local variables

A useful extension of a basic step is the non-deterministic basic step, which non-deterministically selects a value from a range of possible values.

Transitions can consist of more complex steps that do different processing based on different interactions and conditions. In an 'IF-OR' step, a sequence of steps is chosen to execute depending on the triggering condition of the initial step of each sequence. If conditions for several sequences are true, one sequence is chosen non-deterministically. If none are true, the process waits until one is true. It is similar to the '`if :: fi`' statement in Promela. The 'DO-OR' step loops continually, choosing an alternative and executing it. The loop can be exited using GOTO or ENTER. The destination of the GOTO is the location of the LABEL statement for the same '`label-name`'.

The translation to LTS is accomplished by expanding the FSM to a graph of basic steps, where each basic step is linked to other basic steps by equating their sources and destinations.

The steps of a single transition are linked as follows:

- The source of the first step in a transition is the transition's source state.
- The destination of the ENTER step is the destination state.
- By default, a transition loops back to its source state, so the destination of the last step of the transition is the source state.
- The source of each succeeding step is the destination of the previous step.
- The destination of the GOTO step is the source of the step after the LABEL statement.
- Every alternative step in an IF or DO construct has the same source. The destination of the alternative steps of a DO construct loop back to the source.

The hierarchy of states is handled by expanding the model:

- Transitions of superstates are added to the set of transitions of substates. That is, for each transition of the superstate, each substate gets a copy of the transition, where the source of the first step is set to the substate.
- Signal handler steps of superstates are pasted into the location of the CALL steps. This means the source of the first step in the signal handler is the source of the CALL step, and the destination of the last step is the destination of the CALL step.
- Entry and exit steps of corresponding states are pasted into the location of the ENTER steps.

The above rules lead to a confusing feature of ObjectState: If a substate and superstate both define a transitions that are enabled under the same conditions, the transition to execute is chosen non-deterministically. Users may rather prefer to have the substate's transition override the superstate's transition. Instead, this design choice stays close to the language's formal roots in CSP and CCS[54].

If the model does not contain any variables, then the basic steps immediately map to its LTS semantics:

- Each source and destination of a step is a state in the LTS, and each step is an LTS transition between its source and destination LTS states.
- Steps without a send or receive are internal steps, labelled with τ.
- Steps with message send or receive are labelled with `portname.messagename(params)`

Note that the translation into LTS semantics does not distinguish between sends and receives. Thus, it is possible to abuse the notation and cause components to communicate via a pair of sends, or a pair of receives. For simplicity of modelling, the compiler should ensure that only one side of a connection is allowed to send a given message, while the other side is only allowed to receive that message. Also, by not distinguishing sends and receives, the semantics allows further extensions of the language that use simultaneous, multi-way synchronizations.

With data variables, the mapping must take into account the values of the data variables, as explained in the next section.

```
<<data declaration>> ::=  VAR <<variable-name>> => <<Murphi-type>>;
                      |   CONST <<symbol-name>> => <<Murphi-value>>;
       <<condition>> ::=  '[<<Murphi-expression>>]'
            <<send>> ::=  '<<port-name>>!<<message-name>>(<<params>>)'
         <<receive>> ::=  '<<port-name>>?<< message-name>>(<<params>>)'
           <<param>> ::=  <<Murphi-expression>>
                      |   [<<Murphi-expression>>]
                      |   _
          <<action>> ::=  '<<Murphi-statements>>'
```

Figure 3-7 Syntax for transition annotations

3.3.3 Transition annotations

ObjectState provides a very simple programming notation for actions on data, using the Murphi modelling language[42]. Murphi was chosen because it manipulates data procedurally, using syntax and capabilities similar to Pascal: procedures, records, arrays, iteration, recursion, etc. Also, it supports automatic verification, by restricting data types to be finite, and disallowing dynamic structures (e.g. pointers). Finally, the Murphi compiler provides a convenient way to translate basic steps into LTS.

Murphi has features to support efficient verification, such as scalarsets, or multisets, but ObjectState does not currently support these features. Murphi syntax will not be discussed further, aside from noting that it is similar to Pascal. Interested readers may refer to the Murphi manual[42].

A basic step without send or receive is executable if the Murphi expression of the condition (if any) evaluates to true using the current value of the local variables. If the basic step is sending a message, the step is executable if, in addition, the component receiving the message has a receive step that is executable. If the basic step is receiving a message, the triggering condition is satisfied if, in addition, the component sending the message has an executable send step, and the message parameters satisfy the constraints in the receive.

The constraints on receive parameters are as follows:

- `<<Murphi-expression>>` signifies any value is acceptable, and the value is assigned to the value of the expression (the expression must evaluate to an 'l-value')

- `[<<Murphi-expression>>]` signifies the parameter must evaluate to the same value as the expression inside the brackets
- _ signifies any value is acceptable. The value is not stored.

Translation of data to LTS proceeds as in the example in Section 3.2.3. The data in the capsule is expanded into the states of the LTS. Values of variables are used to label the LTS states.

Given a state from the FSM translation, it is expanded to a set of states, indexed by the values of the variables. For example, if there are three variables, and the values of the '`var1`' is 3, '`var2`' is 5, '`var3`' is 1, then a previous state p will be mapped to a set of states, including the state p_{351}.

A transition between states p and q is expanded to a set of transitions, from the values of the variables before the transition's action to the value of the variables after the transition's action. For example, if the transition's action is '`var1 := 2`', then the LTS transition will be p_{351}—τ→q_{251} where the values of the variables in the first state are 3, 5, and 1 respectively. A SELECT step is translated to a set of transitions, one for each value of the variables in the SELECT.

A send evaluates the expression in each parameter and puts the value in the label. For example, '`p!msg(var1+1)`' creates the label `p.msg(3)`. A receive creates a transition for each possible parameter value that satisfies the parameter constraints. The labels of the transitions are simply the parameter values. For example, '`p?msg(var1)`' creates the transitions with labels `p.msg(1)`, `p.msg(2)`, `p.msg(3)`, assuming that `var1` has domain in 1..3. In the destination state, the variable will be assigned the value of the parameter (e.g. `var1` will equal 1, 2, or 3 depending on the message sent).

Note that each step execution is still translated into an atomic transition of the LTS. More complicated data manipulations using procedure and loops are treated the same as a simple assignment: as an atomic step.

3.3.3.1 Example

The example is the behavioural model of a counter from 1 to 3. The counter responds to the messages as follows:

- On – starts the counter with an initial value of 1
- Off – deactivates the counter (disallow increment and decrement)
- Incr(lock) – increments the counter if possible (if count < 3), and respond with the new value of counter; if not possible, the user must wait. In addition, if the parameter 'lock' is 'true', locks the counter, so that no increment is allowed until the counter is decremented
- Decr – decrements the counter, unlocking it if it had been previously locked

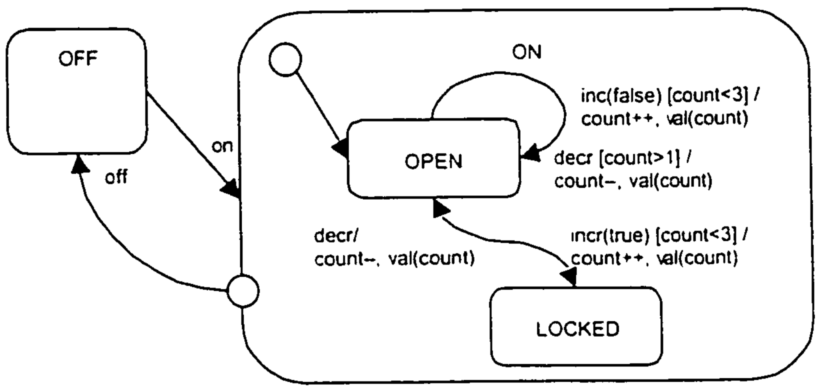

Figure 3-8 State chart diagram of counter component

The UML state machine in Figure 3-8 has annotations in a C++-like notation. The state machine is represented in ObjectState as

```
CONST 'MAXCOUNT' => '3';
VAR 'counter' => '1..MAXCOUNT';

DEFINE 'OFF' => STATE {
  WAIT { OP 'p?on()'; ENTER 'ON'; };
};

DEFINE 'ON' => STATE {
  SUBSTATE 'OPEN', 'LOCKED';
  INIT { OP 'counter:=1'; ENTER 'OPEN'; };
  WAIT { OP 'p?off()'; ENTER 'OFF'; };
};

DEFINE 'OPEN' => STATE {
  WAIT { OP '[counter< MAXCOUNT]', 'p?inc([true])', 'counter:=counter+1';
```

```
      OP 'p!val(counter)'; ENTER 'LOCKED';
    };
    WAIT { OP '[counter< MAXCOUNT]', 'p?inc([false])', 'counter:=counter+1';
      OP 'p!val(counter)';
    };
    WAIT { OP '[counter>1]', 'p?dec()', 'counter:=counter+1'; };
  };

  DEFINE 'locked' => STATE {
    WAIT { OP 'p?dec()', 'counter:=counter+1'; ENTER 'OPEN; };
  };
```

Without taking into account the data, the corresponding LTS is shown in Figure 3-9.

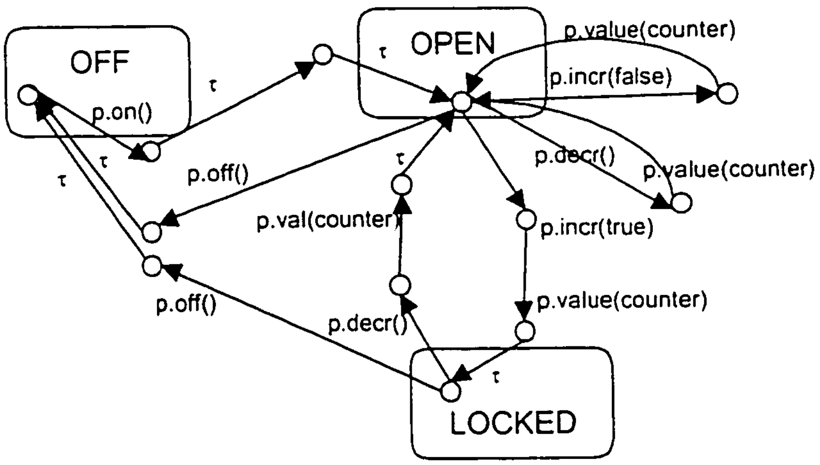

Figure 3-9 Counter model expanded to LTS

With data, the expanded LTS in Figure 3-10 shows that the OPEN state is expanded into three states (for value of the counter = 1..3). The LOCKED state is expanded into two states (counter cannot equal 1 if locked). The OFF state and τ transitions have been left out to reduce clutter.

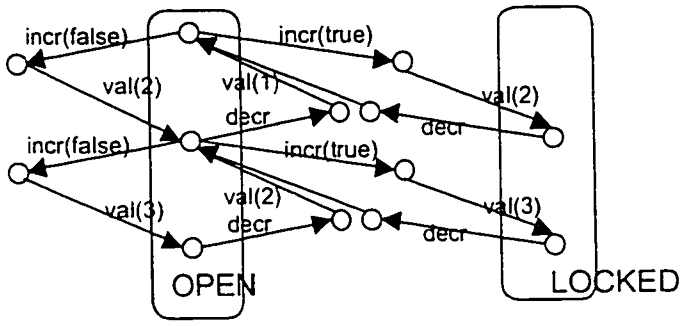

Figure 3-10 The OPEN and LOCKED states of counter LTS expanded with data, τ transitions hidden.

3.4 Formal test requirements with ObjectState

Test requirements, also called test purposes[8], specify interesting sequences of component interactions that should be tested, where the definition of "interesting" is left to the tester. Test requirements are usually written informally. However, formal test requirements allow automatic test generation, can be reused when system is modified, and can be analyzed for regression test selection.

The tester should only specify the important interactions in the sequence, not every detail of inputs and outputs that needs to take place to cause the important interactions. For example, for Jorgensen's MM-paths testing technique, the key interactions comprising the MM-path can be identified and specified formally, without specifying all the data values.

3.4.1 Extension to ObjectState

To write formal test requirements, ObjectState is extended. ObjectState can already specify sequences of interactions a component can have with other components. The same method is used to model a test requirement, only it is specifying interesting interactions. The only additional concept needed is to know when a test requirement is satisfied: this is specified by the ACCEPT step. Test requirements connect to port of components whose interactions are of interest, using the TESTCONN declaration. Then, the interactions of the component must match one of the alternatives specified in the test requirement or the test requirement will not be satisfied.

```
             <<step>> ::=  ACCEPT;
    <<port connection>> ::=  TESTCONN(<<test-capsule-port-index-1-name>> =>
                                  <<test-capsule-port-index-2-name>>);
```

Figure 3-11 Additional syntax for test requirements

3.4.2 Formal semantics

The test requirement is composed with the rest of the system in exactly the same way as other components. By LTS semantics, all components that have the same labels synchronize on those labels. This implies that interactions not allowed by the test requirement are not enabled in the composed system. The ACCEPT statement generates a special action labelled *ACCEPT*, which signifies that the execution of the system has satisfied the test requirement.

3.5 Formal coverage criteria with Eventflow

With a formal semantics for state-based models, it is simple to define a formal relation between coverage criteria and the model. This section presents as an example, the formalization of event-flow coverage[50]. Event-flow exercises pairs of 'related interactions' in the component models. Related interactions are user-specified. For example, they can be the set of sends and receives on a particular port.

Event-flow definitions are motivated by data-flow testing. To determine which sequences of interactions to test, event-flow looks at the place in the FSM model the interaction occurs. Even if the same interaction occurs in two places in the model, separate tests would be required for the interactions. The idea is to use the location of the interaction to partition interactions into different 'cases'—the interactions are assumed to occur in different parts of the global state space.

3.5.1 Definitions

To enable testing based on location of interaction, the event-flow definitions are based on steps of the model. A *related step* is a step in transition of the design state-machine model with a related interaction. A *related path* is defined as a path through the design state machine that starts with the first related step, ends with the second related step, and between them, there are no other related steps.

Several levels of coverage can be defined using related paths. The most thorough is all related paths are considered test requirements. Due to the presence of loops, the number of test requirements can be excessive. A less stringent requirement to test one related path for each related step.

A promising level of coverage is to test one related path for each pair of related steps. This level is more thorough than simply exercising every interaction in the design state-machine models (assuming every interaction is part of a related set of interactions). The idea is that the first related interaction creates an erroneous global condition, and the second related interaction could detect the error. It is less thorough, but more practical, than exercising every transition in the global composition of state machines.

Note that the problem of infeasible paths from dataflow testing exists here also. Some related paths cannot be executed once data variables are taken into account, or paths that are executable in the individual state machines cannot be executed once composed with the rest of the system. The advantage of automatically generating test cases to achieve coverage is that infeasible paths can be discovered automatically (if no test can be found to satisfy the test requirement). Infeasible requirements are not taken into account when evaluating the test suite coverage.

3.5.2 Formal semantics

The formal semantics of coverage depends on the level of coverage. The basic idea is to create a test requirement for each path, or each pair of related steps, or each related step, as appropriate.

The test requirements described in the previous section can only match the interactions to take place, but not the step in the model where the interaction is occurring. To allow test requirements to specify which steps must be executed, the formal model of the ObjectState components must be altered: an additional transition is added after each step in the component's state machine, labelled with the identifier for the step. Then the test requirement can specify the exact set of steps that must be taken.

For the requirement of one related path for each pair, only the steps with related interactions are labelled with the step identifier. Suppose the pair of related steps have identifiers *Step*1 and *Step*2, and all steps with related interactions are identified as *Step**. Then, the following test requirement LTS is added to the system:

$$q_0 \xrightarrow{Step1} q_1 \xrightarrow{Step2} q_2 \xrightarrow{ACCEPT} q_3$$

$$q_0 \xrightarrow{Step^*} q_0$$

The LTS specifies that the second related step must directly follow the first related step without any other related steps in between, which is the definition of a related path between the two steps. Any number of unrelated steps may between the two steps.

3.6 Related work

This section compares ObjectState with previous modelling languages, based on modelling features, formal semantics, test requirement notation, and definition of interaction coverage.

3.6.1 Modelling features

In comparison with informal modelling notation, such as UML, ObjectState lacks many useful features for design modelling. However, ObjectState does not need all the features of UML since its purpose is to demonstrate the feasibility of using a formal design language to test interactions. Defining a formal action semantics for UML is a very complicated task[55], especially the interaction between behavioural diagrams with the class diagram (such as associations between classes). Some features, such as hierarchical components, are convenient for organizing models, but introduce no new semantic elements.

Other features are left out because they result in large models, notably asynchronous communications, dynamic component creation and dynamically allocated data structures, passing and interacting with references to components. If needed, all these features can be simulated by explicit buffer components for asynchronous interaction, and arrays and indices for dynamic processes or passing references.

Note that synchronous models are sufficient for systems that are implemented using the following communication mechanisms:

- sequential, event-driven object-oriented programs interacting through class methods
- concurrent programs interacting through Java synchronized blocks or methods, Ada rendezvous, Occam message passing, remote procedure call, send-receive-reply, or other synchronous communication methods
- desynchronization—a provably correct asynchronous implementation of synchronous design[6].

As noted in the example in Section 3.3.1.1, ObjectState resolves an ambiguity in the graphical configuration diagram of UML for Real-Time regarding connections between multiplicity of ports and capsules. One way to resolve the ambiguity is to allow every port on one end of a multiple connection to connect to every port on the other end. This means each message send becomes a broadcast. However, if one port can connect to many ports, there is no need for multiplicities on ports. Instead, ObjectState requires the modeller to explicitly specify a connection for each pair of ports.

In addition, ObjectState state machines are simpler than UML state charts. For example, there is no concurrency within a component, nor is history kept when states are exited. Transitions of the FSM model are not atomic, and only basic steps are atomic. This avoids problems with the semantics of 'micro-steps' in state charts.

Another difference is the handling of exit actions. In UML for Real-Time, a state is exited as soon as an exiting transition is enabled. Thus, exit actions of the state are taken before the actions of the transition. In ObjectState, states are exited only when the ENTER step is taken, leading to a simple understanding of exactly when exit and entry actions take place. This semantics also obviates the need for special handling of 'chaining states'; they are treated the same way as regular states.

Most object-oriented modelling languages employ a graphical notation for architecture and behaviour models. Currently, ObjectState is textual, similar to StateText[32]. However, it would be simple to adapt a graphical representation and editor from another language.

Compared to other formal modelling languages, ObjectState is unique in its use of imperative programming notation for data manipulation. ObjectState allows the free use of data structures, which greatly simplifies modelling. However, it also causes large models. Rather than burden the modeller with creating an efficient model of the data, it relies on automatic algorithms to reduce the model.

The vPromela language adds UML modelling features to Promela, the C-like modelling language for SPIN. Hence, it is similar to ObjectState, except Promela has limited ability to model data types. The models can only be analyzed using SPIN. In addition, a version of ObjectState using Promela is also available, but it is slightly more difficult to use than the Murphi version.

Some formal notations allow protocol roles to have behaviour models, for example, Wright, Rapide, or, in theory, UML/RT. ObjectState does not. Behaviour in protocol roles allows checking for observance of the rules of the protocol, similar to an advanced form of type-checking. However, behaviours for protocols are not needed to model component interactions or to generate tests, and they are not used by object-oriented design methods.

3.6.2 Formal semantics

ObjectState's semantics is better for test generation compared to other modelling languages. The LTS semantics provide a simple definition of formal coverage criteria and efficient algorithms for test generation. Partially-ordered event sets semantics in Rapide, and programming languages like C++ in ROOM do not have efficient algorithms for test generation (i.e. model-checking).

ObjectState's semantics is weaker than others for verification, simulation, or code generation. For example, using C++ for the detail level in ROOM enables more efficient code to be generated. Also, use of arbitrary, though finite, data types makes verification more difficult. Nevertheless with the appropriate tools, ObjectState can fully support verification, simulation, or code generation.

An advantage of ObjectState's semantics is the translation of each step execution into an atomic transition of the LTS. Thus, complicated manipulations of local data using procedure and loops are performed in an atomic step. The only situations that require multiple steps are sends, receives, and non-deterministic executions. Compared with Promela, this removes the need for the '`atomic`' or '`d_step`' statements, as well as complex statement-merging algorithms.

3.6.3 Formal test requirement notation

There have been several notations proposed for writing formal test requirements, such as message sequence charts (MSC)[28], an extended version of TTCN[51], or Promela's 'never' claims[20]. MSCs have the advantage of being a graphical notation, but cannot easily specify complicated alternative sequences. TTCN[4] is a standard language for writing test cases, so extended TTCN can be useful in situations where TTCN is already familiar. Similarly, 'never' claims are useful when Promela is used as the modelling language, and ObjectState's test requirement extensions are useful when ObjectState is also the modelling language.

The Assertion Definition Language (ADL)[61] is an example of a formal test requirement language for testing correct data handling, rather than correct sequences of interactions.

3.6.4 Formal interaction coverage criteria

Event-flow is just one of many coverage criteria that may be defined for ObjectState models. It has the advantage of not requiring much information other than the models (only the definition of related events, which can default to all the interactions at a port). Also, it attempts to strike a balance between covering each step in each component, and covering each step in the global composition. However, the various approaches listed in Section 2.4.2 should be evaluated.

3.7 Chapter conclusions

This chapter presents the object-oriented modelling language, ObjectState, and shows how it is useful for modelling designs. The language combines architecture description features of UML for Real-Time with a Pascal-like language for modelling component data. At the same time, it has a formal semantics that allows it to be reused for interaction testing. Formal test requirements can be defined and test cases can be generated efficiently. However, design models of components tend to generate large state spaces. Effective reduction algorithms must be devised to reduce these state spaces to enable test generation.

Chapter 4

SCALABLE ALGORITHMS FOR TEST GENERATION

4.1 Chapter overview

Formal software component models generate large, complex state spaces, which are much more complex than models of hardware, protocols, or software specifications. Existing algorithms for state-space reduction are not sufficient for generating test cases from software design models. This chapter describes two new algorithms specifically developed to efficiently generate test cases from component models: incremental test case generation and interaction abstraction.

4.2 Strategies for scalable algorithms

Test generation from design models of components must overcome different challenges than verification of hardware, protocols, or software specifications. Difficulties include the following:

- Software designs have a greater number of complex components that interact with many other components
- Components use data and complex data types, and there are complex dependencies between data of different components
- Testing models must be general and detailed: it is not possible to create a specialized abstraction for checking each property

These characteristics prevent the use of abstraction mechanisms used in verification, such as BDDs, normal compositional techniques, or manual abstraction.

On the other hand, test generation is simpler than verification of complicated properties. Test generation is simply a check for reachability of a state. In addition, software models appear more complex than they are. Much of the size of software models comes from the large number of different functionality and aspects that software must deal with. However, when testing a particular aspect, most of the complexity in other aspects can be ignored.

As discussed in Section 2.5.2, reduction algorithms that take advantage the characteristics of component models and test generation will perform better than algorithms that are more general. There are three basic strategies:

- Slicing: remove the other parts of the model that do not affect the parts of interest.
- Abstraction and incremental refinement: remove details of model and analyze; gradually add details to the computed result.
- Compositional analysis: do as much work locally as possible, as effort spent reducing each part can pay off exponentially in the composition; avoid composing loosely-coupled components, as unconstrained composition presents the worst case of state explosion.

Two new algorithms based on these strategies are presented.

4.3 Incremental test case generation algorithm

The first algorithm takes advantage of the fact that test cases can be generated incrementally by parts. The algorithm computes partial test cases, with a subset of inputs and outputs, then expands them to a full test case. By taking a subset of inputs and outputs at a time, the other inputs and outputs can be hidden. Hence, the complexity of the model involved in generating the behaviour of the other inputs and outputs can be abstracted away and the system greatly reduced. The algorithm makes use of slicing and abstraction with incremental refinement by abstracting away complexity associated with other inputs and outputs, and then adding more inputs and outputs to the partial test case. The effort to extract each portion of the test case can be exponentially less than the effort to extract the entire test case at once.

For the purposes of this thesis, a test case can be considered a sequence of inputs and outputs. In practice, a test case should be more than a single sequence. For example, it needs to handle non-deterministic output. Previous works have dealt with these issues[23][28], but this thesis is concerned with reducing the complexity of analysis. Hence, all that is required to generate a partial test case is a model-checker that finds a path to *ACCEPT*, and keep the sequence of inputs and outputs along the path.

The algorithm proceeds as follows:

1. Select one of the ports in the ObjectState model declared as EXTERNAL, and hide all the messages at the rest of the EXTERNAL ports.
2. Minimize each LTS of the components, and extract a path to *ACCEPT* using a model-checking algorithm (preferably an on-the-fly algorithm, to avoid constructing the composition).
3. Create a new LTS with the sequence of inputs and outputs along the extracted path. This new LTS is a portion of the final test case.
4. Compose the new LTS with the system to constrain the behaviour of the system with the inputs and outputs selected so far. The new LTS restricts the number of possible behaviours of the system, so it hopefully reduces the size of the system for model-checking.
5. Select the next external port and repeat the above until no external ports remain.
6. The last search for a path will result in a path with all the external inputs and outputs of the system, and thus is the final test case.

The algorithm is very simple, but can be very effective when the number of external ports is large compared to the number of internal connections.

4.4 Interaction abstraction algorithm

The standard observational minimization algorithms preserve all the internal interactions between components, *even if those interactions cannot lead to different external behaviour.* Clearly, if there are few external actions of interest, and there are many internal connections, then there is great redundancy being left in the LTSs of components. The second algorithm improves the abstraction of LTSs of components by abstracting interactions that do not lead to different external actions.

The interaction abstraction algorithm[49] is a clever algorithm that removes redundant information from the model, while preserving all information necessary to generate the test case. It differentiates between 'distinguished' and 'non-distinguished' actions. For test generation, actions visible to the environment are distinguished, but an interaction between components is not distinguished. For interactions that are not distinguished, it preserves only the 'effect' of the interaction with the rest of the system.

The algorithm is clever in that even if two interactions do not move the other component into the same state, but equivalent states, then the two interactions can still be merged. However, it is not possible to know which states in one component are equivalent until the equivalence of states has been determined in the components it interacts with, and vice versa. Hence, the algorithm must proceed iteratively: it optimistically merges interactions initially, but when it processes the other components, it may find it made a mistake. Then it re-processes the component. The algorithm ends when it finds no more mistakes. The algorithm is guaranteed to terminate.

Since the algorithm reduces each process individually in an iterative computation, it avoids problems with state explosion. It is very effective when only a subset of the system's actions is of interest—in particular, when combined with incremental test generation.

Experiments show that it reduces state spaces dramatically in the cases when only a small subset of the system's actions is of interest. In particular, it is useful for generating test cases from design models using the incremental algorithm.

4.4.1 Effect of interactions

To get an intuition for how the algorithm works, consider the composition of two systems, *S* || *T*, in Figure 4-1. The goal is to find reduced versions, [*S*] and [*T*], so that (*S* || *T*)<a_1, a_2, a_3> ~ ([*S*] || [*T*])< a_1, a_2, a_3>.

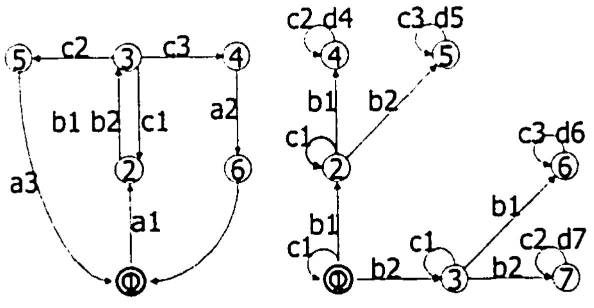

Figure 4-1 Example system with two LTS.

The component on the left can be thought of as a simple model of a telephone, and the component on the right can be thought of as a simple model of a phone directory. Thus, the telephone can go offhook (*a*1), then a '1' is dialled (*b*1), then the number is found to be incomplete (*c*1), in which case, a '2' is dialled (*b*2), and it is found to be a complete, and valid number (*c*3). The telephone then connects to the other phone (*a*2), and so on. On the other hand, dialling a second '1' would result in an invalid number (*c*2), and the telephone would give a busy tone (*a*3).

Suppose we are interested only in the actions offhook, connect, or busy tone (*a*1, *a*2, *a*3). In particular, we are not interested in which numbers are dialled (*b*1, *b*2), nor the internal interactions between the components (*c*1, *c*2). Intuitively, the directory model can be reduced to just three states: from the initial state, it can move to a state with a valid phone number, or an invalid one. The telephone model can be reduced to move from state 2 directly to 5 or 4.

The idea is to achieve the reduction is to record the effect of the interactions of the components, rather than the actual labels. As a first attempt, we can use this idea directly, and relabel the directory model as in Figure 2. For example, the transition 1—*b*1→2 is relabelled by the effect of the interaction on the phone model. The phone model makes the transition 2—*b*1→3, so the directory model gets the transition 1—23→2.

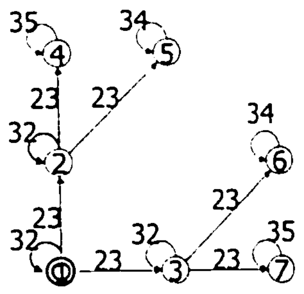

Figure 4-2 LTS component relabelled with effect of interactions.

This relabelling allows the merging of states {4,7}, {2,3} and {5,6}. However, the reduced graph is still unsatisfactory in that the merged state {2,3} is distinguished from state {1}. That means the model tracks how many numbers must be dialled to get a complete number. However, from the point of view of observational equivalence, it does not matter how many internal steps occur between externally visible steps.

The approach to obtain full reduction is to label the model with the transitive closure of the effects of individual interactions. Part of the model labelled with the transitive closure is shown in the left part of Figure 3. The labelling shows the source and destination of the other component after a sequence of internal interactions. An extra transition between states 1 and 4 has been added.

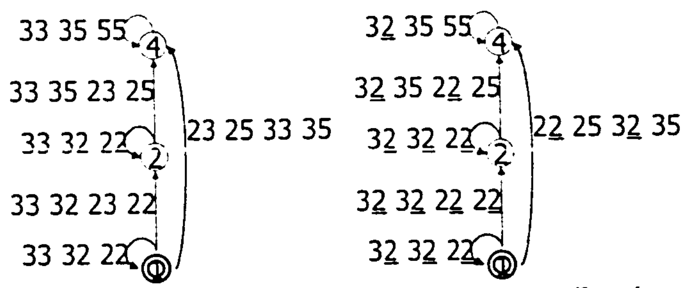

Figure 4-3 Part of component relabelled with transitive effect of interactions.

Unfortunately, the model still cannot be further reduced, as state 1 has a transition —23→ that leads to state 2, while state 2 does not have a transition —23→ that leads to an equivalent state. The problem is the phone model's states 2 and 3 are distinct. But they do not need to be, as they do not result in different external behaviour (as internal interactions are not observable). If states 2 and 3 of the phone model can be merged (labelled as '2̲'), then the portion of the directory model becomes the right side of Figure 3, where it can be seen that states 2 and 1 can be merged.

The idea is to track whether interactions cause the other model to move to equivalent states, rather than just the same states. Unfortunately, which states can be considered equivalent in the other model also depends on which states can be considered equivalent in this model, and vice versa. Thus, the equivalence reduction needs to be computed iteratively.

At the end, the interaction labels in the two reduced models must be matched in order to allow the models to compose.

4.4.2 Algorithm

The steps of the interaction abstraction algorithm for one component are as follows:

1. Calculate transitive effect of interactions:

- Store tuple (p, q, p', q') iff whenever the state (p,q) is reachable, there is a transition $(p,q)\overset{\tau}{\Longrightarrow}(p',q')$ in $(S \mid\mid T)<L>$

2. For a given classification $[T]_0$ of T, relabel S with assumed equivalent effects, to obtain S_1:
   - Remove all transitions with labels not in $L\cup\{\tau\}$
   - Add a transition $p\xrightarrow{[q]_0}p'$ for each tuple (p, q, p', q')
3. Classify S_1 to obtain $[S]_1$:
   - $[p_1]_1=[p_2]_1$ iff $p_1\sim p_2$ in S_1

The iteration for two components is as follows:

4. Repeat steps 2 and 3 until $[S]_k$ and $[T]_k$ are the same as $[S]_{k-1}$ and $[T]_{k-1}$:
   - Update $[T]_k$ using $[S]_{k-1}$ if changed, and vice versa
5. Finally, label interactions in $[S]=[S]_k$ and $[T]=[T]_k$:
   - Remove transitions with labels not in L
   - For each tuple (p, q, p', q') add transitions $[p]\xrightarrow{[p][q][p'][q']}[p']$ in $[S]$, and $[q]\xrightarrow{[p][q][p'][q']}[q']$ in $[T]$

After step 5, $[S]$ and $[T]$ can be composed using the "tuple labels".

4.4.3 Proof of correctness

We want to prove that, at the end of the algorithm, $(S \mid\mid T)<L> \sim ([S] \mid\mid [T])<L>$ by proving that for all (p,q) reachable, $(p,q) \sim ([p],[q])$. And since the initial state (p_0,q_0) is reachable, then the two compositions are equivalent. First, we prove some properties of the algorithm.

LEMMA 1: If $[p_1]_k=[p_2]_k$ then all of the following are true:

a) for all transitions $p_1\overset{\tau}{\Longrightarrow}p_1'$ there exists a transition such that $p_2\overset{\tau}{\Longrightarrow}p_2'$ where $[p_2']_k=[p_1']_k$

b) for all transitions $p_2\overset{\tau}{\Longrightarrow}p_2'$ there exists a transition such that $p_1\overset{\tau}{\Longrightarrow}p_1'$ where $[p_2']_k=[p_1']_k$

c) for all tuples (p_1, q, p_1', q_1), there exists a tuple (p_2, q, p_2', q_2) where $[p_2]_k=[p_1]_k$ and $[q_2]_k=[q_1]_k$

d) for all tuples (p_2, q, p_2', q_2), there exists a tuple (p_1, q, p_1', q_1) where $[p_1]_k=[p_2]_k$ and $[q_1]_k=[q_2]_k$

PROOF:

$[p_1]_k=[p_2]_k$

$\Rightarrow p_1 \sim p_2$ in S_k, by definition of $[S]_k$

$\Rightarrow$ for all transitions $p_1 {=}a{\Rightarrow} p_1'$ there exists a transition $p_2 {=}a{\Rightarrow} p_2'$ where $p_2' \sim p_1'$ in S_k, by corollary to the definition of ~

$\Rightarrow$ for all transitions $p_1 {=}\tau{\Rightarrow} p_1'$ there exists a transition $p_2 {=}\tau{\Rightarrow} p_2'$ where $[q_2]_{k+1}=[q_1]_{k+1}$, by definition of $[T]_{k+1}$, by which follows cases (a) and (b).

For cases (c) and (d), a tuple (p_1, q, p_1', q_1)

$\Rightarrow p_1 {-}q[q_1]_{k+1}{\rightarrow} p_1'$

$\Rightarrow$ there exists a transition $p_2 {=}q[q_1]_{k+1}{\Rightarrow} p_2'$ where $[p_2]_{k+1}=[p_1]_{k+1}$

Since tuples contain transitive effects (Step 1), there exists a tuple (p_2, q, p_2', q_2) where $[q_2]_{k+1}=[q_1]_{k+1}$.

■

LEMMA 2:

a) If there exists a transition $(p,q){=}\tau{\Rightarrow}(p',q')$ in $(S\ //\ T)<L>$ then there exists a transition $([p],[q]){=}\tau{\Rightarrow}([p'],[q'])$ in $([S]\ //\ [T])<L>$

b) If there exists a transition $p{=}a{\Rightarrow}p'$ in S, where $a \notin L$, then there exists a transition $[p]{=}a{\Rightarrow}[p']$ in $[S]$

PROOF:

Proof of (a):

$(p,q) \overset{\tau}{\Longrightarrow} (p',q')$ in $(S \,//\, T)<L>$

$\Rightarrow$ exist tuples (p, q, p', q') by Step 1

$\Rightarrow$ exist transitions $[p] \xrightarrow{[p][q][p'][q']} [p']$ in $[S]$, and $[q] \xrightarrow{[p][q][p'][q']} [q']$ in $[T]$ by Step 5

$\Rightarrow$ $([p],[q]) \overset{\tau}{\Longrightarrow} ([p'],[q'])$ in $([S] \,//\, [T])<L>$ by composition.

Proof of (b): True since the quotient keeps transitions of elements of a class, and the algorithm does not relabel transitions with label a in L.

■

LEMMA 3 (inverse of Lemma 2):

a) If there exists a transition $([p],[q]) \overset{\tau}{\Longrightarrow} ([p'],[q'])$ in $([S] \,//\, [T])<L>$ and (p,q) is reachable, then there exists a transition $(p,q) \overset{\tau}{\Longrightarrow} (p_1',q_1')$ in $(S \,//\, T)<L>$ where $[p_1']=[p']$, $[q_1']=[q']$.

b) If there exists a transition $[p] \overset{a}{\Longrightarrow} [p']$ in $[S]$, where $a \notin L$, then there exists a transition $p \overset{a}{\Longrightarrow} p_1'$ in S where $[p_1']=[p']$.

PROOF:

Proof of (a):

$([p],[q]) \overset{\tau}{\Longrightarrow} ([p'],[q'])$

Since the only interactions are through the "interaction labels",

$\Rightarrow$ exist transitions $[p] \overset{[p][q][p'][q']}{\Longrightarrow} [p']$ in $[S]$ and $[q] \overset{[p][q][p'][q']}{\Longrightarrow} [q']$ in $[T]$

Since $[S]_k=[S]_{k-1}$ and $[T]_k=[T]_{k-1}$,

$\Rightarrow$ exist transitions $[p]{=}\tau{\Rightarrow}[p_2]—[p][q][p'][q']\rightarrow[p_3]{=}\tau{\Rightarrow}[p']$ in $[S]$, and $[q]{=}\tau{\Rightarrow}[q_2]—[p][q][p'][q']\rightarrow[q_3]{=}\tau{\Rightarrow}[q']$ in $[T]$, and $[p_2]=[p]$, $[q_2]=[q]$, $[p_3]=[p']$, $[q_3]=[q']$ by Step 5

$\Rightarrow$ exist transitions $p_1—q_1[q']_k\rightarrow p_1'$ in S_k and $q_1—p_1[p']_k\rightarrow q_1'$ in T_k where $[p_2]_k=[p_1]_k=[p]_k$, $[q_2]_k=[q]_k$, $[p_3]_k=[p']_k$, $[q_3]_k=[q_1]_k=[q']_k$

$\Rightarrow$ exist tuples (p_1, q_1, p_1', q_1'), where $[q_1]_k=[q]_k$, $[p_1]_k=[p]_k$

Since $[p_1]_k=[p]_k$ and $[q_1]_k=[q]_k$ then by Lemma 1(c) and (d),

$\Rightarrow$ exist tuples (p, q_1, p_1', q_1'), where $[q_1]_k=[q]_k$

$\Rightarrow$ exist tuples (p, q, p_1', q_1')

$\Rightarrow$ if (p,q) is reachable, then there exists a transition $(p,q){=}\tau{\Rightarrow}(p_1',q_1')$ in $(S\ //\ T)\langle L\rangle$ where $[q_1']=[q']$ and $[p_1']=[p']$

Proof of (b): True by definition of quotient and by Lemma 1(a) and (b).

■

THEOREM: If (p,q) reachable in $(S\ //\ T)\langle L\rangle$, then $(p,q) \sim ([p],[q])$ in $([S]\ //\ [T])\langle L\rangle$.

PROOF:

The proof is a simple application of the definitions. It only uses the properties of the algorithm given in Lemmas 2 and 3.

Obviously for all (p,q) reachable, $((p,q), ([p],[q]))$ in R_0.

Assume for all (p,q) reachable, $((p,q), ([p],[q]))$ in R_{k-1}.

$(p,q) \xRightarrow{a} (p',q)$ where a in L

$\Rightarrow (p,q) \xRightarrow{\tau} (p_1',p_1) \xrightarrow{a} (p_2',q_2) \xRightarrow{\tau} (p',q)$ by definition of $\xRightarrow{a}$

$\Rightarrow ([p],[q]) \xRightarrow{\tau} ([p_1],[p_1]) \xRightarrow{a} ([p_2],[q_2]) \xRightarrow{\tau} ([p'],[q'])$ by lemma 1 (a) and (b)

$\Rightarrow ([p],[q]) \xRightarrow{a} ([p'],[q'])$ by definition of $\xRightarrow{a}$

And by induction hypothesis, ((p',q), $([p'],[q'])$) in R_{k-1}

For the other direction:

$([p],[q]) \xRightarrow{a} ([p'],[q'])$

$\Rightarrow ([p],[q]) \xRightarrow{\tau} ([p_1],[p_1]) \xrightarrow{a} ([p_2],[q_2]) \xRightarrow{\tau} ([p'],[q'])$ by definition of $\xRightarrow{a}$

$\Rightarrow$ $(p,q) \xRightarrow{\tau} (p_3',q_3) \xRightarrow{a} (p_4',q_4) \xRightarrow{\tau} (p_5',q_5)$ where $[p_3]=[p_1]$, $[q_3]=[p_1]$, $[p_4]=[p_2]$, $[q_4]=[q_2]$, $[p_5]=[p']$, and $[q_5]=[q']$ by lemma 2 (a) and (b)

$\Rightarrow (p,q) \xRightarrow{a} (p_5',q_5)$ where $[p_5]=[p']$, and $[q_5]=[q']$

And by induction hypothesis, ((p_5',q_5), $([p'],[q'])$) in R_{k-1}.

Thus, ((p,q), $([p],[q])$) in R_k

■

Next, it is necessary to show the algorithm always terminates, which can be done using the following lemma.

LEMMA: Let the classification $[T]_k$ be a refinement of the classification $[T]_{k-1}$. Then $[S]_{k+1}$ computed using $[T]_k$ is a refinement of $[S]_k$ computed using the labelling $[T]_{k-1}$.

PROOF:

Suppose $[p_1]_k=[p_2]_k$. We want to show that $[p_1]_{k+1}=[p_2]_{k+1}$, or if $p_1 \sim p_2$ in S_1, then $p_1 \sim p_2$ in S_2.

Since $[T_1]$ is a refinement of $[T_2]$, if labels $q_1[q_1]_1 = q_1[q_2]_1$, then labels $q_1[q_1]_2 = q_1[q_2]_2$. Thus, if two transitions in S_1 have the same labels using the labelling $=q_1[q_1]_1\Rightarrow$, the transitions in S_2 will still have the same labels using the labelling $=q_1[q_2]_2\Rightarrow$.

Supose $p_1 \sim p_2$ in S_1

$\Leftrightarrow$ for any transition $p_1 =u_1\Rightarrow p_1'$, there is a corresponding transition $p_2 =u_1\Rightarrow p_2'$ and $p_1' \sim p_2'$ in S_1

$\Rightarrow$ for any transition $p_1 =u_1\Rightarrow p_1'$, there is a corresponding transition $p_2 =u_1\Rightarrow p_2'$ and $p_1' \sim p_2'$ in S_2, since the labels of the two transitions are guaranteed to be the same in S_2.

$\Leftrightarrow p_1' \sim p_2'$ in S_2

■

The lemma shows that the algorithm is monotonic, that is, each iteration computes a refinement of the classification of the previous iteration. Since the number of refinements is finite, the number of iterations is finite and the algorithm must terminate.

4.4.4 Multiple components, and multi-way interactions

It has been shown how to compute the reduction for two components. To handle multiple components with 2-way interactions, we collect and label interactions between pair of components separately.

For Step 1 of the algorithm, we can simply store S_iS_j:(p_i, p_j, p_i', p_j') if there is a transition (p_i, p_j) $=\tau\Rightarrow$ (p_i', p_j') in $(S_i \,||\, S_j)$<L>. This satisfies the condition that S_iS_j:(p_i, p_j, p_i', p_j') is stored iff whenever the state $(\ldots p_i, \ldots, p_j, \ldots)$ is reachable, there is a transition $(\ldots p_i, \ldots, p_j, \ldots)$ $=\tau\Rightarrow$ $(\ldots p_i', \ldots, p_j', \ldots)$ in $(S_1 \,||\, \ldots \,||\, S_n)$<$L$>.

For Step 2, for given classifications $[S_j]_0$ of S_j, use the following conditions to compute $[S_i]_1$:

a) for all tuples S_iS_j:(p_1, q_1, p_1', q_1'), and for all $[q_2]_0=[q_1]_0$, there exists a tuple S_iS_j:(p_2, q_2, p_2', q_2'), where $[p_2']_1=[p_1']_1$, $[q_2']_0=[q_1']_0$ and

b) for all tuples S_iS_j:(p_2, q_2, p_2', q_2'), and for all $[q_2]_0=[q_1]_0$, there exists a tuple S_iS_j:(p_1, q_1, p_1', q_1'), where $[p_2']_1=[p_1']_1$, $[q_2']_0=[q_1']_0$

For the iteration step 2, update $[S_i]_k$ if any $[S_j]_{k-1}$ with which it interacts has changed. For step 3, add transitions $[p_i]$—S_iS_j:$[p_i][p_j][p_i'][p_j']\rightarrow[p_i']$ for all tuples S_iS_j:(p_i, p_j, p_i', p_j') in $[S_i]$ and add transitions $[p_j]$—S_iS_j:$[p_i][p_j][p_i'][p_j']\rightarrow[p_j']$ for all tuples S_iS_j:(p_i, p_j, p_i', p_j') in $[S_j]$.

Lemma 1 is changed to

c) If there exists a transition $(\ldots p_i,\ldots, p_j,\ldots) \overset{\tau}{\Longrightarrow} (\ldots p_i',\ldots, p_j',\ldots)$ in $(S_1 \,//\, \ldots \,//\, S_n)\langle L\rangle$ then there exists a transition $(\ldots[p_i],\ldots, [p_j],\ldots) \overset{\tau}{\Longrightarrow} (\ldots[p_i'],\ldots, [p_j'],\ldots)$ in $([S_1] \,//\, \ldots \,//\, [S_n])\langle L\rangle$

d) If there exists a transition $p_i \overset{a}{\Longrightarrow} p_i'$ in S_i, where $a \notin L$, then there exists a transition $[p_i] \overset{a}{\Longrightarrow} [p_i']$ in $[S_i]$

Lemma 2 and the proofs are all changed similarly: p, q is replaced with $\ldots p_i,\ldots, p_j,\ldots$ and $S \,//\, T$ is replaced with $S_1 \,//\, \ldots \,//\, S_n$

The proof of the theorem is changed so that a τ-transition $(p_1,\ldots, p_n) \overset{\tau}{\Longrightarrow} (p_1',\ldots, p_n')$ in $(S_1 \,//\, \ldots \,//\, S_n)\langle L\rangle$ must be broken down into a sequence of constituent τ-transitions with pair-wise interactions, such as $(\ldots p_i,\ldots, p_j,\ldots) \overset{\tau}{\Longrightarrow} (\ldots p_i',\ldots, p_j',\ldots)$. Then, the lemmas are applied to each constituent τ-transition.

For multi-way interactions, interactions for each subset of interacting components is collected and labelled separately. For example, for a 3-way interaction, the stored vectors are $S_1S_2S_3(p_1, p_2, p_3, p_1', p_2', p_3')$ if there is a transition $(p_1, p_2, p_3) \overset{\tau}{\Longrightarrow} (p_1', p_2', p_3')$ in $(S_1 \,//\, S_2 \,//\, S_3 \,//\, \ldots \,//\, S_n)\langle L\rangle$. The conditions, the other steps, and the proof proceed similarly.

4.4.5 Algorithm complexity

An upper bound for the algorithm complexity can be obtained by adding up the cost of basic operations.

Since interactions with each pair of components are collected separately, reductions for a component can be computed without composing the rest of the system. The number of stored interactions for a component S_i (for 2-way interactions) is $S_jS_i(p_i, p_j, p_i', p_j')$. Thus, the maximum number of stored interactions is the number of interacting components, S_j, times the fourth power of the number of states of the components (p_i, p_j, p_i', p_j'). Thus, the number of interactions of S_i is at worst nm^4 for a system of n components, all with m states. For multi-way interactions, the number of interactions is at worst nm^{2k} if there are at most k-way interactions.

Minimization of each component by observational equivalence can be performed in $O(ne)$ *time*, where e is the number of transitions of the relabelled components. The number of transitions is the number of interactions plus the number of visible transitions. At worst, this is $lm^2 + nm^4$, where l is the number of externally visible labels. Assuming l is unrelated to m and n, then the number of transitions is $O(nm^4)$. Thus, the minimization has complexity $O(n^2m^4)$. (For k-way interactions, the minimization has complexity $O(n^2m^{2k})$.)

During one iteration, at most n minimizations are required. The number of iterations is at most the sum of the states of the components, because the size of one reduced component must increase, or else the algorithm terminates. Thus, the number of iterations is at most $O(n^2m)$, and the overall complexity is $O(n^4m^5)$. (For k-way interactions, the complexity is $O(n^4m^{2k+1})$.)

4.4.6 Algorithm notes

Collecting interactions with each pair of components separately means reductions can be computed without composing the rest of the system. However, the disadvantage is that interactions with different components are being distinguished from each other, thus lessening the amount of reduction possible. For example, interactions of S_i with two different components would still be distinguished even if the interactions do not change the states of either of the other components. In particular, a system with *all* labels hidden would not reduce to a set of 1-state abstractions!

Collecting interactions with each pair of components still results in adding many transitions in the relabelled components. In many cases, the same edges in S (labelled $p_1[p_2]$) may cause many different edges in T (labelled $q_1[q_2]$). But, for the purpose of reducing the components, many of the labels $q_1[q_2]$ are redundant. Two labels $q_1[q_2]$ are redundant if they always appear together in all transitions labelled $p_1[p_2]$, since they can never be used to distinguish any states in S.

Formally, it is safe to merge the labels a and b if, whenever there is a transition labelled with $p_1 \overset{a}{\Longrightarrow} p_2$, then there is also a transition labelled with $p_1 \overset{b}{\Longrightarrow} p_2$, and vice versa. In addition, duplicate transitions (same source, label, and destination) can then be removed.

Experiments show this optimization greatly reduces the number of transitions, and significantly speeds up the minimization of components.

4.5 Related work

This section compares incremental test case generation and interaction abstraction with previous reduction techniques, especially their ability to exploit characteristics of test generation from models of software components.

4.5.1 Incremental test generation

There are no techniques similar to the incremental test case generation algorithm, although some work has been done to create algorithms for test case generation.

Goal-oriented execution for LOTOS[31] specifications can be used to efficiently generate test cases. They rely on static analysis of the specification source code to prune away any expansion of the state space that is guaranteed not to lead to states of interest. This technique is useful when the states of interest are isolated in small parts of the specification source code. In that case, much of the other behaviour can be essentially 'sliced' away, using the special expansion rules.

Two other studies applied existing reduction techniques to test generation. One study modified the compositional model generation algorithm to preserve information needed for deterministic testing. The algorithm was applied to the simple examples in Carver and Tai's study[13]. The other study adapted partial-order methods for efficient bounded search of SDL[65].

None of the algorithms considered generating test cases incrementally. Incremental generation is important because the fewer behaviours of a model that are needed, the greatei the reduction that can be obtained.

It should be noted that a practical test generation tool does more than generate a sequence of inputs and outputs. For example, it needs to handle non-deterministic output. Existing techniques for generating test cases should be incorporated into the algorithm by replacing the path generator in Step 2 of the algorithm. Instead, a tool would generate a portion of a useful test case as an LTS that handles inputs and outputs. That LTS of the portion of the test case is then used in the rest of algorithm.

4.5.2 Interaction abstraction

Interaction abstraction is effective for models of software components, which have the following characteristics:

- the system is made up of many loosely-coupled components; each component communicates with a limited number of other components
- each component is small enough to be generated and minimized, but compositions of components are too large
- large components are the result of handling many loosely-related aspects, and many connections with other components
- only a small part of the system's behaviour is of interest at one time.

Note that interaction abstraction preserves observational equivalence for all externally visible behaviour. In particular, liveness properties are also preserved. Thus the algorithm is suitable for use in general verification tasks, and not just for test generation. The algorithm can yield useful reductions for verification if the number of undistinguished interactions is large, and the number of distinguished interactions is small.

Interaction abstraction is able to effectively reduce a wider range of useful systems than other techniques, which depend on very restrictive forms of models. For example, BDD-based reduction can be very effective in case of systems with a linear topology, but it is ineffective in systems with a net topology where many parts of the system interact with each other.

Interaction abstraction is similar to other techniques that require a specific type of redundancy, such as data-independence or symmetry. It requires models to have unrelated or loosely related aspects, but only one aspect is of interest at a time. This kind of redundancy is likely to exist in general-purpose models of a system, rather than models specifically designed to verify a particular property, such as suggested by Holzmann[37]. In addition, the algorithm is able to work together with other algorithms to handle models with many kinds of redundancy. It is expected to work particularly well with commutativity-based methods (Section 2.5.1.4), since interaction reduction results in a large number of small components.

Interaction abstraction is able to reduce global state spaces exponentially, whereas other general reduction techniques, such as on-the-fly search or supertrace (Sections 2.5.1 and 2.5.1.2), can only offer linear reductions in size.

The techniques most similar to interaction abstraction are compositional minimization algorithms (Section 2.5.1.3). In contrast to those algorithms, interaction abstraction uses many contexts to reduce each component; it does not need to compose contexts, nor user-created contexts.

Interaction abstraction also differs from compositional minimization in that it does not require composing components to obtain reduction. Instead, it reduces each process by itself using information about interactions with other processing. Thus, it avoids the state explosion problem altogether.

The goal of interaction abstraction often differs from compositional minimization, which typically tries to construct a minimal global model on which many different properties can be evaluated. Interaction abstraction is much more effective when creating a different reduced model for each property of interest. Construction of different reduced models is cost-effective since each specific reduced model is, in general, exponentially smaller than a general model.

4.6 Chapter conclusions

Two new algorithms are presented that aim to efficiently generate test cases from formal state space models of components. The first incrementally builds a complete test case by generating partial test cases for subsets of inputs and outputs. By reducing the number of behaviours of interest at one time, the size of the models can be reduced greatly. The other algorithm reduces sizes of components by abstracting away behaviour that is not of interest. It is proved that it preserves behaviour of interest, but abstracts uninteresting interactions, preserving only their effect on interesting behaviour. The reduction algorithm does not suffer from state-explosion as it does not compose components, but iteratively reduces each component individually.

Chapter 5

IMPLEMENTATION AND EVALUATION

5.1 Chapter Overview

This chapter evaluates the feasibility of generating test cases from formal design models. The method is feasible if a formal design model can be created with not much more effort than an informal model, and the algorithms to analyze the models can handle realistic components in a reasonable amount of time.

The modelling of a realistic software design in ObjectState is evaluated. The model was found to be easy to understand and it was reasonably easy to create compared to an informal model—as simple as writing a short program.

Next, the performance of analysis algorithms on this model is examined. The example model is found to be too large for standard algorithms to handle, but using the incremental and abstraction algorithms, useful test cases can be generated from portions of the model. The new algorithms are shown to be very effective in situations of state explosion where previous reduction methods are ineffective.

5.2 Tool implementation

5.2.1 ObjectState Translator

The translator tool converts ObjectState model and coverage criteria into their LTS semantics. The translator steps are shown in Figure 5-1: it expands the ObjectState model into its graph of steps, then translates the steps into Murphi code, and finally the Murphi code generates LTS transitions from each step.

The language Perl was used to greatly simplify the task of creating the translator. Only 1300 lines of Perl were required (Table 5-1). The flexibility of Perl allowed ObjectState models to be written directly as Perl code. The statements in ObjectState, such as PROCTYPE or WAIT, call Perl functions to generate the graph of the steps of the model, and eventually generate the Murphi code. In effect, the Perl compiler acts as the parser for the ObjectState program.

Each component is translated into a separate Murphi program. The component's variables are translated into Murphi variables and each step is translated into a Murphi rule. The start-rule sets the initial values of the variables and initial state. Executing the Murphi code expands the steps of the model with all reachable data values. To generate an LTS, the standard Murphi compiler was modified to use the BCG library, which is part of the Caesar-Aldebaran Development Package (CADP)[24] toolkit. The BCG library allows reading and writing of LTS models in Binary Coded Graph (BCG) format.

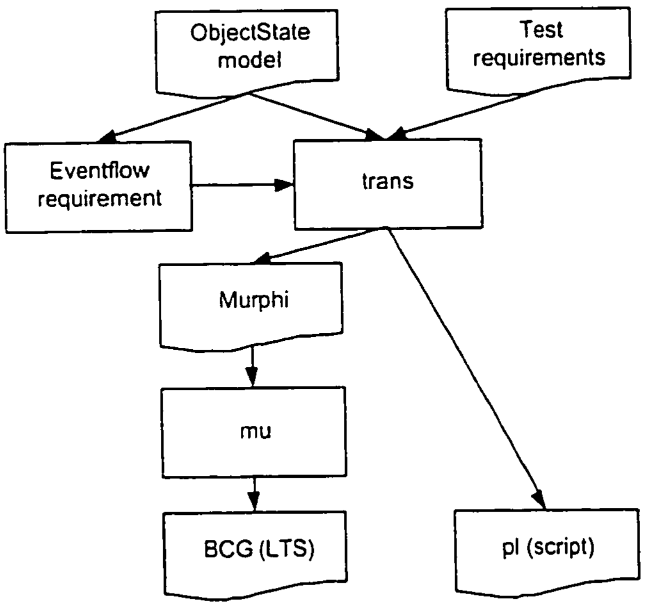

Figure 5-1 Structure of translator tool.

Test requirements are treated as regular components. In the current prototype, the translator does not support multi-way synchronization. Thus, the TESTCONN declaration is not supported, and a test requirement must CONNECT to a component's free ports.

The event-flow coverage generator creates test requirements that guarantee one related path for each related pair. Related events are defined as the interactions at a given port. The implementation of the coverage generator realizes another benefit of using Perl to represent ObjectState models: that is, the full power of Perl is available to automate the creation of ObjestState models, such as test requirements.

File	Lines
Config.class	27
Config.pm	107
Genconfig.prog	133
Genmodel.pm	255
Parsemodel.pm	723
Wrapreq.pm	68
Total	1313

Table 5-1 Lines of code for translator.

In addition, a second translator exists for Promela-annotated ObjectState models. The model is expanded into steps in the same way, but then the steps are translated into Promela code. Note that the Promela model would generate a different, but similar, transition system than a corresponding Murphi model, due to the differences in the languages and semantics. This difference impacts the comparison of performance of analysis.

5.2.2 State Space Analyzer

The analyzer abstracts the model and generates a path that satisfies the test requirement. The three components of the analyzer are basic operations, abstraction, and path generation.

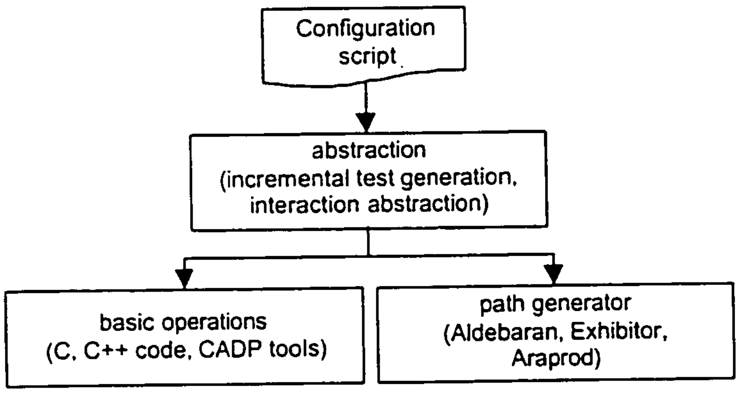
Configuration script
abstraction (incremental test generation, interaction abstraction)
basic operations (C, C++ code, CADP tools)
path generator (Aldebaran, Exhibitor, Araprod)

Figure 5-2 Components of the analyzer tool.

Basic operations manipulate LTS models in BCG format, such as minimization, composition, and relabelling. Most of the functionality is provided by the CADP toolbox[24]. The CADP

File	Lines
dynbitvec.h	30
maps.h	83
transitive.C	393
weakreduct.c	136
maps.C	229
merger.C	38
transops.C	533
transops.i	62
analysis.pl	701
config.pl	620
toara.pl	55
toprom.pl	34
Total	2914

Table 5-2 Lines of code of analyzer

toolbox is very useful because it uses LTS as a very general and powerful input/output format. Many other tools, such as SPIN, use a very tool-specific modelling language, which would not have been appropriate to implement the analyzer. A convenient Perl interface is provided for the basic operations.

However, the CADP tools do not provide manipulations transitive closure of interactions, and reduction of interaction labelled transitions (Section 4.4.6). Since there are potentially a large number of interactions, the interactions were manipulated as BDDs, using the BuDDy package[48].

The abstractor implements both the incremental test generation algorithm and the interaction abstraction algorithm. The abstractor is implemented in Perl, and uses the basic operations to transform the LTS.

The path generator performs model-checking on the composition of components, to generate test sequences. The analyzer interfaces with two model-checkers:

- Exhibitor, part of the CADP toolbox, using simple on-the-fly breadth-first search
- Araprod implements on-the-fly breadth-first search with partial-order reduction

In addition, the SPIN model-checker is used to analyze the version of ObjectState that uses Promela.

5.3 Example software model

A software design was modelled using ObjectState. The software manages the call processing of a private branch exchange (PBX). The call processing software is responsible for monitoring the state of the PBX hardware for changes in the state, such as phones going off-hook and on-hook, and phone digits being dialled. The software responds to changes in state by causing appropriate changes in the hardware state, such as connecting a phone to a dialtone, or connecting the ring generator to a phone. The hardware state is detected and modified through a memory-mapped interface.

This example was chosen because it represents a realistic, complex software design. It has a fair number of components that interact in complex ways. This software example has been used as a course project[19] for an undergraduate software engineering course by the Electrical and Computer Engineering Department at the University of Waterloo. A group of four students is expected to design and implement the system over a four-month term. On the other hand, the example is small enough to be modelled completely, and in detail.

5.3.1 Design structure

The model of the PBX is based on the design of an actual implementation. The design comprises the following processes:

- Linescan (LS): receives requests from other processes to monitor the switchhook status of phones. It polls the phone hardware and notifies the process when the switchhook status has changed.
- Call manager (CM): waits for notification of a phone going off-hook, then assigns an available call handler process to handle the interactions for making a call. Also responds to queries about whether a phone is currently involved in a call.
- Database (DB): stores the directory numbers of phones, as well as the in-service status.

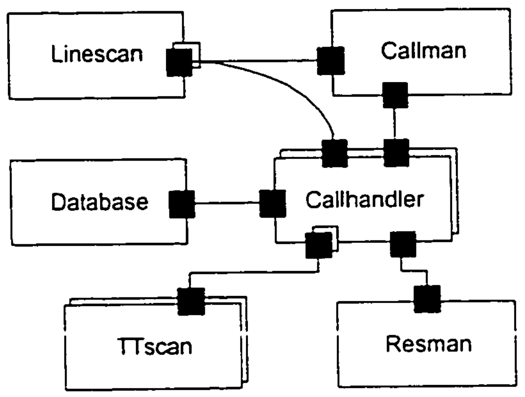

Figure 5-3 Configuration of the components of PBX software

- Resource manager (RM): allocates available channels and touchtone receivers to requesting processes.
- Multiple touchtone scanners (TS): poll the touchtone receiver hardware state and notifies the process that requested information about digits dialled.
- Multiple call handlers (CH): interact with telephone users by telling the PBX hardware state to set up connections between phones and tone generators, touch-tone receivers, etc.

The ObjectState source for the models is given in Appendix A.1-A.6.

Module	Lines
Callhandler.pm	301
Callmanager.pm	101
Connectreq.pm	26
Database.pm	42
Linescan.pm	85
Resource_manager.pm	65
Datatypes.pm	219
Porttypes.pm	100
Total	939

Table 5-3 Lines of ObjectState source for PBX model components

5.3.2 Modelling effort

The model of the PBX was found to be quite easy to create and understand. Although it was certainly more difficult to create than an informal model, it was similar to writing a small program. The formal model was 1000 lines of ObjectState (Table 5-3). In comparison, a detailed, informal model in SDL runs about 20 pages. If each SDL symbol can be compared to a line of code, then the informal model is between a third to a half of the size of the formal model. The implementation is 16000 lines of C.

Most of the complexity of the formal model was in the code for maintaining data structures like sets and tables. The code for interactions (sending/receiving messages) and for transiting between states was quite simple and direct. The formal model could be greatly improved with support from libraries of data types and a graphical notation. Table 5-3 shows about 20% of the code is in Datatypes.pm, which implements set and table data types and basic algorithms. It is reasonable to expect a quarter of the code can be eliminated with better data type support and a graphical notation. Then, the formal model would be close to the complexity of an informal model. Note that the key criteria for library code would be ease of use, and not efficiency, as the model is only used for analysis, not code generation.

A great advantage of the formal model is that it is executable, and hence it is easier to correctly model complicated sequences of events. Initially, a mistake was made in creating the call handler model. By abstracting the interactions of the model and displaying the abstracted LTS graphically, the problem was easily identified and fixed. Without seeing the actual dynamic behaviour, the problem would have been very difficult to find.

The software was also modelled using Promela annotations (in order to use SPIN to analyze the model). In comparing the Murphi annotated model against the Promela annotated model, the Murphi language was found to be much easier to use, due to the ability to create procedures with local variables.

5.3.3 Model complexity

The language was found to be expressive enough to handle some tough modelling problems, such as the system's dynamic communication patterns.

Hierarchical state machines are well suited to the modelling of the PBX software. The use of entry and exit actions greatly simplified modelling of the complicated call-handling component. In particular, it was used to separate the resource management aspect from the main problem of call handling. As shown in Figure 5-4, many of the superstates deal with resource management. Entry and exit actions of these states obtained and released resources transparently for each part of the call handling sequence.

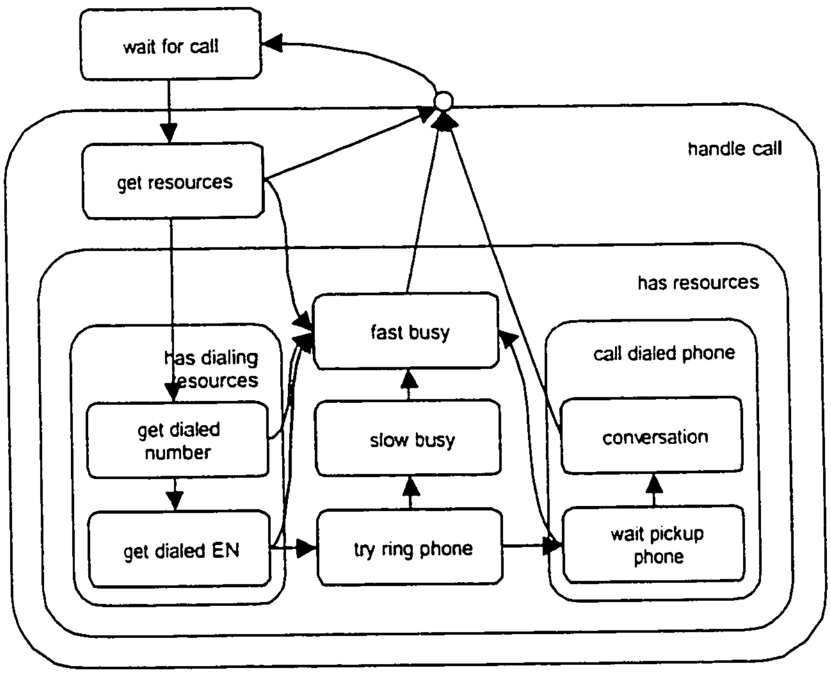

Figure 5-4 State hierarchy and connections for call handler component.

In addition, signal handlers were used to allow each part of the model to add its own code to respond to an event. For example, different actions need to be taken in response to an onhook message, depending on which state the call handler is in. In some states, timers or linescan requests have to be cancelled or redirected. Signal handlers allow states to share certain actions, while adding other actions.

An interesting feature of the model is its dynamic communication pattern. For example, depending on which touchtone receiver was allocated to a call handler, it would communicate with different touchtone scan processes to obtain digit information. Similarly, the linescan process will accept a request from one process to notify a different process of the switchhook status of a phone. In essence, processes need to pass and use references to other processes.

To model dynamic communications, a component would have a port connected to each processes with which it may need to communicate. The ID of each port would be the ID of the component it was connected to. The component IDs then could be passed as parameters in messages, and used to select components to communicate with.

The effect of the dynamic communication is to increase the number of states and transitions in LTS semantics. Each component would add one transition for each interaction at each port. The more components that it can possibly communicate with, the more transitions there are in its LTS model.

The greatest benefit of using a modelling notation, such as ObjectState, is that details of the communications between components are abstracted. Most of the C code of the implementation deals with details of inter-process communications, such as marshalling and unmarshalling parameters.

5.4 Algorithm performanc

This section investigates the performance of different algorithms under different conditions of state-explosion. Paths were generated for subsets of the PBX model using different algorithms and the time required were compared. The new algorithms performed significantly better than previous algorithms on the PBX model. While previous algorithms were able to analyze only two components of the PBX, the new algorithms were able to generate tests for useful subsets of five components. The results show the new algorithms are effective against the kind of state-explosion that occurs in the PBX model and other design models.

5.4.1 Experiment decisions

The PBX model is a good test of the feasibility of component interaction testing from formal models. It is a complete model of components and their interactions, and contains many loosely related aspects, such as resource handling, dialling, and call processing.

In comparison, example formal models used in previous studies are not good tests of the feasibility of interaction testing. Firstly, those examples model either hardware, protocols, or software specifications (see Section 2.4.2). Secondly, in order to be analyzable by tools, either they are very small models, or they are heavily redundant in one specific aspect, which can be exploited by an algorithm. As a result, they lack the complexities of components and interactions that make analysis difficult for interaction testing. In addition, they lack redundancy that results from a complete model of software that has to handle many aspects, but only some aspects are of interest for each test case.

The following model-checking tools and algorithms were compared:

- Exhibitor (simple breadth-first search),
- SPIN (depth-first, partial-order reduction, supertrace),
- Araprod (breadth-first, partial-order reduction),
- Aldebaran (minimal model generation, BDD),
- new analyzer with incremental test generation,
- new analyzer with incremental test generation, and interaction abstraction.

The parameters given for each tools are shown in Appendix A.

The tools were chosen because they implement advanced and successful model-checking algorithms. In addition, the tools should be able to handle LTS input (with the exception of SPIN) and give LTS output. SPIN cannot take LTS input, but was chosen specifically because it is a very advanced system, and can be considered a standard for tools. Also, Aldebaran's minimal model generation algorithm can only compute if a path exists, but cannot generate a path. It is not appropriate for test generation, but it was included to compare the effectiveness of the compositional minimization algorithm. All the tools are freely obtainable for research purposes.

The tests were conducted using a simple test requirement: the system must connect two phones in a conversation, two times. (The source code for the test requirement is in Appendix A.7.) This test requirement was chosen to provide a strenuous test for the algorithms as it would require a long sequence of interactions to satisfy. To connect two phones, one phone must go offhook, resources must be available to make a call, the number of another phone must be dialled, and the phone must be picked up, and so on.

The sequence of all the events that must occur is not specified in the requirement, and must be computed by the analyzer. The test requirement also does not specify which phone or phones must call and which phone or phones must answer. Any sequence that involves at least two connections will satisfy the requirement.

5.4.2 Test results

Beginning with only one component, the tools were given more components to analyze until they exceeded available memory or failed to give an answer in a reasonable amount of time.

The PBX model is much too large for the tools to handle. Even for the simplest case of finding a path for a single call handler process, the BDD and commutativity algorithms failed. SPIN failed for the interaction of a call handler with the database process. The simple breadth-first search in Exhibitor performed better than the more complex algorithms, but it eventually failed to compute a path for a call handler with the database and call manager.

	REQ	DB	CM	LS	CH1	CH2	CH3
REQ +CH	7				85956		
REQ +CH +DB	7	291			85956		
REQ +CH +DB +CM	7	291	146		85956		
REQ +2xCH +DB +CM	7	579	390		85956	85956	
REQ +3xCH +DB +CM	7	579	390		85956	85956	85956
REQ +2xCH +DB +CM +LS	7	579	390	41371	85956	85956	

Table 5-4 Sizes of models: original number of states for model components

	Searched states	Expected coverage
REQ +CH	3000	>= 99.9% on avg.
REQ +CH +DB	1,120,840	>= 99.9% on avg.

Table 5-5 Number of states searched by SPIN

With incremental test generation, an additional component can be analyzed. Adding interaction abstraction, five components can be analyzed: three call handlers, the database and the call manager. However, it also fails when the linescan process was included.

The results are shown in Table 5-6. The sizes of the models are shown in Table 5-4. Each row shows a subset of components that was analyzed. Note that the same component in different subsets has a number of states because the component has to interact with different numbers of other components, and hence requires more states. The names of the components are abbreviated as follows:

- REQ – test requirement
- CH – Call handler (3×CH means 3 call handlers, while CH1 refers to the first call handler)

	PROD	Aldebarar	SPIN	Exhibitor	Incremental test generatio	Incremental + interaction abstraction
REQ +CH	Memory out	Time out	554	790	365	3079
REQ +CH +DB	-	-	Not found	7644	534	5159
REQ +CH +DB +CM	-	-	-	Memory out	677	6061
REQ +2×CH +DB +CM	-	-	-	-	Time out	7374
REQ +3×CH +DB +CM	-	-	-	-	-	13558
REQ +2×CH +DB +CM +LS	-	-	-	-	-	Time out

Table 5-6 Times (seconds) for generating paths

- DB – Database
- CM – Call manager
- LS – Linescan

For the Promela-annotated models, Table 5-5 shows the number of states searched by SPIN. Although the Promela-annotated models were implemented as similarly as possible to the Murphi-annotated ones, Promela semantics are somewhat different, and the number of states of each component would differ a little. SPIN does not give a count of the number of states for each component.

5.4.3 Scalability of new algorithms

Such a small sample cannot give a general indication of the scalability of the new algorithms. However, there are several encouraging aspects in the experimental results. In spite of the rough nature of the prototype implementation, the analyzer tool is able to handle quite large components.

In theory, the abstraction algorithm avoids state explosion by avoiding composition of components and abstracting each component individually. Nevertheless, the theoretical complexity of $O(n^4m^5)$ looks quite daunting for practical use. However, the actual situation is much better in the experiment.

The major factor in the cost is the set of interactions between two components. This set has a theoretical size complexity of $O(m^4)$, which seems to actually occur, such as with the linescan component (see Section 5.4.4.4). It is important to realize, however, that this complexity simply results from interactions between two components. Any type of model checking that takes into account interactions between components must face at least this level of complexity.

Let us assume that individual components are small, so that the tools built are able to handle compositions of two components in a reasonable (i.e. constant) time. This assumption eliminates the four powers of m. Also at the most four iterations of reductions were required for each component in experiments. That is much better than the worst case of $O(nm)$ minimizations of each component. Further, assume that components only interact with a limited (i.e. bounded) number of other components. This eliminates one more power of n.

Thus, for well-behaved models under these assumptions, the actual complexity is proportional to $O(n)$ and the effort depends linearly on the number of components.

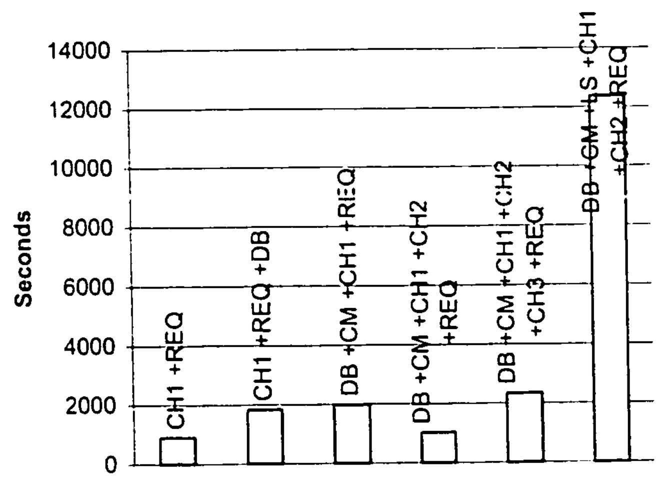

Figure 5-5 Maximum time for reductions over all external port reductions.

The maximum times for reductions are shown in Figure 5-5. The times for reduction differs depending on the external port being hidden. The figure shows that times for reduction do not depend on the number of components, but rather on the number of interactions between components. The test run shown in the rightmost bar has the same number of components as the test run for the second right-most bar. The large time requirement for reduction in the rightmost bar is caused by the linescan component, which has a huge number of interactions with other components.

Even though the reduction algorithms can avoid state-explosion, the path generation algorithms must still deal with the composition of the reduced components. Thus, the scalability of this step depends on the reductions achieved. Table 5-7 shows that impressive reductions were achieved (compare with Table 5-4). The table shows the maximum sizes of the components after reduction. The sizes after reduction differ depending on the external port being considered. The actual sizes for each step are usually much smaller than the maximum sizes.

	REQ	DB	CM	LS	CH1	CH2	CH3
REQ +CH	5				374		
REQ +CH +DB	5	3			510		
REQ +CH +DB +CM	5	4	56		759		
REQ +2xCH +DB +CM	5	7	116		135	224	
REQ +3xCH +DB +CM	5	10	258		135	149	759
REQ +2xCH +DB +CM +LS	5	7	148	861	312	256	

Table 5-7 Maximum numbers of states for components (over all external ports reductions) after reduction by interaction abstraction

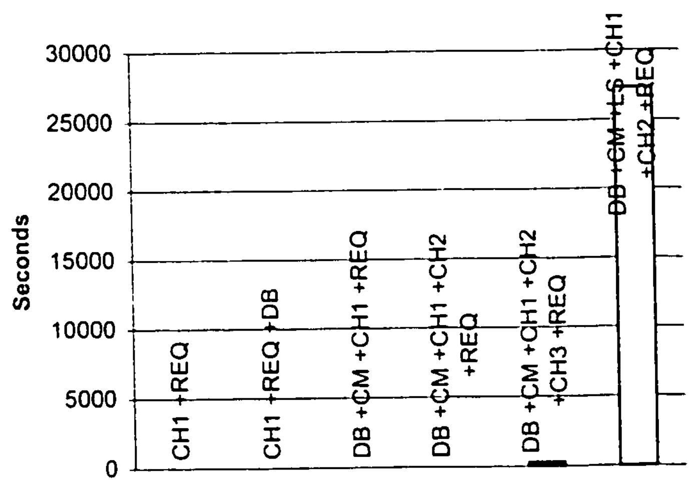

Figure 5-6 Maximum times for partial path generation over all external port reductions.

The maximum times required to generate a path is shown in Figure 5-6. The figure shows that in most of the test runs, the time to extract a partial path is kept to a few seconds. Nevertheless, as the number of states of each component grows slowly, the number of global states grows exponentially. The exponential growth can be seen in the rightmost bar, as the number of states and components is largest.

5.4.4 Analysis of state explosion

To understand the scalability of the algorithms requires understanding the factors that cause state explosion and the effect of the algorithms on these factors. This section compares the effectiveness of each reduction algorithm on different sources of state explosion and their effectiveness in exploiting redundancies in the models.

5.4.4.1 Large state spaces from expanded data structures

BDD reduction algorithm performed very poorly on the example models: it could not analyse even a single call handler component with 80000 states. This result is in stark contrast to work in hardware verification, where huge state spaces, greater than 10^{20} states, can be analyzed[12]. The difference, of course, is that the hardware models were very regular in structure, whereas the call handler model is a monolithic LTS with no apparent structure. BDD-based algorithms depend on finding a good ordering of the BDD variables, where variables far apart in the ordering should not depend on each other. The LTS model does not provide any information on how to find a good ordering. In any case, as discussed in Hu's thesis[39], high-level models such as software designs contain variables that correlate with many other variables, so that no good orderings exist.

The reason the LTS models are so large is that data structures and variables are used liberally in the models. Expanding the data causes the explosion in the LTS size.

The partial-order algorithm also does not help data complexity, since there is only one component and a test requirement and hence little interleaving of actions. The simple breadth-first search algorithm performed better simply because it did not have the overhead of attempting to compute reductions. However, it is very susceptible to state explosion with increasing numbers of components, as seen in the table.

A solution for the problem of data is proposed for verification. The approach is to try various abstract interpretations of the data structures in the model[18], until one is found that sufficiently reduces the state space, while providing an answer that is guaranteed to hold for the concrete system. The idea behind abstract interpretation is that only some properties of the data structure are relevant to the verification, rather than all the details of the data structure. The abstract interpretation only models that aspect, and throws away the information that is not relevant. If it happens that the discarded information is actually relevant to the verification, then the answer is not guaranteed to be valid.

	REQ	DB	CM	LS	CH1	CH2	CH3
REQ +CH	5				12790		
REQ +CH +DB	5	4			12390		
REQ +CH +DB +CM	5	4	125		12790		
REQ +2CH +DB +CM	5	7	223		12790	12790	
REQ +3CH +DB +CM	5	7	223		12790	12790	12790
REQ +2CH +DB +CM +LS	5	7	223	1665	12790	12790	

Table 5-8 Numbers of states after reduction using observation equivalence

Choosing abstract interpretations requires much manual effort. A similar, but automatic, method is to abstract states based on the visible actions of the component. Details of the data structures that are only internal, and invisible to other components, cannot affect the verification of properties of the system, nor test generation. This technique is used in the compositional minimization algorithm, which alternates minimization with composition of parts of the system in an attempt to avoid the state-explosion that results from composing all components at once.

The sizes of components reduced by observational equivalence are shown in Table 5-8 (compare with Table 5-4). A particular benefit of minimization by observational equivalence is that remnant variable values are automatically removed (see Section 2.5.1.3). Thus, it saves the modeller from having to optimize the model by un-initializing variables when they are not needed.

Incremental test generation

Observational equivalence yields greater reduction when fewer actions are visible. Unfortunately, it leaves a large state-space if a component has a "wide" interface, that is, has many actions that interact with other components or the environment. Incremental test generation is effective because it initially hides all external port actions except for the actions at one port. Once a path is extracted for this port, the actions the next port is made visible. However, the previous path for the first external port is also composed into the system, and constrains the states that will be explored. This continues, so that even as larger models are

	REQ	CH	Extracted paths
Ext 1	5	633	
Exts 1..2	5	1038	11
Exts 1..3	5	1038	11, 8
Exts 1..4	5	1038	11, 8, 1
Exts 1..5	5	12390	11, 8, 1, 5
Exts 1..6	5	12715	11, 8, 1, 5, 5
Exts 1..7	5	12790	11, 8, 1, 5, 5, 12

Table 5-10 Numbers of states of REQ+CH after reduction via incremental test generation, plus extracted paths.

considered, the models are constrained further, and thus, each path generation needs to explore relatively few states.

	REQ	DB	CM	CH	Paths extracted
Ext 1	5	4	42	5769	
Exts 1..2	5	4	42	12465	11
Exts 1..3	5	4	42	12465	11, 8
Exts 1..4	5	4	42	12465	11, 8, 3
Exts 1..5	5	4	42	12790	11, 8, 3, 3
Exts 1..6	5	4	125	12790	11, 8, 3, 3, 12

Table 5-9 Numbers of states of REQ+CH+DB+CM after reduction via incremental test generation, plus extracted paths.

Table 5-10 shows its effect on reductions of the simplest case of one call handler. Initially reduces models to 5 and 633 states. Those models are obtained by hiding all external interactions except those belong to external port 1 ('Ext 1'). This is significantly less than 7 and 85956 ('Raw'). From the reduced models, it obtains a path for these interactions consisting of 11 states, or 10 interactions. Next, it hides all external interactions except external ports 1 and 2, which leads to bigger models of 5 and 1038 states. However, the constraint from the previously generated path means much fewer states are actually explored. Indeed, times taken to extract each path remain quite constant (Figure 5-7).

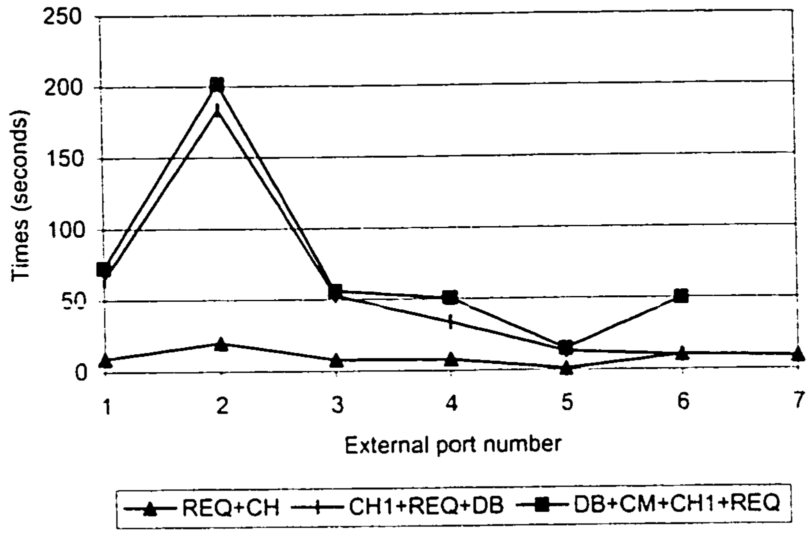

Figure 5-7 Times to extract paths for each external port

The reduction is not as impressive in the larger case of three components (CH +DB +CM) in Table 5-9. Less reduction is obtained because fewer ports are external, and more ports connect with other components; hence, they cannot be hidden. However, the time required to extract a path remains flat, same as in the previous case (Figure 5-7).

With more components and connections, and fewer external ports, incremental test generation alone will fail to create much benefit.

Interaction abstraction

The interaction abstraction algorithm is more clever than the previous approaches in finding regularity in the structure of the state space. It looks at how the state space is actually used by (i.e. interacts with) other components of the system. Then, all states that are used in the same way can be merged. Its effect is similar to abstract interpretation, but its reduction always yields valid results for model checking. It may not reduce models as much as a cleverly chosen abstract interpretation, but it is completely automatic. The reductions achieved with the algorithm are impressive (Table 5-7).

It can be seen from Figure 5-8 to Figure 3-8 that interaction abstraction is much more effective than incremental test generation alone. The figures compare the reduction of model size for each step. For the component, the size at each step is plotted as a percentage of the size of the observationally minimized model. As the number of components increase, incremental test generation alone quickly loses any ability to reduce model sizes. However, interaction abstraction continues to be very effective.

Furthermore, with incremental test generation, model sizes increase as more external ports are considered. However with interaction abstraction, the previously generated paths are used to abstract the models, with the result that the model sizes remain almost constant. Not only do the model sizes remain constant, the time to extract paths and compute reductions also remain constant (Figure 5-13 and Figure 5-14).

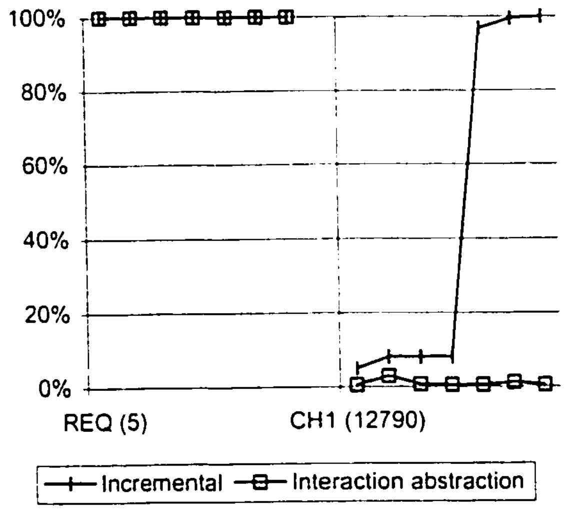

Figure 5-8 Size of reduced REQ+CH as a percent of observationally minimized models. (Number in brackets is size of observationally minimized models.)

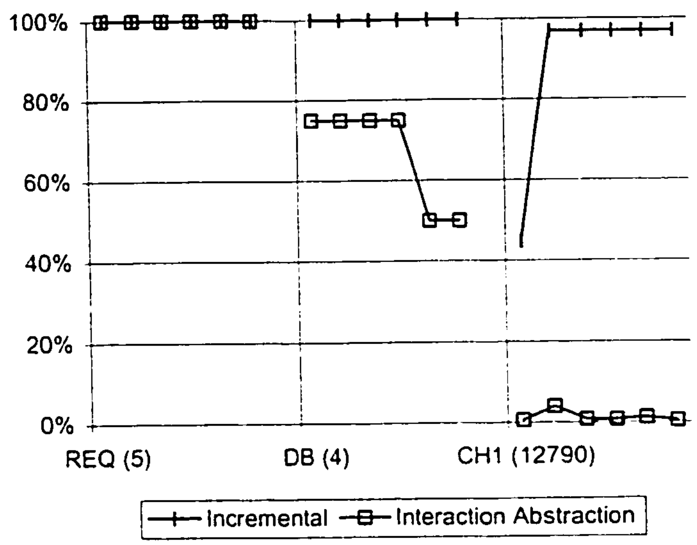

Figure 5-9 Size of reduced REQ+DB+CH as a percent of observationally minimized models. (Number in brackets is size of observationally minimized models.)

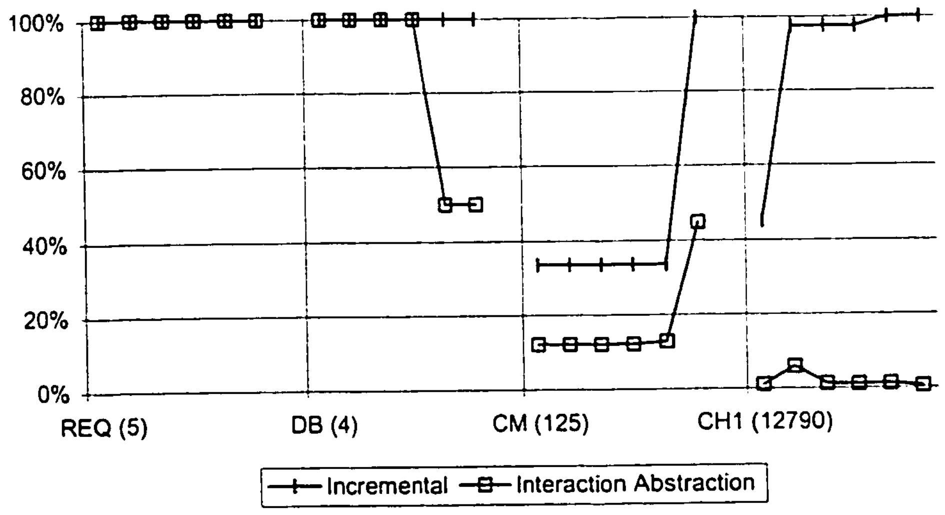

Figure 5-10 Size of reduced REQ+DB+CM+CH as a percent of observationally minimized models. (Number in brackets is size of observationally minimized models.)

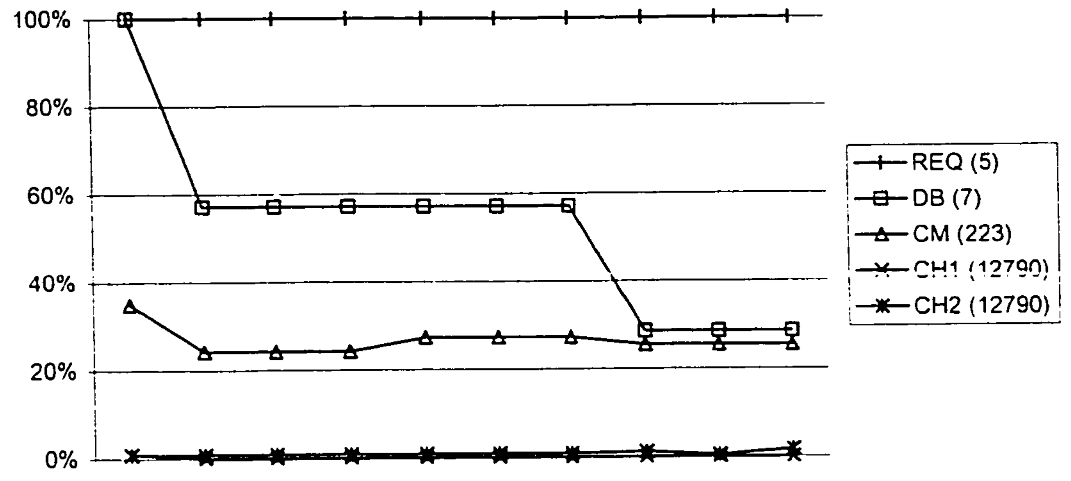

Figure 5-11 Size of REQ+DB+CM+CH1+CH2 reduced by interaction abstraction as a percent of observationally minimized model. (Number in brackets is size of observationally minimized models.)

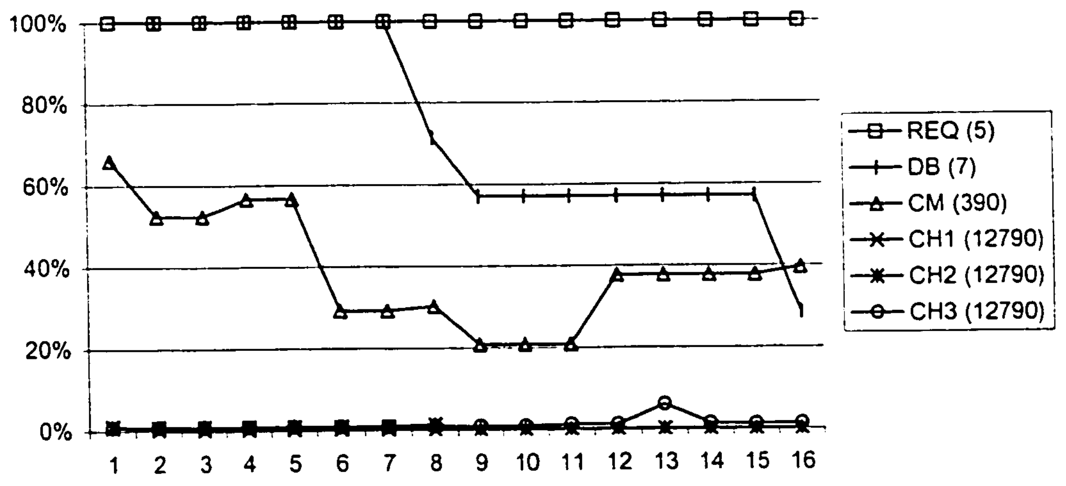

Figure 5-12 Size of REQ+DB+CM+CH1+CH2+CH3 reduced by interaction abstraction as a percent of observationally minimized model. (Number in brackets is size of observationally minimized models.)

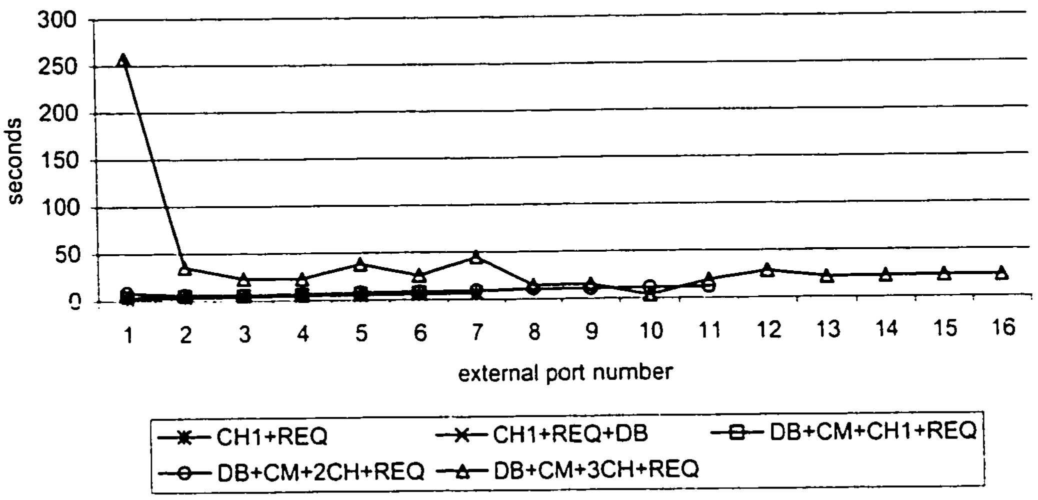

Figure 5-13 Time (seconds) to extract paths for each step from models reduced by interaction abstraction.

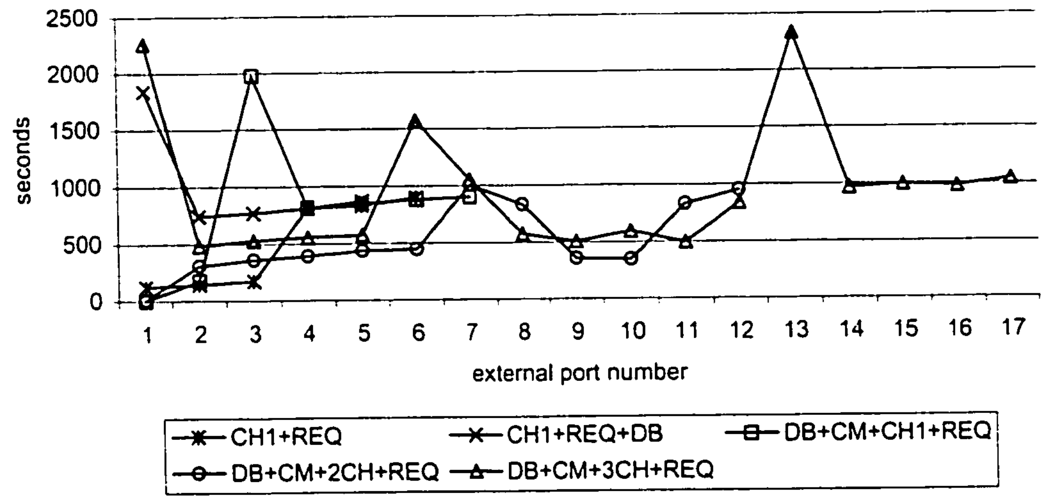

Figure 5-14 Time (seconds) to compute interaction abstraction for each step from models.

Note that interaction abstraction would not work well without incremental test generation, since it does not abstract external actions at all. If all external actions are distinguished, many states must remain distinct, and there is little opportunity to abstract interactions.

5.4.4.2 Redundancy in similar data values

The depth-first algorithm with supertrace performed very poorly for a Call handler with a Database component. It actually completed without finding a path satisfying the test

	SPIN
REQ +CH	554
REQ +CH +DB	27H (Not found)
Reduced REQ +CH +DB	370
Reduced REQ +CH +DB +CM	Time out

Table 5-11 Effect of reducing data in Promela model on model-checking time with SPIN

requirement, in spite of the fact that coverage was estimated to be >99.9% of the states in the model.

Again, the difficulty is due to the data. In the model, there are more than 144 2-digit directory numbers than can be dialled, but only two that correspond to phones (as there are only 2 phones in the system). In the simple case without a database, any directory number dialled can be considered a valid telephone number, since the database's responses are controlled externally. Otherwise, only 2 of 144 are valid.

If the search chooses an execution path where an invalid directory number is dialled, a connection cannot be made, but the search will continue along that path. The breadth-first algorithm performed better because it does not commit to a particular path, but continues to search along all execution paths equally. However, the amount of storage required to track all execution paths can be exponential in the length of the paths.

To demonstrate the effect the redundant data had on the search, the model was modified so that there were only two invalid telephone numbers. Then the depth-first algorithm successfully computed a path in only a few minutes (Table 5-11). (Nevertheless, SPIN was not able to extract a path for even the reduced model with four components.)

Clearly, it is not necessary to explore so many paths with invalid directory numbers. For this test requirement, they all behaved similarly: no connection can be made. Thus, by taking advantage of knowledge about the test requirement, the model can reduce significantly. However, it is not practical to do this manually for each test requirement.

Again, interaction abstraction is of great benefit as it takes advantage of which actions are distinguished by the test requirement and which are not: those: that cause a transition between the same pair of states are not distinguished (or those that do not cause any transition at all). Hence, similar data is merged, and only one representative path needs to be searched.

To see how effective interaction abstraction is, compare the values of Figure 5-15. The first value is the number of states of the call handler with all actions visible, reduced by observational equivalence. The second value shows call handler's interactions with the database completely hidden. The large difference between them is completely due to the range of possible directory numbers. The third value is shows the database interactions abstracted with the actual database, which effectively eliminates the many similar directory numbers, and coalesces them into two distinct abstract interactions.

5.4.4.3 Redundancy in similar components

In addition to not distinguishing invalid directory numbers, the test requirement does not distinguish between call handlers. That is, it does not matter which call handler(s) make the connections between phones. Thus, only one call handler is needed to satisfy the test requirement, and any extra call handlers are redundant.

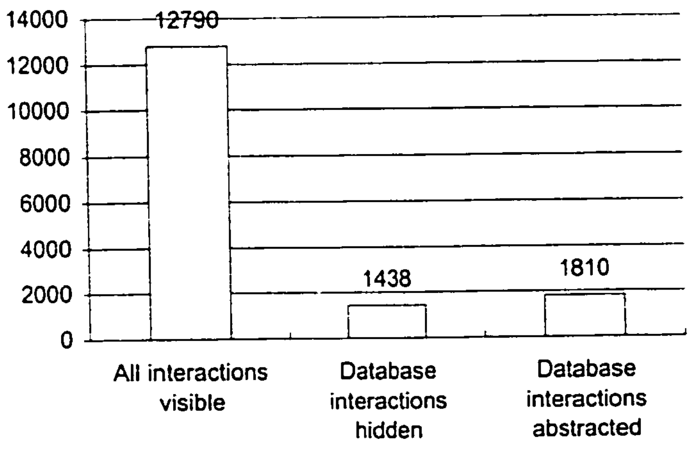

Figure 5-15 Comparison of number of states with interactions hidden versus interactions abstracted.

Incremental test generation and interaction abstraction can take advantage of redundant components to reduce model sizes. From Table 5-12, showing the progress of analyzing REQ+3CH+CM+DB, it can be seen that the first 11 extracted paths have one state, which means there are zero interactions. The first 11 extracted paths are for the external ports of the first two call handlers. Thus, they are not used at all to satisfy the test requirement: all connections are made by the third call handler. As a result, the composition of the extracted paths with the system reduces the first two call handlers to only two states.

The reason the first two call handlers are not used is because the breadth-first search finds the shortest path satisfying the requirement. When considering the first call handler, actions of one of the external ports are visible. Hence, its abstraction is larger than the abstraction of the last

	REQ	DB	CM	CH1	CH2	CH3	Paths extracted
Ext 1	5	10	258	135	105	105	
Exts 1..2	5	7	204	6	105	105	1
Exts 1..3	5	7	204	2	105	105	1
Exts 1..4	5	7	221	5	105	105	1
Exts 1..5	5	7	221	5	105	105	1
Exts 1..6	5	7	114	2	105	105	1
Exts 1..7	5	7	114	2	105	105	1
Exts 1..8	5	5	118	2	149	105	1
Exts 1..9	5	4	81	2	2	105	1
Exts 1..10	5	4	81	2	2	105	1
Exts 1..11	5	4	81	2	2	135	6
Exts 1..12	5	4	147	2	2	135	12
Exts 1..13	5	4	147	2	2	759	3
Exts 1..14	5	4	147	2	2	158	3
Exts 1..15	5	4	147	2	2	139	8
Exts 1..16	5	2	154	2	2	153	11

Table 5-12 Numbers of states of REQ+3CH+CM+DB after reduction via interaction abstraction, plus length of paths extracted for each external port

REQ	DB	CM	LS	CH1	CH2	Abstraction	Extract
5	7	148	861	312	256	12365.7	27300.9
5	4	124	529	19	258	3677.5	2952.8
5	4	124	529	5	258	1462.4	730.5
5	4	124	529	5	258	1494.9	731.3
-	-	-	-	-	-	Time out	-

Table 5-13 Number of states reduced via interaction abstraction, plus times to extract path for each external port, REQ+2CH+CM+DB+LS

two call handlers, which have all external port actions hidden. Hence its interactions will produce longer paths than the last two call handlers, the test requirement would be satisfied using the shorter, abstract paths from the last two call handlers.

In situations of redundant components, the algorithm will bypass partially abstracted components and use fully abstracted components to satisfy the test requirement. Thus, the additional cost of redundant components is the cost of the fully abstracted versions, rather than each partially abstracted version. Ideally, the fully abstracted components are very small, so that the state-explosion is controlled. Table 5-6 shows that the time required by the algorithm grows very slowly with the number of call handlers.

5.4.4.4 Highly-correlated data

The limitation of interaction abstraction can be seen when the linescan process is added to the analysis. Table 5-13 shows the reduced number of states for the analysis. There is a large reduction, but the reduced sizes are still very large. The other notable feature is the jump in the time required for abstraction over other models (from thousands of seconds per analysis to ten thousands of seconds).

The reason for the large jump in reduction time is the linescan process. The process is basically a large table that stores which process has requested to be notified of onhook or offhook events for each phone. New requests overwrite previous requests for a phone. Since table items can be arbitrarily added or changed, the model has many transitions from one state to many other states. In many data structures, the state space is highly interconnected, and there is a transition between nearly every pair of states. As a result, the reduction time has complexity $O(n^4)$.

In addition to longer abstraction times, the reduced components are also relatively large with linescan. The reason is that linescan introduces very complicated interactions between components. If one component overwrites another's request for notification, the other component will not receive any notification. Thus, the interaction of each component with linescan depends closely on the interaction of the other components with linescan. Hence, almost all the information in the linescan component must be retained. This causes a domino effect in the other components so that they must also keep more information about the interaction with linescan.

As an experiment, the linescan model was changed to reduce its raw size. Instead of having linescan poll the hardware and issue phone hook notifications at predefined intervals, the model was changed so that linescan could poll a phone and issue a notification at any time. This reduced the raw number of states of linescan to 11675 compared to 41371. Interestingly, however, the reduced size of linescan after the first external port reduction remained at 861. This experiment shows that the remaining complexity of the reduced linescan component is 'intrinsic' to the design, and not an accident of the particular model code.

5.4.5 Applicability of algorithms

Although the algorithms gave impressive results on this example, one sample is not enough to say definitively if it can perform as well on other models. However, the previous analysis of the results showed several strengths and weaknesses of the algorithms, which may indicate its effectiveness on other models.

The algorithms performed well because there were 144 phone numbers, but only two phones. The same behaviour may be expected for models with data structures, such as lists or sets, when the contents of the data structures are not related to the behaviours of interest. For example, the phone numbers can be longer sequences of digits, and more phones can be added, but the reduced models would remain the same.

Another reason the algorithms worked well was that hiding external ports produced small reduced models, and short test paths. (Short paths greatly reduce the time for extracting paths.) On the other hand, the algorithms *would not* have work well if arbitrary actions were hidden. For example, if some phone numbers and phone offhook were hidden, but other numbers and phone onhook were visible, then very little reduction would be obtained. It is necessary to hide and expose all the phone numbers together, and hide and expose onhook and offhook together.

Intuitively, most software models would have the property that actions at a particular port would be highly related to each other, and hiding them as a whole would abstract away a whole aspect of the behaviour of the software model. In the same way, actions at different ports would involve different aspects, and would affect each other little. Hence, we may expect the same kind of reduction for other models.

On the negative side, the PBX model has a high degree of concurrency: components do not have to wait for each other very much. As well, there are many redundant, relatively independent components. Certainly, the more independent the component, the better the reduction obtained by interaction abstraction. This suggests that the algorithms would have more problems with systems that have tightly-coupled components. For these cases, there is probably not much that can be done, since the complexity is intrinsic to those systems, and not an accidental feature of the representation. They would be very difficult to handle for intelligent humans as well.

In fact, the weakness of the algorithm shown in the example was the handling of the linescan component. The tight coupling and large set of interactions of the component caused the algorithms to fail. This may point to a serious weakness, because message queues used in buffered communications have the same characteristics. Further research is required to develop techniques to handle these situations. One possible approach is to use the interaction abstraction algorithm to abstract more than two interacting components at a time. That is, instead of distinguishing interactions of a component with two different components, view the interaction among all three components as the same kind of interaction (Section 4.4.4). New techniques may be needed to handle the explosion in the number of interactions that result.

5.5 Conclusions

This chapter shows that the techniques proposed in this thesis provide foundations for feasible interaction testing from formal models.

The ObjectState language proved to be suitable for modelling the call processing software for a private branch exchange. The model was found to be simple, and the effort required was reasonable. It required a little more effort compared to an informal model, but it was easier to ensure the model's correctness.

The algorithms to analyze the models can handle realistic components in a reasonable amount of time. The PBX model was too large to fully analyze. However, the interaction abstraction algorithms combined with incremental test generation allowed useful subsets of the models to be analyzed.

The new algorithms were found to be effective at exploiting the type of redundancy that exist in models of software components. In particular, software models for testing must contain many loosely related aspects, and the algorithms were able to exploit these redundancies. They dramatically reduced large state spaces due to liberal use of data, exploiting redundant data values and components. However, the algorithms were not able to prevent state explosion in the case of components that interact closely through a data structure.

Chapter 6

CONCLUSIONS

This thesis introduces techniques that provide foundations for feasible interaction testing from formal models. It presents an object-oriented modelling language that is easy to use for software design, but it also has formal semantics for re-use in interaction testing. Formal coverage criteria and test requirements can be defined for the models, and test cases can be generated using a model-checking approach. Advanced algorithms were created to contain state explosion in the analysis of the formal models for test generation. An experiment using a realistic software model showed that the techniques could handle useful component models with reasonable effort.

6.1 Contributions

The following contributions were made in this thesis.

A formal model of component interactions was selected. Labelled transition systems capture the essential characteristics of interactions. LTS models have a rich theory and a body of efficient algorithms, which allow efficient tools to be created for test generation, simulation, or verification. The formal model is appropriate for sequential, event-driven object-oriented programs, or concurrent programs that interact synchronously, such as by Java's synchronizing methods, Ada rendezvous, or remote procedure calls.

Coverage criteria for interaction testing was formally defined. The formal model and definition of previously informal coverage criteria enables automatic analysis of coverage of a test suite, automatic generation of test cases to achieve required coverage, and objective experiments on coverage criteria effectiveness (not dependent on individual testers).

A formal language was created to model designs of component-based software. The language, ObjectState, combines an easy-to-use informal object-oriented modelling language, such as UML/RT, with a Pascal-like programming language. High-level component interactions are modelled in the object-oriented modelling language, while detailed manipulation of component data is modelled in the programming language. The language was given a formal semantics as LTS.

As an experiment, the call-processing software of a private branch exchange (PBX) was modelled. The features of ObjectState made the model simple and easy to create.

The language was extended to model test requirements, allowing test requirements to be formally specified in a simple way.

The main contribution in the thesis was the creation of two new algorithms to enable test generation from LTS models of software. The incremental test generation and interaction abstraction algorithms significantly reduce the state-explosion problem when analyzing LTS models. The algorithms enable the analysis of far larger models than previous algorithms could handle. At the same time, the algorithms can be used in conjunction with previous algorithms to further increase the size of models that can be analyzed.

The incremental test generation algorithm takes advantage of the fact that test cases can be generated incrementally. Generating a part of a test case only requires a small amount of information, which means the system can be reduced exponentially. As a result, extracting each portion of the test takes much less effort than extracting the entire test case at once.

The interaction abstraction algorithm cleverly removes redundant information from the model, while preserving all information necessary to generate the test case. It is especially effective working with incremental test generation, since it means only a portion of the system behaviour is of interest at a time. The interaction abstraction algorithm avoids state explosion by iteratively minimizing each component. It is shown to have complexity $O(n^4 m^5)$, where n is the number of components, and m is the number of states of a component. (This complexity holds for components with two-way communication. For simultaneous, k-way interactions, the complexity is $O(n^4 m^{2k+1})$.) The correctness of the algorithm is proved.

Interaction abstraction is effective for models of software components, which have the following characteristics:

- the system is made up of many loosely-coupled components; each component communicates with a limited number of other components
- each component is small enough to be generated and minimized, but compositions of components are too large
- large components are the result of handling many loosely-related aspects, and many connections with other components
- only a small part of the system's behaviour is of interest at one time.

Under these reasonable assumptions, the algorithm depends linearly on the number of components, while potentially reducing the global state space exponentially.

Additionally, the interaction abstraction algorithm is useful for general model-checking purposes, especially for verification of properties that depend on a small subset of the actions of a system.

Experiments conducted on the model of the PBX showed the algorithms effectively reduce realistic models of software for test generation. They eliminate redundancy in models that use data structures and variables, contain variables with similar data values, and redundant components. The experiment showed the feasibility of analyzing useful, realistic models for generating test cases.

6.2 Limitations and Future work

The following limitations of the current work need to be addressed.

6.2.1 Testing analysis

The algorithms in this thesis reduce state space models to enable test case generation. However, the paths generated by a standard model-checker are not satisfactory test cases. For example, they do not allow for non-deterministic output from the system. Several previous studies have developed techniques for generating executable test cases from state space models[23][28]. These techniques need to be combined with the reduction algorithms to generate test cases for software component interaction.

Beyond test generation, the formal definition of coverage criteria allows many other types of testing analysis, such as

- generating a **minimal** set of test cases to satisfy a coverage criteria
- evaluating the level of coverage given a set of test cases
- estimating the minimal number of (further) test cases needed to achieve a level of coverage
- selecting tests that are needed for regression testing, given a notion of change in model

Efficient algorithms should be derived for these questions.

6.2.2 Modelling language

The ObjectState language only shows the feasibility of creating formal design models using object-oriented modelling features and an imperative programming language. However, many more features are needed in a practical, formal modelling language. Foremost, a graphical notation and editor should be adapted from an informal languages, such as UML for Real-Time. In addition, features such as dynamic process creation and dynamic communication patterns should be supported. The features can be simulated using ObjectState language constructs, but are common enough to warrant being included in the modelling language.

The most difficult part of detailed models, such as the PBX model, is the code relating to the data structures. The model could be simplified significantly by providing easy-to-use, predefined data types.

6.2.3 Algorithms

Although the algorithms given in this thesis significantly increase the size of models that can be analyzed, they are still too limited for most applications. One of the key problem areas is when many components interact closely using data structures.

A possible direction to explore is extending the interaction abstraction algorithm to consider interactions with a group of components together, rather than distinguishing between interactions with different components (as currently done in Section 4.4.4). Many issues would need to be solved, such as how to prevent an explosion in the number of interactions, or how to compose components that have been reduced in this way.

Another incremental improvement can be obtained by extending the algorithm to handle coarser equivalence relations, such as safety equivalence[9], or trace equivalence. Currently, the algorithm preserves observation equivalence, which is stronger than necessary for testing. A coarser relation would allow greater reduction.

An attractive idea is to create a similar algorithm that preserves partial-order or symmetry equivalence in addition to observation equivalence, using for example, Valmari's technique ([66], page 511).

To go beyond incremental improvements would likely require manipulating data in symbolic rather than expanded form (as concrete transitions and states). A combination of interaction abstraction with abstract interpretation may be useful.

6.2.4 Tools

While prototype tools have been implemented for the techniques in this thesis, the ideas should be incorporated into industrial-strength tools in order to be used for practical applications. In particular, the implementation of the algorithms can be greatly speeded up if tools supported hook transitions directly. For example, minimization tools should treat them as internal transitions.

Appendix A

OBJECTSTATE MODELS OF PBX

A.1 Model of Call handler

```
#!Perl
package models::Callhandler;
use strict;
use trans::Parsemodel;
use models::Porttypes;

DEFINE call_handler_proc =>  PROCTYPE {
    CONST
       'digit_1_timeout : 5;
         digit_2_timeout : 10;
         ringing_timeout : 120;
        fast_busy_timeout : 120;
        slow_busy_timeout : 120;';
    PORT
       Timer => 'timer_interface',
       Call_manager => 'call_manager_interface',
       Line_scan => 'line_scan_interface',
       Database => 'office_database_interface',
       'TTRX_scanner[ttrx_EN]' => 'TTRX_scanner_interface',
       Resource_manager => 'resource_manager_interface',
       HW => 'hardware_interface',
       ;
    VAR
       Handled_phone_EN => 'phone_EN',
       Dialled_DN => 'directory_number',
       Dialled_digit => 'digit',
       Dialled_phone_EN => 'phone_EN',
       Dialled_phone_status => 'equipment_status',

       TTRX => 'ttrx_EN',
       Out_channel => 'channel_id',
       In_channel => 'channel_id',
       ;

 # the code:
     STATES 'wait for call', 'handle call';
     ENTER 'wait for call';

 ######### main sequence
     DEFINE 'wait for call' =>  STATE {
       WAIT {
        OP 'Call_manager?handle_call(Handled_phone_EN)';
        ENTER 'handle call';
       };
     };
     DEFINE 'handle call' =>  STATE {
       STATES 'get resources', 'has resources';
       INIT {
        OP 'Line_scan!request(MYPID, Handled_phone_EN, onhook)';
       ENTER 'get resources';
      };
      ONEXIT {
       OP 'Call_manager!finished_call(Handled_phone_EN)',
       'undefine Handled_phone_EN';
      };
      WAIT {
       OP 'Line_scan?notify(Handled_phone_EN, [onhook])';
       CALL 'handled phone onhook';
       OP 'Line_scan!request(call_manager_pid, Handled_phone_EN, offhook)';
       ENTER 'wait for call';
      };
      WHEN 'idle handled phone' => HANDLE {
       OP 'Line_scan!cancel(Handled_phone_EN)'; # need this?
       OP 'Line_scan!request(call_manager_pid, Handled_phone_EN, onhook)';
       ENTER 'wait for call';
      };
    };

########### resources part
    DEFINE 'get resources' => STATE {
      INIT {
       OP 'Resource_manager!request()';
       IF {
         OP 'Resource_manager?grant_all(In_channel, Out_channel, TTRX)';
         ENTER 'get dialled number';
       } OR {
         OP 'Resource_manager?grant_channels(In_channel, Out_channel)';
         ENTER 'fast busy phone';
       } OR {
         OP 'Resource_manager?refuse()';
         CALL 'idle handled phone';
       };
      };
    };
    DEFINE 'has resources' => STATE {
      STATES 'fast busy phone', 'slow busy phone',
         'try ring phone', 'call dialled phone', 'get dialled EN';
      ONEXIT {
       OP 'Resource_manager!release_channels(In_channel, Out_channel)';
       CALL 'set idle connection';
      };
    };
 ############## dialing part
    DEFINE 'get dialled EN' => STATE {
      STATES 'has dialing resources', 'look up dialled DN';
      ONEXIT {
        OP 'undefine Dialled_DN';
      }
    };
    DEFINE 'has dialing resources' => STATE {
      STATES 'get dialled number';
      ONEXIT {
       OP 'Resource_manager!release_TTRX(TTRX)';
      };
    };
    DEFINE 'get dialled number' => STATE {
      INIT {
       OP 'Timer!set(digit_1_timeout)';
       OP 'initialize(Dialled_DN)';
       CALL 'set dial tone connection';
      };
      ONEXIT {
```

```
      CALL 'remove dial connection';  # TTRX
     };
     WAIT {
      OP 'Timer?timeout()';
      ENTER 'fast busy phone';
     };
     WAIT {
      OP 'TTRX_scanner[TTRX]?digit_dialled(Dialled_digit)',
        'append_digit(Dialled_DN, Dialled_digit); undefine Dialled_digit';
      CALL 'set idle tone connection';
      IF {
        OP '[is_complete(Dialled_DN) & is_valid(Dialled_DN)]',
          'Timer!cancel()';
        ENTER 'look up dialled DN';
      } OR {
        OP '[!is_complete(Dialled_DN) & is_valid(Dialled_DN)]',
           'Timer!set(digit_2_timeout)'; # cancel first?
      } OR {
        OP '[!is_valid(Dialled_DN)]',
           'Timer!cancel()';
        ENTER 'fast busy phone';
      };
     };
     WHEN 'handled phone onhook' =>  HANDLE {
      OP 'Timer!cancel()';
     };
    };
    DEFINE 'look up dialled DN' =>  STATE {
      INIT {
       OP 'Database!request_EN(Dialled_DN.Digits[1], Dialled_DN.Digits[2])';
       OP 'Database?reply_EN(Dialled_phone_EN, Dialled_phone_status)';
       IF {
         OP '[!is_error_EN(Dialled_phone_EN) & is_active(Dialled_phone_status)]',
         'undefine Dialled_phone_status';
         ENTER 'try ring phone';
       } OR {
         OP '[is_error_EN(Dialled_phone_EN) | !is_active(Dialled_phone_status)]',
         'undefine Dialled_phone_EN; undefine Dialled_phone_status';
         ENTER 'fast busy phone';
       };
      };
    };

####### ringing part
    DEFINE 'try ring phone' =>  STATE {
      INIT {
       OP 'Call_manager!get_phone(Dialled_phone_EN)';
       IF {
         OP 'Call_manager?phone_busy(_)',
         'undefine Dialled_phone_EN';
         ENTER 'slow busy phone';
       } OR {
         OP 'Call_manager?phone_obtained(_)';
         ENTER 'wait pickup phone';
       };
      };
    };

######### call dialled phone
    DEFINE 'call dialled phone' =>  STATE {
      STATES 'wait pickup phone', 'conversation';
      ONEXIT {
       OP 'Call_manager!release_phone(Dialled_phone_EN)';
      };
      WHEN 'idle dialled phone' => HANDLE {
       OP 'Line_scan!request(call_manager_pid, Dialled_phone_EN, onhook)';
      };
    };
    DEFINE 'wait pickup phone' => STATE {
      INIT {
       OP 'Line_scan!request(MYPID, Dialled_phone_EN, offhook)';
       OP 'Timer!set(ringing_timeout)';
       CALL 'set ringing connection';
      };
      ONEXIT {
       CALL 'remove ringing connection';
      };
      WAIT {
       OP 'Line_scan?notify([Dialled_phone_EN], [offhook])';
       OP 'Timer!cancel()';
       ENTER 'conversation';
      };
      WAIT {
       OP 'Timer?timeout()';
       OP 'Line_scan!cancel(Dialled_phone_EN)';
       ENTER 'fast busy phone';
      };
      WHEN 'handled phone onhook' => HANDLE {
       OP 'Line_scan!cancel(Dialled_phone_EN)';
       OP 'Timer!cancel()';
      };
    };
    DEFINE 'conversation' => STATE {
      INIT {
       OP 'Line_scan!request(MYPID, Dialled_phone_EN, onhook)';
       CALL 'set conversation connection';
      };
      ONEXIT {
       CALL 'remove conversation connection';
      };
      WAIT {
       OP 'Line_scan?notify([Dialled_phone_EN], [onhook])';
       CALL 'idle dialled phone';
       CALL 'idle handled phone';
      };
      WHEN 'handled phone onhook' => HANDLE {
       CALL 'idle dialled phone';
      };
    };

######### tones part
    DEFINE 'slow busy phone' => STATE {
      INIT {
       CALL 'set slow busy connection';
       OP 'Timer!set(slow_busy_timeout)';
      };
      WAIT {
       OP 'Timer?timeout()';
       ENTER 'fast busy phone';
      };
      WHEN 'handled phone onhook' => HANDLE {
       OP 'Timer!cancel()';
      };
    };
    DEFINE 'fast busy phone' => STATE {
      INIT {
        CALL 'set fast busy connection';
        OP 'Timer!set(fast_busy_timeout)';
       };
       WAIT {
        OP 'Timer?timeout()';
        CALL 'idle handled phone';
       };
       WHEN 'handled phone onhook' => HANDLE {
        OP 'Timer!cancel()';
       };
     };

########## connection part
    DEFINE 'set dial tone connection' => SEQUENCE {
      OP 'HW!set_idle(Handled_phone_EN, idle_off)';
      OP 'HW!connect(Handled_phone_EN, TTRX, Out_channel)';
      OP 'HW!connect(Dial_tone_EN, Handled_phone_EN, In_channel)';
    };

    DEFINE 'set idle tone connection' => SEQUENCE {
      OP 'HW!connect(Idle_tone_EN, Handled_phone_EN, In_channel)';
    };

    DEFINE 'remove dial connection' => SEQUENCE {
      OP 'HW!disconnect(Out_channel)';
    };

    DEFINE 'set ringing connection' => SEQUENCE {
      OP 'HW!connect(Ring_tone_EN, Handled_phone_EN, In_channel)';
      OP 'HW!set_ringing(Dialled_phone_EN, ringing_on)';
    };

    DEFINE 'remove ringing connection' => SEQUENCE {
      OP 'HW!connect(Idle_tone_EN, Handled_phone_EN, In_channel)';
      OP 'HW!set_ringing(Dialled_phone_EN, ringing_off)';
    };

    DEFINE 'set idle connection' => SEQUENCE {
      OP 'HW!set_idle(Handled_phone_EN, idle_on)';
    };

    DEFINE 'set conversation connection' => SEQUENCE {
      OP 'HW!set_idle(Dialled_phone_EN, idle_off)';
      OP 'HW!connect(Dialled_phone_EN, Handled_phone_EN, In_channel)';
      OP 'HW!connect(Handled_phone_EN, Dialled_phone_EN, Out_channel)';
    };

    DEFINE 'remove conversation connection' => SEQUENCE {
      OP 'HW!disconnect(In_channel)';
      OP 'HW!disconnect(Out_channel)';
      OP 'HW!set_idle(Dialled_phone_EN, idle_on)';
    };

    DEFINE 'set slow busy connection' => SEQUENCE {
      OP 'HW!connect(Slow_busy_tone_EN, Handled_phone_EN, In_channel)';
    };

    DEFINE 'set fast busy connection' => SEQUENCE {
      OP 'HW!connect(Fast_busy_tone_EN, Handled_phone_EN, In_channel)';
    };

  }; # proctype
```

A.2 Model of Call manager

```
#!Perl
package models::Callmanager;

use trans::Parsemodel;
use models::Porttypes;

DEFINE call_manager_proc => PROCTYPE {
    PORT
        'Handlers[handler_pid]' => 'call_manager_interface',
       Line_scan => 'line_scan_interface',
       :
    VAR
       Free_handlers => 'set_of_handler_pids',
       Free_phones => 'set_of_phone_ENs',
       Handler => 'handler_pid',
       EN => 'phone_EN';

    OP 'set_initialize_handler_pids(Free_handlers);
       for i : handler_pid do
           set_insert_handler_pids(Free_handlers, i);
       end;
       set_initialize_phone_ENs(Free_phones);';
    # send all the requests
    DO {
       # order of scan requests is not determined
       SELECT [Next_EN => 'phone_EN'] =>
         '[ !set_exists_phone_ENs(Free_phones, Next_EN)]',
         # Next_EN != error_EN &
         'Line_scan!request(MYPID, Next_EN, offhook)',
         'set_insert_phone_ENs(Free_phones,Next_EN);';
    } OR {
       OP '[forall i : phone_EN do set_exists_phone_ENs(Free_phones, i) end]';
         # i=error_EN |
       ENTER 'handle requests';
    };

    DEFINE 'handle requests' => STATE {
       WAIT {
           OP 'Line_scan?notify(EN, [onhook])',
           'set_insert_phone_ENs(Free_phones, EN)';
           OP 'Line_scan!request(MYPID, EN, offhook)',
           'undefine EN';
       };
       WAIT {
           OP 'Line_scan?notify(EN, [offhook])',
             'set_remove_phone_ENs(Free_phones, EN)';
           IF {
              # which handler is not determined
              SELECT [Free_handler => 'handler_pid'] =>
                  '[set_exists_handler_pids(Free_handlers, Free_handler)]',
                  'set_remove_handler_pids(Free_handlers, Free_handler);
                   Handler := Free_handler';
              OP 'Handlers[Handler]!handle_call(EN)',
              'undefine EN; undefine Handler';
           } OR {
              OP 'Line_scan!request(MYPID, EN, onhook)',
              '[set_no_more_handler_pids(Free_handlers)]',
              'set_remove_phone_ENs(Free_phones, EN);
                 undefine EN';
           }
        };
```

```
      WAIT {
          OP 'Handlers[Handler]?finished_call(EN)',
             'set_insert_handler_pids(Free_handlers, Handler);
             undefine Handler; undefine EN';
      };

      WAIT {
          OP 'Handlers[Handler]?get_phone(EN)',
             '[set_exists_phone_ENs(Free_phones, Message_Phone)]',
             'set_remove_phone_ENs(Free_phones, EN)';
          OP 'Handlers[Handler]!phone_obtained(EN)',
             'undefine Handler; undefine EN';
      };
      WAIT {
          OP 'Handlers[Handler]?get_phone(EN)',
          '[!set_exists_phone_ENs(Free_phones, Message_Phone)]';
          OP 'Handlers[Handler]!phone_busy(EN)',
          'undefine Handler; undefine EN';
      };
      WAIT {
          OP 'Handlers[_]?release_phone(EN)',
          'set_insert_phone_ENs(Free_phones, EN);
           undefine EN';
      };
    }; # state
};

if (0) {
    CONST
        SCANINTERVAL => '1',
        ;
    PORT
       Timer => 'timer_interface';
    WAIT {
       OP 'Timer?timeout()';
       OP 'Timer!set(SCANINTERVAL)'; # done, restart timeout
    };
};

1;
```

A.3 Model of Linescan

```
#!Perl
package models::Linescan;
use strict;
BEGIN {
   use Exporter    ();
   use vars        qw($VERSION @ISA @EXPORT @EXPORT_OK %EXPORT_TAGS);
   @ISA          = qw(Exporter);
   @EXPORT    = qw($scanreq_table);
}
use vars @EXPORT;

use trans::Parsemodel;
use models::Datatypes;
use models::Porttypes;

$scanreq_table = table_of('scanreq', 'phone_EN'=> 'process_id','hook_status');

DEFINE line_scan_proc => PROCTYPE {
    CONST
        'SCANINTERVAL : 4;',
          ;
```

```
  PORT
     'Reqline[process_id]' => 'line_scan_interface',
     HW => 'hardware_interface',
     ;
  VAR                              # these names get aliased
     En => 'phone_EN',
     Requestor => 'process_id',
     Hook => 'hook_status',
     Table => 'table_of_scanreqs',
     ;
  STATES 'wait';

  OP 'table_initialize_scanreqs(Table);';
  ENTER 'wait';

  DEFINE 'wait' => STATE {
    WAIT {
       OP 'Reqline[_]?request(Requestor,En,Hook)',
         'table_set_scanreqs(Table, En, Requestor, Hook)';
    };
    WAIT {
       OP 'Reqline[_]?cancel(En)',
         'table_remove_scanreqs(Table, En);';
    };
    WAIT {
       SELECT [Check_EN => 'phone_EN'] =>
         '[table_exists_scanreqs(Table, Check_EN)]',
         'table_get_scanreqs(Table, Check_EN, Requestor, Hook);
          En := Check_EN';
       IF {
           OP 'HW?hook_status([En], [Hook])';
           OP 'Reqline[Requestor]!notify(En, Hook)',
            'table_remove_scanreqs(Table, En);';
       } OR {
           OP 'HW?hook_status([En], _)',
            '[Message_Status != Hook]';
       }
     };
    };
};

if (0) {
    PORT
       Timer => 'timer_interface',
       ;
    VAR
       Iter => 'set_of_phone_ENs',
       ;
    OP 'Timer!set(SCANINTERVAL)';
    OP 'set_initialize_process_ids(Iter);';
      WAIT {
         OP 'Timer?timeout()';
         CALL 'scan';
         OP 'Timer!set(SCANINTERVAL)';
      };
    DEFINE 'scan' => SEQUENCE {
      OP 'for i : phone_EN do
           if table_exists_scanreqs(Table, i) then
             set_insert_process_ids(Iter, i);
           end;
         end;';
      DO {
       SELECT [Next_EN => 'phone_EN'] =>
```

```
            '[set_exists_phone_ENs(Iter, Next_EN)]',
            'set_remove_phone_ENs(Iter, Next_EN);
             table_get_scanreqs(Table, Next_EN, Requestor, Hook);
             En := Next_EN;';
       IF {
           OP 'HW?hook_status([En], [Hook])';
           OP 'Reqline[Requestor]!notify(En, Hook)',
              'table_remove_scanreqs(Table, Requestor);';
       } OR {
           OP 'HW?hook_status([En], _)',
              '[Message_Status != Hook]'
       }
      } OR {
         OP '[set_no_more_process_ids(Iter)]';
         GOTO 'endloop';
      };
      LABEL 'endloop';
    };
}

1;
```

A.4 Model of Database

```
#!Perl
package models::Database;
use strict;
use trans::Parsemodel;
use models::Datatypes;
use models::Porttypes;

DEFINE office_database_proc => PROCTYPE {
  PORT
    'Reqline[handler_pid]' => 'office_database_interface',
  ;
  VAR
    Digit1 => 'digit',
    Digit2 => 'digit',
    Requestor => 'handler_pid',
    DNTable => 'array [1..8] of array [1..2] of digit',
  ;
  OP 'DNTable[1][1]  := key1; DNTable[1][2]  := key1;
      DNTable[2][1]  := key4; DNTable[2][2]  := key4;
      DNTable[3][1]  := key5; DNTable[3][2]  := key3;
      DNTable[4][1]  := key2; DNTable[4][2]  := key3;
      DNTable[5][1]  := key5; DNTable[5][2]  := key2;
      DNTable[6][1]  := key7; DNTable[6][2]  := key4;
      DNTable[7][1]  := key4; DNTable[7][2]  := key3;
      DNTable[8][1]  := key2; DNTable[8][2]  := key6';
  DO {
    OP 'Reqline[Requestor]?request_EN(Digit1,Digit2)';
    IF {
      OP '[forall Phone : phone_EN do
           Phone = error_EN |
           (DNTable[Phone][1] != Digit1 | DNTable[Phone][2] != Digit2)
         end]',
         'Reqline[Requestor]!reply_EN(error_EN, inactive)';
    } OR {
      SELECT [Phone => 'phone_EN'] =>
       '[Phone != error_EN &
        (DNTable[Phone][1] = Digit1 & DNTable[Phone][2] = Digit2)]',
       'Reqline[Requestor]!reply_EN(Phone,active)';
    }
  }
}
```

A.5 Model of Port types

```
#!Perl
package models::Porttypes;
use strict 'vars';
use strict 'refs';
BEGIN {
   use Exporter   ();
   use vars       qw($VERSION @ISA @EXPORT @EXPORT_OK %EXPORT_TAGS);
   @ISA          = qw(Exporter);
   @EXPORT = qw(&forgetful_buffer_of );
}
use vars @EXPORT;
use trans::Parsemodel;

DEFINE timer_interface => PORTTYPE {
  'set' => [ Time => 'timeout_interval' ],
  timeout => [ ],
  cancel => [ ],
};
DEFINE line_scan_interface => PORTTYPE {
  request => [ Requestor => 'process_id',
             Phone => 'phone_EN',
             Hook => 'hook_status'],
  notify => [ Phone => 'phone_EN',
             Hook => 'hook_status'],
  cancel => [  Phone => 'phone_EN', ],
};
DEFINE call_manager_interface => PORTTYPE {
  handle_call => [ Phone => 'phone_EN'],
  finished_call => [ Phone => 'phone_EN'],
  get_phone => [ Phone => 'phone_EN'],
  release_phone => [ Phone => 'phone_EN'],
  phone_busy => [ Phone => 'phone_EN'],
  phone_obtained => [ Phone => 'phone_EN'],
};
DEFINE office_database_interface => PORTTYPE {
  request_EN => [ Digit1 => 'digit', Digit2 => 'digit' ],
  reply_EN => [ Phone => 'phone_EN',
             Status => 'equipment_status'],
};
DEFINE TTRX_scanner_interface => PORTTYPE {
  request => [ ],
  cancel => [ ],
  digit_dialled => [ Digit => 'digit'],
};
DEFINE resource_manager_interface => PORTTYPE {
  request => [ ],
  grant_all => [ Channel1 => 'channel_id',
            Channel2 => 'channel_id',
            TTRX => 'ttrx_EN'],
  grant_channels => [ Channel1 => 'channel_id',
                   Channel2 => 'channel_id',],
  refuse => [ ],
  release_channels => [ Channel1 => 'channel_id',
                     Channel2 => 'channel_id'],
  release_TTRX => [ TTRX => 'ttrx_EN'],
};
DEFINE hardware_interface => PORTTYPE {
  'set_idle' => [ EN => 'phone_EN',
               Status => 'idle_status'],
  'connect' => [ EN1 => 'equipment_number',
                 EN2 => 'equipment_number',
              Channel => 'channel_id'],
  disconnect => [ Channel => 'channel_id'],
  'set_ringing' => [ Line => 'phone_EN',
                    Status => 'ringing_status'],
  hook_status => [Line => 'phone_EN',
                   Status =>'hook_status'],
};

1;
```

A.6 Model of Data types

```
#!Perl
package models::Datatypes;
use strict;
BEGIN {
   use Exporter    ();
   use vars        qw($VERSION @ISA @EXPORT @EXPORT_OK %EXPORT_TAGS);
   @ISA          = qw(Exporter);
   @EXPORT   = qw($phone_types $set_of_phone_EN $set_of_handler_pid
               $directory_number_type &set_of &table_of);
}
use vars @EXPORT;
use trans::Config;

######################## PBX-specific data types ########################
$phone_types = sub {
  DECLARE 'const
    error_EN : NUM_PHONES + 1;
    FIRST_TTRX_EN : error_EN + 1;
    LAST_TTRX_EN : FIRST_TTRX_EN + NUM_TTRXES-1;
    FIRST_TONE_EN : LAST_TTRX_EN;
    Dial_tone_EN : FIRST_TONE_EN+1;
    Idle_tone_EN : FIRST_TONE_EN+2;
    Ring_tone_EN : FIRST_TONE_EN+3;
    Slow_busy_tone_EN : FIRST_TONE_EN+4;
    Fast_busy_tone_EN : FIRST_TONE_EN+5;
    LAST_TONE_EN : Fast_busy_tone_EN;
    LAST_EN : LAST_TONE_EN;';
  DECLARE 'type
    phone_EN : 1..error_EN;
    ttrx_EN : FIRST_TTRX_EN..LAST_TTRX_EN;
    equipment_number : 1 .. LAST_EN;

    channel_id : Scalarset(NUM_CHANNELS);
    equipment_status : enum {active, inactive};
    hook_status : enum {offhook, onhook};
    ringing_status : enum {ringing_off, ringing_on};
    idle_status : enum {idle_off, idle_on};

    timeout_interval : 1..600;';
  DECLAREPROC '
 function is_error_EN(EN : phone_EN) : boolean;
   return EN = error_EN;
 end;
 function is_active(Status : equipment_status) : boolean;
   return Status = active;
 end;
 ';
   INCLUDE $directory_number_type;
 };
```

```
$directory_number_type = sub {
  DECLARE 'type
  digit : enum {key0,key1,key2,key3,key4,key5,key6,key7,key8,key9,
              keyhash,keystar};
  directory_number : record
    Digits : array [1..2] of digit;
    NumDigits : 0..2;
  end;';
  DECLAREPROC '
procedure append_digit(var DN:directory_number; Digit:digit);
begin
  if DN.NumDigits < 2 then
    DN.NumDigits:=DN.NumDigits+1;
    DN.Digits[DN.NumDigits] := Digit;
  end;
end;
function get_digit(DN:directory_number; DigitNum : 1..2) : digit;
begin
  return DN.Digits[DigitNum];
end;
procedure initialize(var DN:directory_number);
begin
  DN.NumDigits:=0;
end;
function is_complete(DN:directory_number) : boolean;
  return DN.NumDigits = 2;
end;
function is_valid(DN:directory_number) : boolean;
begin
  return forall i : 1..2 do
    (i > DN.NumDigits) | ((DN.Digits[i] != keyhash) & (DN.Digits[i] != keystar))
  end;
end;';
};

$set_of_phone_EN = set_of('phone_EN');
$set_of_handler_pid = set_of('handler_pid');


######################## generic data types #################
# table, set

# implementation of sets
sub set_of {
    my $elemtype = shift;
    return sub {
       DECLARE "type
    set_of_${elemtype}s : Array [ ${elemtype} ] of boolean;";
       DECLAREPROC "
procedure set_initialize_${elemtype}s (var set : set_of_${elemtype}s);
begin
    for i : ${elemtype} do
        set[i] := false;
    end;
end;

procedure set_insert_${elemtype}s (var set : set_of_${elemtype}s;
                                   elem : ${elemtype});
begin
  if isundefined(elem) then
      return;
  end;
  set[elem] := true;
```

```
end;

procedure set_remove_${elemtype}s (var set : set_of_${elemtype}s;
                                   elem : ${elemtype});
begin
  if isundefined(elem) then
      return;
  end;
  set[elem] := false;
end;

function set_exists_${elemtype}s (set : set_of_${elemtype}s;
                                  elem : ${elemtype}) : boolean;
begin
  if isundefined(elem) then
      return false;
  end;
  return set[elem];
end;

function set_no_more_${elemtype}s (set : set_of_${elemtype}s)
    : boolean;
begin
  return forall i : ${elemtype} do
     set[i] = false
  end;
end;";
     };
}

sub table_of {
    my $tabletype = shift;
    my $indextype = shift;
    my @elemtypes = @_;
    my $paramdecl = "";
    my $varparamdecl = "";
    my $paramsets = "";
    my $paramgets = "";
    for (my $i=0; $i< @elemtypes; $i++) {
        $paramdecl .= "\n    p${i} : $elemtypes[$i];";
        $varparamdecl .= "\n    var p${i} : $elemtypes[$i];";
        $paramsets .= "\n    p${i} := table[index].p${i};";
        $paramgets .= "\n    table[index].p${i} := p${i};";
    }
    return sub {
       DECLARE "type
  element_of_${tabletype} : Record
    in_set : boolean; ${paramdecl}
  end;
  table_of_${tabletype}s : Array [ ${indextype} ] of element_of_${tabletype};";
       DECLAREPROC "
procedure table_initialize_${tabletype}s (var table : table_of_${tabletype}s);
begin
    for i : ${indextype} do
        table[i].in_set := false;
    end;
end;

procedure table_set_${tabletype}s (var table : table_of_${tabletype}s;
                                   index : ${indextype}; ${paramdecl});
begin
  if isundefined(index) then
      return;
```

```
   end;
   ${paramgets}
   table[index].in_set := true;
end;

procedure table_get_${tabletype}s (table : table_of_${tabletype}s;
                                   index : ${indextype}; ${varparamdecl});
begin
   if isundefined(index) then
      return;
   end;
   ${paramsets}
end;

procedure table_remove_${tabletype}s (var table : table_of_${tabletype}s;
                                      index : ${indextype});
begin
   if isundefined(index) then
      return;
   end;
     undefine table[index];
     table[index].in_set := false;
end;

function table_exists_${tabletype}s (table : table_of_${tabletype}s;
                                   index : ${indextype}) : boolean;
begin
   if isundefined(index) then
      return false;
   end;
   return table[index].in_set;
end;

function table_no_more_${tabletype}s (table : table_of_${tabletype}s)
      : boolean;
begin
   return forall i : ${indextype} do
      table[i].in_set = false
   end;
end;";
      };
}
```

A.7 Model of Test requirement

```
#!Perl
package models::Connectreq;
use strict;
use trans::Parsemodel;
use models::Porttypes;

DEFINE connect_2_req => PROCTYPE {
     VAR
        Connected => '1..3';
     PORT
        'HW[handler_pid]' => 'hardware_interface';
     OP 'Connected := 1';
     DO { OP 'HW[_]?connect(_, _, _)',
         '[Message_EN1 <= NUM_PHONES & Message_EN2 <= NUM_PHONES & Connected < 3]',
         'Connected := Connected + 1';
      }
     OR { OP 'HW[_]?connect(_, _, _)',
         '[Message_EN1 <= NUM_PHONES & Message_EN2 <= NUM_PHONES & Connected = 3]';
```

```
          ACCEPT;
       }
     OR { OP 'HW[_]?connect(_,_,_)';}
     OR { OP 'HW[_]?disconnect(_)';}
     OR { OP 'HW[_]?set_idle(_,_)';}
     OR { OP 'HW[_]?set_ringing(_,_)';}
};
```

Appendix B

PARAMETERS FOR PATH GENERATION

B.1 Parameters for Araprod

```
araprod -s -m 18000000 -o comp -M acceptstate -T file.lts -t
file.lts ...
probe comp
query volatile verbose bspan(true) %1
```

where acceptstate is the name of the state where the test requirement is satisfied and file.lts is the LTS representations (in ARA format) of a component and test requirement.

B.2 Parameters for Exhibitor

```
exp.open file.exp exhibitor < accept.seq
```

where file.exp contains the composition expression of the LTS representations of the components, and accept.seq is a sequence pattern specifying the satisfaction of the test requirement.

B.3 Parameters for Aldebaran

```
aldebaran -pequ -bdd -bddsize 120 -labels 40000 -hide
      conn.hide file.exp accept.aut
```

where file.exp contains the composition expression of the LTS representations of the components, and accept.aut is an LTS specifying the satisfaction of the test requirement.

B.4 Parameters for Spin

```
spin -a analyzer.prom
gcc -DBITSTATE -DMEMLIM=200 -DSAFETY -DVECTORSZ=2000 pan.c
./a.out -w28
```

where analyzer.prom is the Promela description of the system of components with the test requirement..

BIBLIOGRAPHY

[1] Robert J. Allen, David Garlan, and James Ivers. Formal Modeling and Analysis of the HLA Component Integration Standard. *Proceedings of the Sixth International Symposium on the Foundations of Software Engineering (FSE-6)*, November 1998.

[2] R. Alur and M. Yannakakis. Model checking of hierarchical state machines. *Sixth ACM Symposium on the Foundations of Software Engineering*, pp. 175-188, 1998

[3] Paul Ammann and Paul E. Black. A specification-based coverage metric to evaluate test sets. In *HASE '99: Proceedings of the 4th IEEE International Symposium on High-Assurance Systems*, pages 239-248, Washington, DC, November 1999.

[4] B. Baumgarten and A. Giessler, *OSI Conformance Testing Methodology and TTCN*, 1994, North Holland.

[5] Boris Beizer, *Software Testing Techniques (2nd edition)*, Van Nostrand Reinhold, 1990

[6] Benveniste, B. Caillaud, P. Le Guernic. From synchrony to asynchrony, *Proceedings of CONCUR'99, Concurrency Theory, 10th International Conference*. Eindhoven, The Netherlands, August 1999, LNCS 1664, Springer.

[7] Robert V. Binder. *The FREE Approach to Object-Oriented Testing: An Overview*. Available at http://www.rbsc.com/pages/FREE.html.

[8] Gregor V. Bochmann and Alexandre Petrenko. Protocol testing: review of methods and relevance for software testing. *Proceedings Of The 1994 International Symposium On Software Testing And Analysis*, p109-124

[9] A. Bouajjani, J.-C. Fernandez, S. Graf, C. Rodriguez and J. Sifakis. Safety for Branching Time Semantics. *18th ICALP*, Springer-Verlag, July 1991.

[10] A. Bouajjani, J.C. Fernandez, N. Halbwachs, C. Ratel, and P. Raymond. Minimal state graph generation. *Science of Computer Programming*, 18(3), June 1992.

[11] R.E. Bryant. Graph-Based Algorithms For Boolean Function Manipulation. *IEEE Transactions On Computers*, C-35 (8) 1986, pp. 677-691.

[12] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and J. Hwang. *Symbolic Model Checking: 10^{20} states and beyond.* Technical Report, Carnegie Mellon University, 1989.

[13] R. H. Carver and K.-C. Tai. Use of Sequencing Constraints for Specification-Based Testing of Concurrent Programs. *IEEE Transactions on Software Engineering*, v. 24, n. 6, June 1998.

[14] CCITT Recommendation Z.100. *Specification and Description Language SDL*. Annexes A-F to Z.100. 1988. (Blue Book, Volume VI.20 - VI.24, ITU, General Secretariat-Sales Section, Places des Nations, CH-1211 Geneva 20)

[15] S. C. Cheung and J. Kramer. Context Constraints for Compositional Reachability Analysis. *ACM Transactions on Software Engineering and Methodology*. October 1996.

[16] Tsun S. Chow. Testing Software Design Modeled by Finite State Machines. *IEEE Transactions on Software Engineering* v. SE-4, n. 3, Jan 1978. 178–86.

[17] Roong-Ko Doong and Phyllis G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology*. Volume 3 , Issue 2 (1994) Pages 101-130

[18] Matthew B. Dwyer and David A. Schmidt. Limiting State Explosion with Filter-Based Refinement. In *Proceedings of the ILPS'97 Workshop on Verification, Model Checking and Abstract Interpretation*, October, 1997.

[19] Electrical and Computer Engineering Dept. *ECE455 Software Engineering Course Project V1.4.* University of Waterloo, 1995.

[20] André Engels Loe Feijs Sjouke Mauw. Test Generation for Intelligent Networks Using Model Checking. *Tools and Algorithms for the Construction and Analysis of Systems, TACAS '97*, Lecture Notes in Computer Science 1217, Springer, 1997.

[21] Jean-Claude Fernandez. An Implementation of an Efficient Algorithm for Bisimulation Equivalence. *Science of Computer Programming*, volume 13, number 2-3, pages 219-236, May 1990.

[22] Jean-Claude Fernandez and Laurent Mounier. Verifying Bisimulations "On the Fly". *Proceedings of the 3rd International Conference on Formal Description Techniques FORTE'90* (Madrid, Spain), November 1990

[23] J. Fernandez, et al.. Using on-the-fly verification Techniques for the generation of test suites. In *Proceedings CAV '96*, LNCS 1102, Springer Verlag.

[24] Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Radu Mateescu, Laurent Mounier, and Mihaela Sighireanu. CADP: A Protocol Validation and Verification Toolbox. *Proceedings of the 8th Conference on Computer-Aided Verification* (New Brunswick, New Jersey, USA), pages 437-440, August 1996

[25] Jean-Claude Fernandez and Claude Jard and Thierry Jéron and Laurence Nedelka and César Viho. An Experiment in Automatic Generation of Test Suites for Protocols with Verification Technology. *Science Of Computer Programming*, v 29, n 1-2, pages 123-146, July 1997

[26] Jean-Claude Fernandez and Laurent Mounier. A Toolset for deciding Behavioral Equivalences. *Proceedings of CONCUR'91* (Amsterdam, The Netherlands), August 1991

[27] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems, An Approach to the State-Explosion Problem.* LNCS 1032, Springer-Verlag 1996.

[28] J. Grabowski, et al.. Test Case Generation with Test Purpose Specification by MSCs. *SDL'93 - Using Objects, in Proceedings 6th SDL Forum*, Darmstadt, Germany, October, 1993.

[29] J. Grabowski and R. Scheurer and Z. Dai and D. Hogrefe. Applying SAMSTAG to the B-ISDN Protocol SSCOP. *Proceedings of the IFIP International Workshop on Testing of Communicating Systems (IWTCS'97)*, Korea (1997)

[30] J-Ch. Grégoire. State space compression in SPIN with GETSs. *Second SPIN Workshop*, August 1996.

[31] Mazen Haj-Hussein, Luigi M. S. Logrippo, and Jacques Sincennes. Goal-oriented execution for LOTOS. In Michel Diaz and Roland Groz, editors, *Proc. Formal Description Techniques V*, pages 311-328. North-Holland, Amsterdam, Netherlands, October 1992.

[32] K. Havelund and K.G. Larsen. *StateText - A Textual Language for State Charts with Data.* Technical Report. Available as http://ic-www.arc.nasa.gov/ic/projects/amphion/people/havelund/Publications/statetext.ps.

[33] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall 1985

[34] G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall 1991.

[35] G.J. Holzmann and D. Peled. An Improvement in Formal Verification. *Proc. FORTE 1994 Conference*, Bern, Switzerland

[36] G.J. Holzmann. The Spin Model Checker. *IEEE Trans. on Software Engineering*, Vol. 23, No. 5, May 1997, pp. 279-295.

[37] G.J. Holzmann. Designing executable abstractions. *Proc. Formal Methods in Software Practice*, ACM Press, Clearwater Beach Florida USA, March 1998.

[38] G.J. Holzmann. The engineering of a model checker: the Gnu i-protocol case study revisited. *Proc. of the 6th Spin Workshop* LNCS, Vol. LNCS 1680, Springer Verlag, Toulouse France, Sept. 1999.

[39] Alan John Hu. *Efficient Techniques for Formal Verification Using Binary Decision Diagrams.* Ph.D. Thesis, Stanford University, December 1995.

[40] C. Norris Ip and David L. Dill. State Reduction Using Reversible Rules. *33nd Design Automation Conference*, pp. 564-567, June, 1996

[41] Hakim Kahlouche, César Viho, and Massimo Zendri. Hardware Testing using a Communication Protocol Conformance Testing Tool. *Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'99* (Amsterdam, The Netherlands), March 1999.

[42] Norris Ip and David L. Dill. Better Verification through Symmetry. *Formal Methods in System Design*, Volume 9, Numbers 1/2, pp 41-75, August 1996.

[43] L. Jagadeesan, C. Puchol, A. Porter, J. C. Ramming, and L. Votta. Specification-Based Testing of Reactive Software: Tools and Experiments, In the *Proceedings of the 19th International Conference on Software Engineering*. 1997. Boston, MA.

[44] K. Jensen. *Coloured Petri Nets. Volume 2, Analysis Methods*. Monographs in Theoretical Computer Science 575, Springer-Verlag 1992, pp.192-202.

[45] Paul C. Jorgensen. *Software Testing: A Craftsman's Approach*, CRC Press, 1995.

[46] Stefan Leue and Gerard Holzmann. v-Promela: A Visual, Object-Oriented Language for Spin. *Proceedings of the Second IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, 3rd.-5th. May 1999, Saint Malo, France, IEEE Computer Society Press, 1999.

[47] Johan Lilius and Ivan Porres Paltor. vUML: a Tool for Verifying UML Models. *TUCS Technical Report No. 272*, May 1999

[48] Jorn Lind-Nielsen. BuDDy - A Binary Decision Diagram Package. *Technical Report*, Department of Information Technology, Technical University of Denmark, 1999

[49] Wayne Liu. Interaction Abstraction for Compositional Finite State Systems. To appear in *Proceedings of The 7th International SPIN Workshop on Model Checking of Software*. Lecture Notes in Computer Science volume 1885, Springer-Verlag. September 2000.

[50] Wayne Liu and Paul Dasiewicz. Selecting System Test Cases for Object-oriented Programs Using Event-Flow. *Proceedings of the Canadian Conference on Electrical and Computer Engineering (CCECE '97)*

[51] Wayne Liu and Paul Dasiewicz. Extended TTCN in Software Testing. *Proceedings of the Canadian Conference on Electrical and Computer Engineering (CCECE '98)*

[52] Andrew Lyons. UML for Real-Time Overview. *RATIONAL Software Corporation Whitepaper*, April 1998, Available at http://www.rational.com/.

[53] David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan, Walter Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, Special Issue on Software Architecture, 21(4):336-355, April 1995.

[54] R. Milner. *Communication and Concurrency*. Prentice-Hall 1989

[55] OMG Document. *Action Semantics for the UML RFP*. Document ad/98-11-01. Available http://www.omg.org/.

[56] Jeff Offutt and Zhenyi Jin. Coupling-based Criteria for Integration Testing. *The Journal of Software Testing, Verification, and Reliability*, 8(3):133--154, September 1998.

[57] A. Puri and G.J. Holzmann. A Minimized automaton representation of reachable states. *Software Tools for Technology Transfer*, Vol. 3, No. 1, Springer Verlag.

[58] RATIONAL Software Corporation. *UML Notation Guide*. September 1997, Version 1.1, Available at http://www.rational.com/.

[59] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1998.

[60] J. Rumbaugh, I. Jacobson and G. Booch. *Unified Modeling Language Reference Manual*. Addison-Wesley, 1997.

[61] S. Sankar and R. Hayes, "*ADL: An Interface Language for Specifying and Testing Software*", Proceedings of the Workshop on Interface Definition Languages, January 1994.

[62] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-time Object-oriented Modelling.* Wiley, 1994.

[63] B. Steffen, S. Graf, G. Lüttgen "Compositional Minimization of Finite State Systems". *International Journal on Formal Aspects of Computing*, Vol. 8, pp. 607-616, 1996.

[64] J-P. Talpin. A. Benveniste, B. Caillaud, C. Jard, Z. Bouziane, and H. Canon. BDL, a Specifiation Language for Distributed Object-Oriented Real-Time Systems. In *Proceedings of the International Symposium on Object-Oriented Real-Time distributed Computing ISORC'98*, 1998.

[65] D. Toggweiler. Efficient Test Case Generation for distributed Systems specified by Automata. *PhD Thesis, University of Berne, Institute for Informatics and Applied Mathematics* (1995)

[66] A. Valmari. The State Explosion Problem. *Lectures on Petri Nets I: Basic Models*, Lecture Notes in Computer Science 1491, Springer-Verlag 1998, pp. 429-528

[67] A. Valmari. *State Space Generation: Efficiency and Practicality.* PhD Thesis, Tampere University of Technology Publications 55, 1988.

[68] Kimmo Varpaaniemi, Jaakko Halme, Kari Hiekkanen, and Tino Pyssysalo. *PROD reference manual.* Technical Report B13, Helsinki University of Technology, Digital Systems Laboratory, Espoo, Finland, August 1995.