Using Modal Logic Proofs to test Implementation-Specification Relations

Alan Paxton

Doctor of Philosophy
University of Edinburgh
2000

Abstract

This thesis shows how to make use of the intensional information relating specifications to implementations. It views the proofs of properties of specifications as identifying the intensional parts of implementations relevant to the property. It provides a concrete instance of such proofs by adopting labelled transition systems, modal-mu calculus and the tableau methods of Stirling and Bradfield as a framework for generating intensional information.

The intensional information generated from proofs about models of systems can be used to verify behaviours of implementations of systems. By annotating implementations of systems with the atomic actions of their models we can apply oracle techniques to verifying implementation behaviour. The extra richness of intensional information allows oracles derived from proofs, rather than just from properties, to be much more discriminating of failures in the implementation.

The emphasis of oracle-based testing and verification is on practical improvements in the quality of distributed systems. Therefore the intensional idea is developed into a framework for a practical system. Case study systems are examined to identify where system developers can be helped by computerised systems to integrate auditioning into the software development process.

Acknowledgements

I must first thank my supervisor, Stuart Anderson, whose unfailing enthusiasm and willingness to help despite his permanent condition of overwork were instrumental in my finishing at all. His ideas were the beginning.

Peter Hancock, Russ Green, Chris Davies and Ian Pattison of Digital provided me with much help in the case study work, and Russell Robles originally persuaded the company to take part in the grant. Peter also stands as a great moral and intellectual example.

Antonella Bertolino of IEI (CNR) was my host and mentor in Pisa; she contributed greatly to my understanding of testing and oracles, and helped greatly to facilitate my stay.

Most of all I thank Hazel Christie for her love, support and humanity.

To all the others who helped in some way, my thanks, and my apologies for the lack of a personal mention.

I was financed by an SERC (CASE) Studentship jointly with Digital Equipment Co. and my sixth months in IEI (CNR) Pisa were paid for by a Young Researcher grant from the EU OLOS Network.

Declaration

I declare that this thesis was composed by myself and that the work contained therein is my own, except where explicitly stated otherwise in the text.

(*Alan Paxton*)

Table of Contents

Chapter 1

Introduction

In this chapter I describe the background and motivation for this thesis, introduce the concept of program auditioning which is at its core, and sketch the structure of the rest of the thesis.

1.1 Verifying Distributed Computing Systems

Large, distributed computer systems are amongst the most complex artifacts built by human beings. It is natural therefore that we have to ask whether they behave correctly. The question can take many forms, and answering it has been a major concern of computing research for many years [MC84].

In order to claim that a system is correct, we must know what constitutes correctness. Sometimes this is obvious; we have a reasonable idea of what a simple piece of hardware such as an adder or multiplier must do. So we can at least define a set of tests for it and say what it is for the tests to be passed. But even here there are deep questions about what tests are sufficient to support the claim that the hardware must be correct.

In distributed systems even defining correctness is not so simple. Distributed systems can most fruitfully be analysed reactively; each component's behaviour is just a response to the behaviour of the other components of the system. Ultimately the external behaviour responded to is a model of the *outside world.* But viewing a distributed system as a function from input to output gives no insight into the distribution. Some notion of correct behaviours is required, and this can broadly be termed a language. More concretely, these languages are *specifications*, and relationships are defined between systems and specifications. Correctness is precisely expressed in terms of which relationships hold between system and specification. Many different specification languages and correctness relationships exist, and different instances are suitable to different domains.

The process of model-checking provides an example. A particular distributed system may be expressible as an initial state of a Labelled Transition System (LTS). A modal logic such as CTL or CTL* expresses properties of LTSs. For a

finite LTS, algorithms exist to search the state space and determine for which states a property holds. The property holds when and only when the initial state is in the resultant state set.

Although clever representational techniques such as BDDs [Bry92] have made it possible to model check quite large pieces of hardware, these still lie at the smaller end of the size spectrum of computing *systems*. For many a larger system the problem of formally modelling the implementation is huge [Bow95, Phi90]. Proving that it satisfies a specification expressing its behaviour, or an aspect of its behaviour is therefore still impractical. To write even a specification of such a system is an onerous task [Win]. To do it, we must choose a useful level of abstraction in the system in question at which to work, and we must make assumptions in the specification about how our system interacts with surrounding systems. For instance, the filesystem component of an operating system must interface with a disk driver and a memory manager. So any proof we make about the *specification* is contingent on our having made the correct assumptions about the interacting systems. And if the other components are not formally verified we cannot know that our assumptions hold.

This size-based criticism applies equally to any fully formal method, not just model-checking. Fully formal software is more usually developed formally, by way of stepwise refinement from specification to implementation [KST94].

It is reasonable then to assume that full formality is not yet usable in large systems. Since I am particularly interested here in large systems, the question stands as to what can usefully be done with such a system to ensure that it is acceptably robust. Present industrial practice involves an array of useful procedures:

Testing encodes the programmer's implicit understanding of the specification as a prediction of the system's behaviour under particular inputs. It often reveals problems but, pace Dijkstra [Dij76], it does not guarantee their absence.

Writing Specifications alone is a useful discipline for programmers. The thinking involved in expressing a system rigorously in a way other than in the programming, can often be enough to reveal deep problems. Like testing it suffers from the fact that it is necessarily not exhaustive.

Peer Review of design and coding has in practice proved to have reliability benefits which can at least partly be ascribed to forcing a programmer to write argumentatively *justifiable* code.

While Dijkstra's criticism is formally valid, it can be overemphasised. While the attempt to develop methods to derive fully correct programs can still be considered long-term research, there is an important role to be played by other

methodologies which tend to improve the reliability of software, at a realistic cost.

My aim in this thesis is to present a method which spans the large gap between these useful but mathematically informal methods and any of the methods which demands formalisation of the implementation. Viewing system development as moving from small and very abstract expressions of the system (specifications derived from requirements) through a series of refinement steps to large and very concrete expressions (implementations), I show how to use formal proofs of relationships at the upper levels to test and monitor the implementation.

1.2 Program Auditioning

The germs of an intermediate approach can be seen in the work of [JLSU87, Bat95, CD95, CG95]. These authors have experimented with testing mechanisms where the most direct expressions of system behaviour are adopted as languages with which to define and test system correctness. An architecture has been defined by [OAR92] which encompasses their approach and the others, and into which this work also fits. We describe it thus

Definition 1.1 (Oracle-based testing/verification) How to do it:

- Infiltrate software probes into a system implementation which signal important events in the running system (*annotation*).
- Decide which are acceptable behaviours of the system in terms of these events, and which are unacceptable.
- Express these decisions in an *oracle* system. An oracle is a system logically independent from the system under observation, which watches the probes and determines whether the observed behaviour is acceptable.

Where earlier uses of annotated systems have differed is in the definition of the events in the systems which are considered important. Some systems [JLSU87, Bat95, CD95, CG95] have an informal notion of specifications, and express properties directly as sequences of events. Others [DR96] use a modal logic to define properties and derive the necessary annotations from the properties.

In our variant, which we term *auditioning*, we seek to derive the checkable behaviours from *proofs about the specification* of the system. The motivation is that proofs encode exactly the relevant information as to how properties are satisfied, while logical formulae alone ignore this information. If this is the case, then auditioning will provide a more discerning test of the correctness of distributed systems.

The elements needed to audition a system are:

Specification of the system or subsystem. This is expressed as a transition system relating specification states by named actions.

Proof of a property or properties of that system, for which I choose to use modal-logic and tableaux.

Implementation of the system.

It is the job of the user to mark up the implementation according to where she understands the actions of the specification are simulated. This involves her in developing some conceptual relationship between the specification and the implementation. Clearly such a relationship ought to exist, and the assumption that it does lies behind all annotation systems. But it is not clear what the relationship should be: refinement ? some form of simulation ? By choosing to study a proof-based system we are led to ask, in particular, whether and how logical properties and their proofs can be used as the expression of acceptable behaviours, and how clear the relationship between specification and implementation is when we use this expression. So we might say

Formal annotation consists of

properties combined with *proofs* to construct *oracles*

From an oracle we are able to produce two artefacts as a contribution to the verification process. These are a recorder of implementation actions for post-hoc analysis and a monitor of implementation actions for immediate analysis. Monitoring is appropriate where we want to check in real-time the operation of a safety-critical system and take automatic or operator-initiated remedial action when it behaves incorrectly. Recording is appropriate where we want, after the fact, to discover the root cause of a failure in a system under development (testing, debugging).

What is novel about auditioning is that although it makes use of formal proof and specification it does not require a major reversal of the way software is developed. This may allow it to be adopted in industrial software development more readily than classic fully-formal top-down development [KST94], and even to be used as a horse to carry specification into Troy. The contrast between this and the more usual way of using proof reveals the tradeoff between total rigour and practicality:

- Formal refinement from specification to implementation gives us complete confidence in the correctness of our final implementation in as much as we trust the refinement methodology and meta-theory. Auditioning provides no such guarantees, only the increase in confidence formerly suggested.

- Refinement requires a strictly ordered development process which yields an implementation as a final artifact. Auditioning can be carried out on an

already implemented system by writing a specification of the appropriate form and proving appropriate properties.

- Formal annotation can proceed in an incremental manner. Proving and annotating more properties provides more security for an implemented system.

It is still the case that many software development efforts are unable or unwilling to make use of fully-formal development. Such development necessitates a particular style and process of working which can be alien to current practitioners in the industry. Thus there is a vital pragmatic advantage in a technique which can be adopted without a wholesale re-organisation of process.

1.3 Thesis Outline

This thesis is structured in two threads. In the first I examine the place of auditioning as a tool in the development of computing systems. I describe a methodology for doing auditioning and the infrastructure necessary for it.

In the second thread I consider modal logic, and proof, more formally. The application of proof in auditioning provokes reflection on encodings of modal-logic proofs. Semantic-tableaux as a notation for writing proofs and model-checking games as a metaphor for understanding modal logics are related in well-known ways; I recall these and introduce another proof notation, *verities*, designed to be used for auditioning. I consider whether verities are of interest themselves; they emphasise the static content of proofs and avoid the directedness of proving inherent in tableaux.

The two threads come together when I describe the formalities of auditioning, using verities. With the formal background in place I present a larger example and move on to consider how auditioning might be made more usable through the development of supporting tools.

In Chapter 2 I start the first thread by reviewing the state of research in test oracles, and analyse the structure of the oracle mechanism. I review the complementary notions of intensionality and extensionality. Proofs encapsulate intension, and I motivate the use of proof-based oracles by explaining the way in which the move from specification to implementation necessarily introduces the intensional.

Subsequently, in Chapter 3, I work through the mechanics of auditioning a system, and illustrate the main points of the process by means of a small example. At this stage I necessarily keep the exposition informal; the intent is to provide a broad understanding of the process which motivates the questions dealt with in detail in later chapters. Background, meaning the CCS specification language [Mil89] and the mu-calculus formal logic, are sketched here in an informal way; just enough to allow the reader to follow the example.

In Chapter 4 I describe the previously sketched background in the detail necessary for formal treatment. Taking the mu-calculus as exemplary among process logics, I review its labelled-transition-system semantics, with particular emphasis on the fixpoints and ordinal approximants as these bear most heavily on the intensionality of auditioning. I recall the Calculus of Communicating Systems (CCS [Mil89]), and its use as a language of LTS systems. I briefly point out some alternative logics to the modal mu-calculus, and rehearse the advantages of a logic-based approach to specification and refinement, especially noting that it conforms with the asymmetric refinement-based view which the methodology takes. Finally I look at Bradfield's [Bra91] tableau system, which is a natural way in which to develop proofs of properties. I reprise the formal definition for and basic results about this system, in preparation for presenting a translation into verities in Chapter 6.

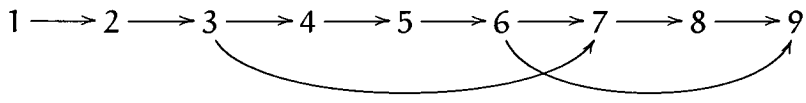
In Chapter 5 I continue the formal theme by looking at formal representations of proofs. The chapter concentrates on model-checking games, or more precisely strategies for winning them. The links between tableaux and strategies for winning games are clear. I present my alternative view of a proof, the *verity*, and link it to the other views.

The immediate importance of verities is to serve as the object from which oracles are generated. In Chapter 6 I describe the details of this process. This is the formal counterpart of Chapter 3, and I examine the steps involved in converting verities into the clearly mechanisable form necessary to run an oracle. I show that the oracle process so generated correctly permits *correct* behaviours and detects important classes of incorrect behaviours. I also look at some possible simplifications which may be applied in the oracle generation process and which present the user of an auditioning system with choices about trading-off the power of an oracle against its smallness.

Having described the process of oracle generation in its entirety, in Chapter 7 I present a larger example in more formal detail. This in turn raises questions of what mechanical support is possible for the methodology, and Chapter 8 presents several ways in which the process can be eased. These include possible alternative characterisations of CCS which help to treat infinite families of processes, and mechanical extension of *partial-proofs* which provide a way of supporting the human development of proofs.

I conclude by reflecting on where the two threads may lead. For the practical thread I ask what work is necessary to produce an industrially usable auditioning system and whether such a system might prove useful in practice. For the formal thread I ask whether further work with verities could be a fruitful avenue for study of the modal mu-calculus.

Figure 1.1 Flow of Chapters



Chapter 2

Oracles and Intensionality

Summary

In this chapter I lay the ground for my thesis in more depth. In Section 2.1, I contrast testing and formal verification, and argue for a pragmatic combination of the two.

An oracle is a device for automatically validating that a system's behaviour is as expected. If we can construct a powerful oracle for a distributed system, we go a long way towards easing the testing/monitoring problem. I describe how oracles have put testing into a more formal framework, and review the research that has been done into generating oracles from logical properties.

In Section 2.2 I discuss the concept of extensionality versus intensionality, constrasting what a property says (extensional) with how a proof shows that a property holds (intensional). This provides a theoretical framework for the oracles from proofs method. I show why, using proofs of formal logical properties, we can expect to construct richer and hence more useful oracles than can be constructed from logical properties alone. I view system development as adding an intensional implementation to an extensional specification. In this view, verification naturally requires selective use of the intensional implementation information.

In Section 2.3 I examine the practical question of how to relate the behaviour of a software program to the behaviour of a formal specification; after this I discuss the forms that the formal specification can take. In the context of the auditioning framework I use the term *abstract implementation* to distinguish between this and a set of one or more logical properties which are sometimes termed a specification. Finally, Section 2.5 steps back to look at the overall picture.

2.1 Verification and Testing

The complexity of distributed systems means that it is difficult to know when they are behaving correctly. This makes testing and monitoring especially difficult.

Testing and proving are often viewed as alternative approaches to software validation, selected according to a tradeoff between the *correctness* of the software and the *cost* of development. The more enlightened rightly reject this view and make intelligent ad-hoc use of both approaches in the same project, viewing each as a useful, but not universal, tool for constructing software which satisfies its requirements.

An explicit structure for software development can encompass both of:

1. Proving properties of specifications
2. Testing using oracles on traces/output

I will show that the two together can be used to cross some of the gaps in the hierarchy of abstractions which characterises the refinement from specification to implementation.

2.1.1 Formal Specification and Proof

The formal methods community has extensively researched methods for developing software rigorously from first specification of requirements in formal language, through multiple stages of refinement (preserving properties at each stage) to the production of an implementation which consequently provably meets the requirements at the top level [KST94, BH95, Bow95]. However such development is still too onerous to employ in a large project, and its use remains more of a research goal than a reality. Restricted use of formal methods, proving limited properties about interesting fragments of large systems has however been used succesfully in large software engineering projects [Phi90, BH95].

Another criticism of formal development is to point out that performance requirements are not captured within the system. Although performance can be considered separately from *correctness*, the heavily structured nature of formal development imposes more work in the redesign of algorithms which is usually needed to make significant performance improvements.

2.1.2 Testing and Oracles

In contrast to proving, testing is an immediately practical approach to software validation. A tester asks herself what behaviour (or output) the system should perform under given external conditions (or input). She then establishes these external conditions and checks that the behaviour is as she predicted. Intelligently carried out, this is a productive approach, but there are problems

- Testing gravitates towards the most readily testable components
- It is difficult to conclude that testing is thorough, that every component has been properly examined, and that none of the matrix of possible combinations of components yields strange interactions.

The testing community has studied how to make tests [Bei90] which usefully cover the behaviours of the system under consideration, more rigorous [BDZ89]. While effort has concentrated on defining a collection of tests which is in some sense *complete* for the system under test, verifying that the behaviour of the system while running the test is as prescribed has been neglected [Wey82]. It has too often been assumed, without justification, that failing behaviour can be easily distinguished from correct behaviour. Oracles have been proposed as an approach to make the acceptance/rejection criteria for test more rigorous.

Several systems have been developed in which the behaviour of systems under test is made discrete and divided into the acceptable and the known-to-be-incorrect [CG95, JLSU87, CD95, Bat95] Richardson et. al. [OAR92] have attempted to put things on a more rigorous basis by defining an architectural structure in which the language of specifications, implementations and oracles are related through a series of mappings. They also demonstrate some case studies using this framework.

With the architecture defined it is possible to think more formally about the components. Dillon et. al. have defined a general technique for constructing oracles from a class of logics [DR96], in particular applying it [ORD96] to their own temporal logic, GIL [DKMMS94], which has an attractive pictorial presentation. Similar approaches have been taken by others; for example [ABG96] describes a system which automatically checks test outputs against safety requirements specified in the logic ACTL [DV90].

Although these systems have demonstrated practical systems for automatic checking, some problems remain. One is that the logics used tend not to be standard. Another problem is that the purely logical expression of acceptable behaviours can be too weak to discover many faulty systems. Systems which are incorrect can always fail to produce traces which are conclusive evidence of incorrectness: empty traces produced by deadlocked systems can always be viewed as prefixes of other (more obviously acceptable) traces. Deriving an oracle from a logic immediately restricts the power of testing to mirror the power of the logic. If the difference between two system behaviours is not expressible in the logic, then that difference cannot be tested for.

In particular, external behaviour may be correct only relative to the internal state. To write finer properties just to distinguish between states of an implementation is to subvert the idea of using a logic in the first place: much better to be explicit about state where we have to be. So we need to

- Be pragmatic, and work in a standard logic.

- Reduce the gap between the system not working and the oracle declaring this fact by finding a place for system states in our framework.

2.1.3 Integration of Proof and Testing

Auditioning uses techniques from formal methods and oracle-based testing. A developer uses a proof system for a formal logic to construct a proof that an abstract implementation has certain logical properties. She annotates the (concrete) implementation to reflect the behaviour of the abstract implementation. The auditioning system provides her with an oracle which encapsulates the information in her proof and she uses it to verify the absence of wrong behaviour in the running implementation.

The oracle obtained contains information about the structure of the system specification, where it is relevant to the property proved. Relevance of information is established just because it appears in the formal proof. As the system specification changes the proof must be revised, but from this a new and different oracle can automatically be derived. The reason that a property-only oracle does not need to change is precisely that it cannot distinguish between any two system specifications.

The result is an oracle with finer coverage of the system behaviour, compared to a property-only oracle, but with a size penalty which is local rather than global.

2.2 Extensional and Intensional Oracles

It is the incorporation into the oracle of exactly the system-specific information relevant to the proved properties which distinguishes auditioning from earlier work on oracles. Thus this approach can be labelled intensional, while others are extensional. Extensionality is suggested by the word *what*, and intensionality by the word *how*. Using only a property (such as a logical formula) to derive an oracle encapsulates no knowledge about the system under investigation. Knowing how the property is satisfied, in terms of states of the system and subproperties which they in turn satisfy, results in an oracle which encapsulates all the knowledge about the system relevant to the property. And a proof of this property for the system in question contains exactly this knowledge.

I contend that the level of intensional detail provided by a proof may be useful in closing the *abstraction gap* between specification and implementation. If we write proofs about the implementation we take on the work of fully-formal development, and gain nothing from generating oracles for testing already-proven properties. Instead I suggest a framework in which the move from specification to implementation can be seen in two stages:

High Level Where properties, often expressed in a modal or temporal logic, but in any case having an extensional flavour, are refined to the abstract implementation. The specification still has a high level of programming abstraction, but unlike the properties it involves a model which typically distinguishes several components and describes how they interact. This is canonically intensional, and the model is often expressed in a language such as CCS or CSP.

It has been suggested that *software architecture* [PW92, SG96] is the description of how components fit together, rather than how the components are constructed. In these terms such a specification is indeed architectural. But it is probably wrong to set too much store on this assertion, because the phenomenom is manifest at many levels; each higher level of abstraction wires up the components at the previous level, and provides a new set of super-components which are just the collection of components and their interaction.

Low Level Is where the abstract implementation is refined further to produce the implementation. Although significant programming tasks must be carried out here, and complex demands such as efficiency and maintainability must be reconciled, something of the structure or architecture is carried through from the abstract implementation.

In drawing this conclusion I in no way seek to deprecate the intellectual tasks performed by the programmer. Her work is surely that of reflecting the abstract system in the messy and complicated world of runnable software. But it is the key to correctness that this reflection does happen, and that we therefore have some concept of how to judge correctness.

In the two-level framework we can write proofs at the high level which show that abstract implementations satisfy certain properties. Auditioning provides a process by which these high-level proofs are bundled up to generate an oracle which is perhaps larger than one generated by the GIL-algorithm [ORD96] but should be more discriminating in its identification of aberrant behaviour. And there is still a degree of flexibility in deciding at how abstract a level to stop developing the abstract implementation. More elaboration and hence more detailed oracles must be traded off against the work required.

The expectation is that the extra level of discrimination in the oracle will lead to practical improvements in the recognition of failures in the system. Of course such claims will need to be backed up with experience, but notice for example that a deadlock can never be detected by a purely extensional oracle. Figure 2.2 shows the different approaches to development and how they bridge the abstraction/refinement chasm.

Figure 2.1 Legend for subsequent figures

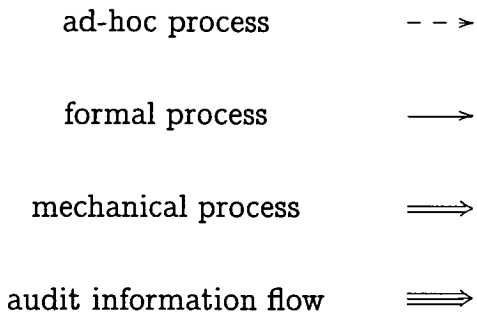
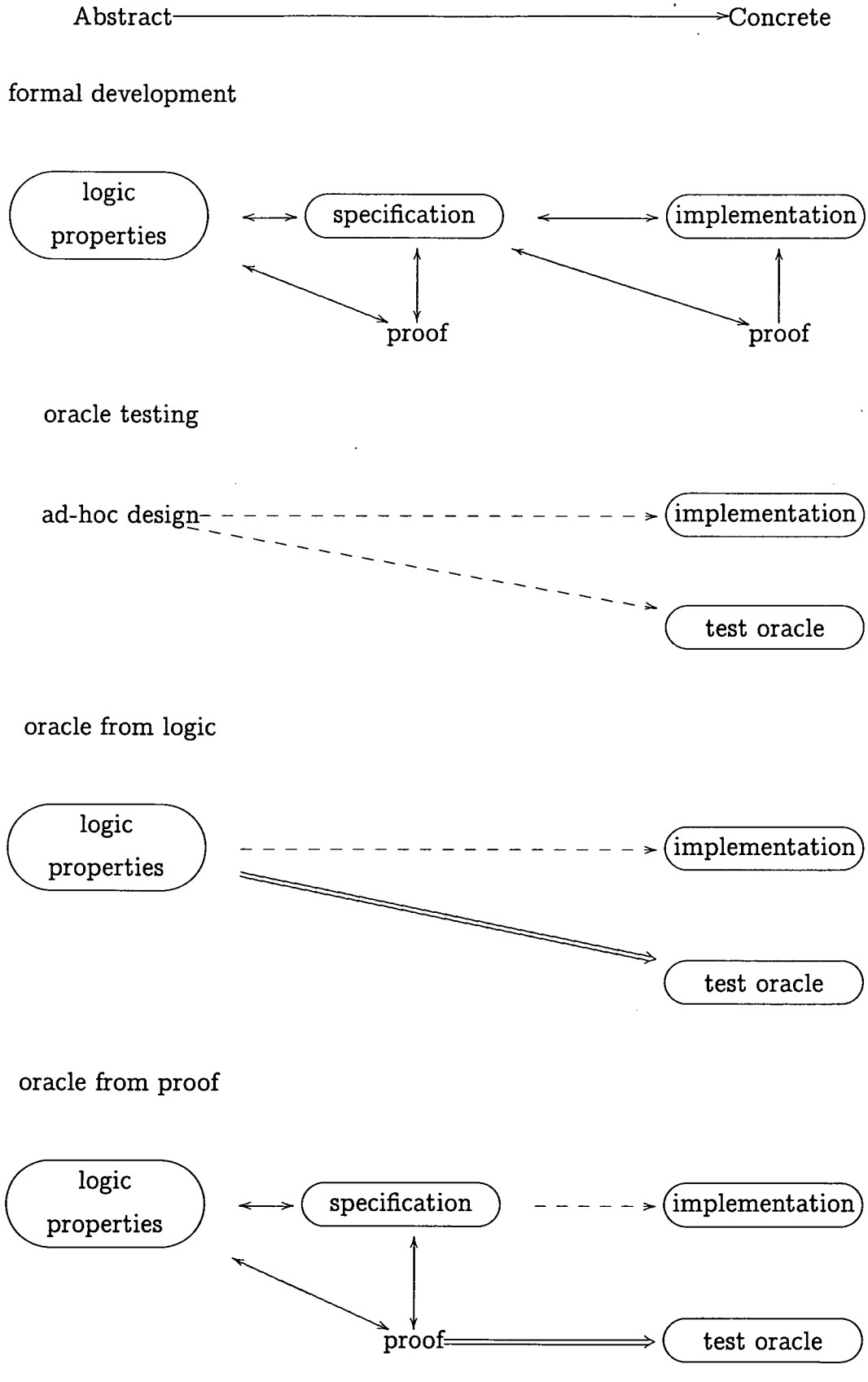


Figure 2.2 From abstract to concrete



To summarise, the approach is to split the difference between formal development and testing. The developer has to produce specification(s) at various abstract implementation levels, of the whole, or interesting parts, of the system, and to annotate the implementation to allow the oracle to make the link with the proof. The rest of the chapter covers these steps in more detail.

2.3 Annotating the Implementation

In order to provide the information necessary to drive the oracle, the implementation must be decorated with *transition callouts*. A transition callout is a code stub which communicates with a listener (the oracle) and says in effect

> Consider an α-action (where α is a parameter of the callout) to have taken place.

The code is annotated by the insertion of transition callouts. We read them as declaring certain points in the code to be equivalent to transitions of the abstract implementation. More generally, extra code can be added to the implementation to compute complicated conditions for these callouts; multiple points may declare the same action if, for instance, each calls a runtime library procedure which is considered to implement the transition. Any run of an annotated system generates a sequence of transition callouts (a trace) which the oracle can choose to accept, reject or hold counsel on.

The only constraint introduced at this stage is that the annotation must be done relative to a specific abstract implementation. Of course the annotation is a manual process which requires knowledge of both the implementation and the specification, but the developer should hold this knowledge and with some thought be able to express it in the required form.

Annotation is done in an ad-hoc fashion because we have no theory of the structure of such abstractions and the languages of implementations are in any case too diverse to impose one. The result is that a 'C'-program will be annotated by writing a series of extra functions which observe its internal state and make call-outs to the observer of actions entirely at the discretion of the implementer. But there is no way round this. If we were willing to pay the costs to make a formal relationship between the specification and the abstract implementation we would not need to test the relationship, with oracles or in any other way.

2.4 The Abstract Implementation

The way that the abstract implementation is written can be flexible. An oracle can be produced from any description of a set of rejectable and acceptable

traces of the abstract implementation. In fact we can do this with any labelled transition system (LTS), together with a language of traces for the LTS. But to justify the importance given to proof-derived oracles we must see how to generate an oracle from a proof about a system. This requires the choice of a logic and a proof formalism consistent with the labelled transition systems used to write the specification. Within these constraints we could imagine using any number of formalisms. In this thesis I use the modal-mu calculus [Koz83] for logic because it is sufficiently expressive to raise deep questions about repetition and termination for the oracle generation algorithm we need to develop. Then semantic-tableaux are the natural proof mechanisms for this logic. Other logics/mechanisms would fit, the point is entirely that a system exploiting these ideas could be developed for the notations with which the target user group is familiar.

The job of the developer using the auditioning framework is first to write specification(s) of the whole, or interesting parts, of her system, abstracting away from the *irrelevant* parts of the system. It is for the writer of specifications to decide what is relevant, and we don't claim it is trivial to do this.

With the specification (or abstract implementation) and a suitable logic, important properties can be stated. These are informal properties of the large system, expressed formally as properties of the specification. The abstractness, and hence reduced size, of the specification should make the task of proving properties a reasonably routine one for a human developer familiar with the design, as the case studies (Chapter 7) suggest. These proofs turn out not to be deep; they are mostly expressions of implicit knowledge already held by the designer. In fact it is possible to envisage writing many abstract implementations which further abstract subcomponents of less interest for particular properties, and thus further simplify the job of writing proofs.

2.5 Putting Things Together

Assuming that a set of proofs can be translated into acceptance/rejection rules for an oracle, let's consider the overall structure.

An oracle in our framework examines traces generated by the annotated implementation for consistency with those generated from proofs about the specification. Thus the check carried out by the oracle is really a check of the translation from abstract to concrete implementation. An oracle will always accept all traces from the abstract implementation from which it is generated, though it may reject traces from a different abstract implementation satisfying the same formula(e).

A correct translation from specification to implementation, correctly annotated, will of course result in the oracle accepting any behaviour of the implementation. Less appealingly, an incorrect translation will only necessarily

result in an oracle rejecting some behaviour if the oracle is generated from the proof of a particular property. In any case, we must drive the system so as to provoke this behaviour. That is, we must still make good test case selections. Test case selection is a crucial and difficult problem in testing, and we have to accept that however smart an oracle is, the quality of testing inevitably relies on a sufficiently powerful set of test cases to drive the system into enough places where the oracle can analyse its behaviour.

The verification and testing methodology is summarised in Figure 2.3. It shows the major steps involved in testing with an intensional oracle. We write Spec $\models$ Property for a property holding of a specification. The annotation operation is expressed here as the arrows from specification and implementation to annotated implementation.

Figure 2.3 Overall structure

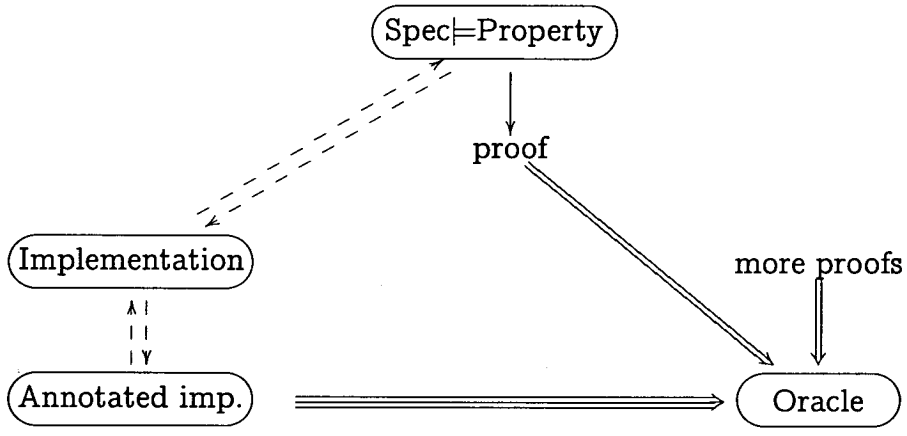
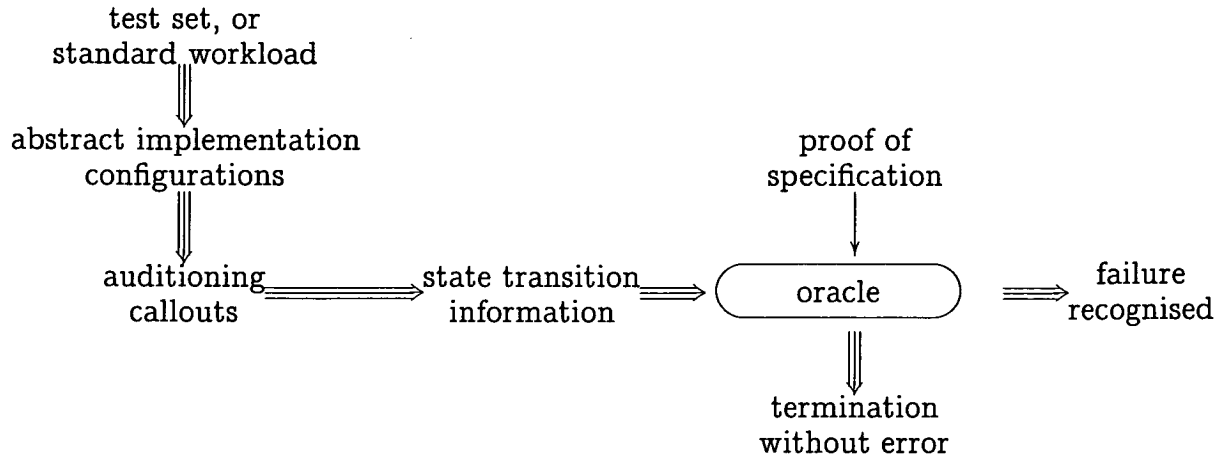


Figure 2.4 shows the dynamic behaviour of a running test, where the oracle extends partial trace information as it receives state changes from the annotated implementation, and at each extension checks that the partial trace is part of some admissible total trace as described by the proof(s) encapsulated within it.

Figure 2.4 Running an oracle



2.6 Interpreting the Oracle's Pronouncements

We must understand what the response of the oracle to a particular execution means. Clearly the lack of false-positives ensured by a correct implementation of the oracle generation is necessary. But precisely because of the intensional nature of the proof, the discovery of a violation by the oracle does not mean that the implementation does not satisfy the property. It does mean that the implementation does not behave in the same way as the specification, with respect to the property, which is at least as bad an error in a properly engineered system.

Although we have a strong informal sense of what a statement of the form $I \models \phi$ means, when a property holds in the implementation, making it formal is only possible through the whole process of formal development, because ϕ is really a property of the abstract implementation. So we use the informal sense below. We write $I \succ_\phi S$ when the implementation I is a correct refinement of the specification/abstract implementation S with respect to the property in question. Of course the $\succ$ relation depends on the notional formal refinement of S to I too. Nevertheless, we can examine how the oracle behaves, given a perfect test set, in all cases (Ω stands for the oracle).

$I \models \phi$	$I \succ_\phi S$	Ω
yes	yes	accepts
no	no	rejects
yes	no	rejects
no	yes	impossible

The third line reflects rejections by the oracle of implementations which do not match the specification, and detecting such failures cannot be done with an

extensional oracle.

2.7 Conclusion

I have suggested a taxonomy of correctness assurance in software development where oracle based testing lies between formal development and ad-hoc development and testing. I have looked at where current ideas break down; deriving oracles from logical properties results in somewhat weak oracles. My suggested solution is to generate oracles from proofs, which embody more information than properties alone. I have outlined a mechanism for doing this; the mechanism involves more work than for property-based oracles, but promises better results from more discerning oracles.

Chapter 3

Example

Summary

Having discussed in general terms the motivation for, and structure of, auditioning, I now make things more concrete by working through an example. The chapter starts by describing a small example system (Section 3.1). I then introduce the minimum amount of specification language (Section 3.2) to present a formal specification of the example system. Describing modal logic (Section 3.3) allows expression of properties of the system, and the tableau proof system (Section 3.4) allows the development of proofs of important properties of the system.

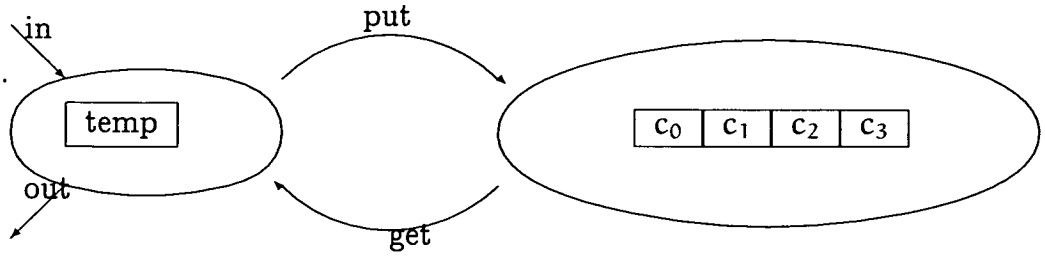
There may appear to be a lot of formal material here, but this chapter contains all the material necessary to proceed to realistic work on large examples, such as that of Chapter 7, and as little beyond that as possible. Later chapters in the formal stream are a comprehensive treatment of the formal derivation of oracles, but are not required for practitioners.

The remainder of the chapter (Section 3.5) is a discussion of how an oracle for the example system is derived from the proofs, and of some of the issues raised by the derivation. With the resulting example oracle it becomes possible to motivate some generally useful transformations and simplifications of oracles.

3.1 Compressing Server

The system is derived loosely from the compressing proxy of [CIW97]. The essence which we derive from it is an input/output process which presents an abstract datatype (ADT) to the outside world and a server process which applies a function to blocks of data input to it. In this case we fix the buffer size of the server to make the analysis tractable; it can clearly be generalised. Figure 3.1 shows this graphically.

Figure 3.1 Compressing Server



The ports *in* and *out* are the external communication ports between the IO system and the outside world. The ports *put* and *get* are internal communication ports to the server part. When a client of the compressing server wants to pass a block for compression it offers it on $\overline{\text{in}}$. the system accepts the block on this port and then both system and outside world continue on their way.

When a client wants to retrieve the next compressed block it accepts it on **out**.

3.2 A Little Gentle CCS

We need to introduce the language CCS [Mil89] in order to write a formal specification of the compressing server. CCS expressions are formed from atomic actions a, the null process **nil** which can perform no action, and various combinators $(|, +, a.P, \ldots)$. Variable names can stand for expressions, which allows recursion.

Processes are understood by their behaviour. The most basic rule is that an action-prefixed process can perform the prefix action and yield the suffix process;

$$a.P \xrightarrow{a} P$$

A sum of processes can choose which to behave like

$$a.P + b.Q \xrightarrow{a} P \text{ but equally } a.P + b.Q \xrightarrow{b} Q$$

Parallel composed processes behave as independent processes placed side-by-side.

$$a.P \mid b.Q \xrightarrow{a} P \mid b.Q \text{ and equally } a.P \mid b.Q \xrightarrow{b} a.P \mid Q$$

but they can interact in the special case where their actions are complements. In that case the interaction is recorded through the special τ (communication) action.

$$a.P \mid \overline{a}.Q \xrightarrow{\tau} P \mid Q$$

The $P\backslash\mathcal{L}$ construction restricts the behaviours of P to those in the set $\mathcal{L}$ (but τ can never be restricted) and P[f] renames actions in P so that

$$P[f] \xrightarrow{f(a)} P'[f] \text{ iff } P \xrightarrow{a} P'$$

The compressing server is specified in CCS as $(\mathbf{C}_0 \mid \mathbf{IO}) \setminus \{\text{put}, \text{get}\}$, where

$$\begin{aligned}
\mathbf{IO} &= \text{in}.\mathbf{I}_1 + \text{get}.\overline{\text{out}}.\mathbf{IO} \\
\mathbf{I}_1 &= \overline{\text{put}}.\mathbf{IO} + \text{get}.\overline{\text{out}}.\mathbf{I}_1
\end{aligned}$$

$$\begin{aligned}
\mathbf{C}_0 &= \text{put}.\mathbf{C}_1 \\
\mathbf{C}_i &= \text{put}.\mathbf{C}_{i+1} + \overline{\text{get}}.\mathbf{C}_{i-1} (0 < i < n) \\
\mathbf{C}_n &= \overline{\text{get}}.\mathbf{C}_{n-1}
\end{aligned}$$

This, together with the rules of CCS defines a labelled transition system, and we can instantiate $n > 0$ to yield an n-place buffer. We'll use $n = 4$ in the example. n limits the number of blocks which can be being processed simultaneously. Even if all n buffers become full and input is held up, we want the system not to deadlock but to finally clear the backlog and accept new inputs. A correct design must always ensure this happens.

Notice that in this specification the data to be compressed is not explicitly modelled. Our specification covers only the interaction of the processing components. This choice embodies the implicit assumption that the correctness of the block compression function is dealt with sufficiently elsewhere. So we can abstract from it and leave ourselves with the specification above, which is sufficient to study control interaction yet small enough to easily write proofs about.

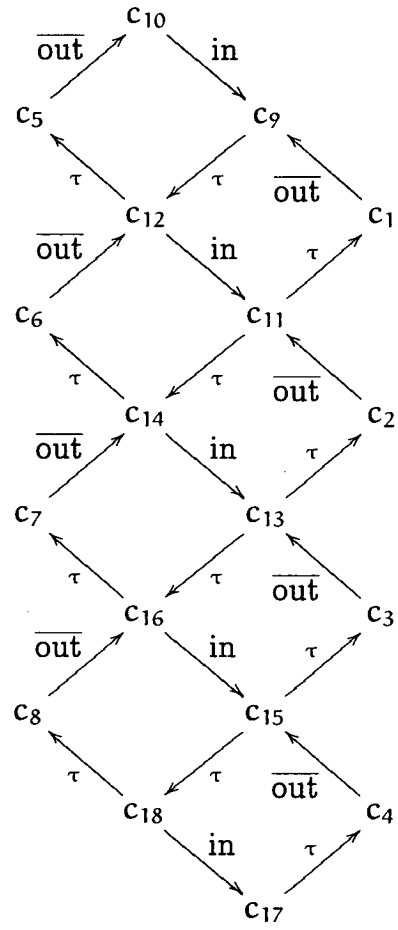
A labelled transition system (LTS) is just a set of states, connected by named arrows. We view a state of an LTS as representing a state of the running system, and an arrow of a particular name exists between a pair of states whenever the system can be transformed from the first state to the second by the action of that name.

The LTS of this CCS specification has 18 states

$$\begin{aligned}
c_{18} &= \text{IO} \mid \text{C4} \setminus \{\text{put}, \text{get}\} & c_{17} &= \text{I1} \mid \text{C4} \setminus \{\text{put}, \text{get}\} \\
c_{16} &= \text{IO} \mid \text{C3} \setminus \{\text{put}, \text{get}\} & c_{15} &= \text{I1} \mid \text{C3} \setminus \{\text{put}, \text{get}\} \\
c_{14} &= \text{IO} \mid \text{C2} \setminus \{\text{put}, \text{get}\} & c_{13} &= \text{I1} \mid \text{C2} \setminus \{\text{put}, \text{get}\} \\
c_{12} &= \text{IO} \mid \text{C1} \setminus \{\text{put}, \text{get}\} & c_{11} &= \text{I1} \mid \text{C1} \setminus \{\text{put}, \text{get}\} \\
c_{10} &= \text{IO} \mid \text{C0} \setminus \{\text{put}, \text{get}\} & c_{9} &= \text{I1} \mid \text{C0} \setminus \{\text{put}, \text{get}\} \\
c_{8} &= \text{C3} \mid \overline{\text{out}}.\text{IO} \setminus \{\text{put}, \text{get}\} & c_{7} &= \text{C2} \mid \overline{\text{out}}.\text{IO} \setminus \{\text{put}, \text{get}\} \\
c_{6} &= \text{C1} \mid \overline{\text{out}}.\text{IO} \setminus \{\text{put}, \text{get}\} & c_{5} &= \text{C0} \mid \overline{\text{out}}.\text{IO} \setminus \{\text{put}, \text{get}\} \\
c_{4} &= \text{C3} \mid \overline{\text{out}}.\text{I1} \setminus \{\text{put}, \text{get}\} & c_{3} &= \text{C2} \mid \overline{\text{out}}.\text{I1} \setminus \{\text{put}, \text{get}\} \\
c_{2} &= \text{C1} \mid \overline{\text{out}}.\text{I1} \setminus \{\text{put}, \text{get}\} & c_{1} &= \text{C0} \mid \overline{\text{out}}.\text{I1} \setminus \{\text{put}, \text{get}\}
\end{aligned}$$

and its transition relation is expressed by the graph of Figure 3.2

Figure 3.2 The Compressing Server's transition relation



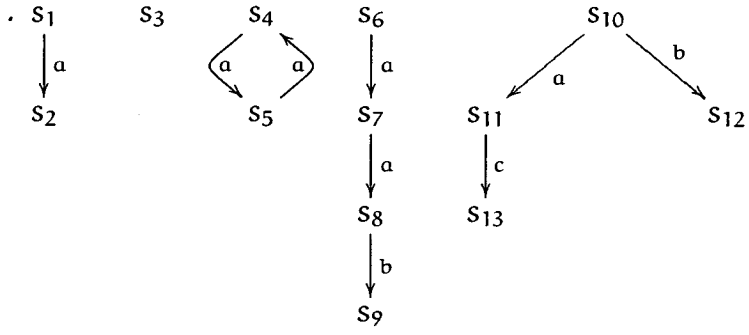
3.3 Logic

The task of a logic is to allow for precise expression of properties. Modal and temporal logics are the logics particular to transition systems. Formulae refer to states after transitions or series of transitions from a state (this is the modal part) as well as properties of the state itself (this is the *propositional* part). The logic I use is the modal-mu calculus [Koz83] with the slight extension to action-sets described in [Bra91].

$$\phi = \mathrm{tt} \mid \mathrm{ff} \mid X \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \mid \langle\mathcal{K}\rangle\phi \mid [\mathcal{K}]\phi \mid \nu X.\phi \mid \mu X.\phi$$

We refer to modal-mu calculus in future as μM. μM is interpreted on labelled transition systems, such as those CCS processes describe. $\mathcal{K}$ refers to sets of transition names in the LTS. We write $s \models \phi$ when the formula ϕ is true for the state s. The following examples are intended to give a rough, intuitive understanding of the interpretation of formulae. Consider the collection of 5 transition systems starting at s_1, s_3, s_4, s_6 and s_{10} of Figure 3.3 as examples:

Figure 3.3 Example transition systems



The following properties (amongst others, of course) hold:

$s_1 \models \langle a \rangle tt$ because an a-action can occur in that state, and the subsequent state $s_2 \models tt$. (All states $s \models tt$, trivially, just as none $s \models ff$).

$s_3 \models [a]\phi$ for any formula ϕ, including ff. s_3 has no successors in the set of actions $\{a\}$, so any statement about *all* s_3*'s* $\{a\}$-*successors* must hold.

$s_4 \models \nu X.[a]X$. We can do a-actions forever, on any path from s_4. Interpreting the formula more directly, s_4 satisfies the fixpoint at X because all a-actions lead to states (s_5) which satisfy the fixpoint at X because all a-actions from s_5 lead to states (s_4) which satisfy the fixpoint at X because...

$s_6 \models \mu X.[a]X \vee \langle b \rangle tt$. It is eventually possible, from s_6 for a b-action to occur. This is true because s_6 satisfies the *least*-fixpoint at X; it goes under all a-actions to s_7 which in turn goes to s_8 which directly satisfies $X \vee \langle b \rangle tt$ because a b-action is possible. The μ-fixpoint captures the notion of a property being approachable finitely (though not within a specific finite number of transitions, in the general case).

$s_{10} \models \langle a, b \rangle \langle c \rangle tt$ as $s_{11} \models \langle c \rangle tt$. This is the case because the a-action takes us to s_{11}, But it is not true that $s_{10} \models [a, b]\langle c \rangle tt$ because $[A]$ requires the successor formula hold of all A-successors, and we don't have $s_{12} \models \langle c \rangle tt$

3.4 Proof

Because we use the proofs of properties such as those just described as formal objects from which to generate oracles, we need a formal notation for the proofs. We use the tableau method of [Bra91, BS92]. This provides for writing proofs that sets of LTS-states satisfy formulae, through the recursive decomposition of the formula. We provide the full set of rules for these tableaux in Figure 4.4. We can outline the tableau for some of the properties in Figure 3.3.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\{s_4\} \vdash \nu X.[a]X}{\{s_4\} \vdash U_0}\,(\text{Def})}{\{s_4, s_5\}^1 \vdash U_0}\,(\text{Thin})}{\{s_4, s_5\} \vdash [a]U_0}\,(\text{Un})}{\{s_5, s_4\}^2 \vdash U_0}\,([\dashv])$$

The root node (the top of the tableau tree) presents a state set and a goal formula. The fixpoint rule for ν demands that the set at the U_0-leaf, $\{s_5, s_4\}^2$, is contained in a U_0-set above it in the tree, to wit $\{s_4, s_5\}^1$. The *Thin* rule allows this containment rule to be applied to a bigger set which contains the set of interest.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\{s_6\} \vdash \mu X.[a]X \vee \langle b\rangle \mathrm{tt}}{\{s_6\} \vdash U_0}\,(\text{Def})}{\{s_6, s_7, s_8\} \vdash U_0(s_8 < s_7 < s_6)}\,(\text{Thin})}{\{s_6, s_7, s_8\} \vdash [a]U_0 \vee \langle b\rangle \mathrm{tt}}\,(\text{Un})}{\dfrac{\{s_6, s_7\} \vdash [a]U_0}{\{s_7, s_8\} \vdash U_0}\,([\dashv]) \quad \dfrac{\{s_8\} \vdash \langle b\rangle \mathrm{tt}}{\{s_9\} \vdash \mathrm{tt}}\,(\langle\dashv\rangle)}\,(\vee)$$

The principle of tableau generation is to break down the formula by construction and to associate a solution set with each subformula. Propositional constants (U_i) are introduced at fixpoints, and special conditions (of which more anon.) are imposed at the fixpoints.

Although the rules are complicated we can determine whether a tableau is correctly instantiated, and the subsequent manipulation of the representation into one more suited to use as an automaton is complex, but is just a matter of coding. For now let us assume we have these proofs in whatever form is most convenient to us. The information a proof contains can be interpreted as defining the set of correct behaviours of the specification with respect to the property. The basic principle is that where $S \vdash \langle[\mathcal{K}]\rangle\psi$ ($[\mathcal{K}]$ or $\langle\mathcal{K}\rangle$) is decomposed to $S' \vdash \psi$ then $s \in S$ under $s \xrightarrow{a \in \mathcal{K}}$ should evolve to $s' \in S'$. In particular this principle sometimes decrees that a specific action must not occur in some set S_i, in effect defining certain runs of actions which are not admissible according to the proof. For instance the proof of $S \models \mu X.[a](X \wedge [b])\,\mathrm{ff}$ prohibits all behaviours of the form

$$ab, aab, aaab, aaaab, aaaaab \ldots$$

For the compressing server, we want to verify that full buffers don't make the system deadlock. We must express this in μM and write a proof for it. The formula we choose is

$$\mu Y.\langle in\rangle \mathrm{tt} \vee ([\dashv] Y \wedge \langle\dashv\rangle \mathrm{tt})$$

which expresses our wish by saying that eventually it becomes possible to input something to the system. We're interested particularly in the states c_{17} and c_{18}

since these are the only states in which the buffers are full. We might better have chosen a slightly more involved property,

$$\nu X.[\vdash]X \wedge (\mu Y.\langle in\rangle tt \vee ([\vdash]Y \wedge \langle -\rangle tt))$$

and examined it for the initial state (c_{10}), to express that the system should never reach a state in which an input can't eventually happen. But the property we use is slightly simpler, to the benefit of the exposition. In either case, a proof is small enough to be tractable and readily understood. Indeed, the proof of any comprehensible property should not be monstrous, and we take this as empirical evidence that our claims of ease of proof-writing are justified.

The proof of $\mu Y.\langle in\rangle tt \vee [\vdash]Y \wedge \langle -\rangle tt$ is (writing $\{c_{i,j,k}\}$ to stand for the set of states $\{c_i, c_j, c_k\}$):

$$\dfrac{\dfrac{\dfrac{\dfrac{\{c_{17,18}\} \vdash \mu Y.\langle in\rangle tt \vee [\vdash]Y \wedge \langle -\rangle tt}{\{c_{17,18}\} \vdash U_0}\,(\text{Def})}{\{c_{1\ldots4,9,11\ldots18}\} \vdash U_0}\,(\text{Thin})}{\{c_{1\ldots4,9,11\ldots18}\} \vdash \langle in\rangle tt \vee [\vdash]U_0 \wedge \langle -\rangle tt}\,(\text{Un})}{\dfrac{\dfrac{\{c_{12,14,16,18}\} \vdash \langle in\rangle tt}{S \vdash tt}\,(\langle -\rangle)}{\dfrac{\dfrac{\{c_{1\ldots4,9,11,13,15,17}\}(S_1) \vdash [\vdash]U_0 \wedge \langle -\rangle tt}{S_1 \vdash [\vdash]U_0}\,(\wedge)}{\{c_{1\ldots4,9,11,16,18}\}(S_2) \vdash U_0}\,([\vdash])\quad \dfrac{S_1 \vdash \langle -\rangle tt}{S_2 \vdash tt}\,(\langle -\rangle)}\,(\vee)}$$

We interpret each node of the tableau as indicating that its subformula must be true for the set of states indicated at the node. The collection of (state, subformula)-pairs is the extra information beyond the bare truth of the root formula which allows us to generate intensional oracles.

3.5 Oracles

Now we are equipped to explain how our oracles are generated, and what they consist of.

If we choose not to treat least fixpoints, the oracle we generate for a tableau is simply a non-deterministic finite-state-machine, where states with ff on the right hand side are labelled *reject*. The oracle has a single state for each node of the tableau. If any rejecting state is reached, the oracle has discovered an error.

From this basis, we can make several observations and improvements.

1. We generate the oracle from a particular *canonical* version of the tableau. It is easy to translate any tableau to canonical form [Bra91], and in our formal treatment we take a further step in notation to help us describe the manipulations of these *proofs* more elegantly in the oracle-generation process.

2. In the case of tableaux which involve least fixpoints we must replace each node by a set of nodes reflecting the closeness to termination of the least fixpoint. In the worst case this set of nodes can be large or even infinite, though in practical examples this is unlikely.

   To be positive about it, an oracle for the pure logic could never check a least fixpoint assertion for termination. The well-foundedness (repeats until termination) information is intensional, i.e. it is dependent on the structure of the specification, and as such it would not be available at all to an extensional oracle. Even a very conservative approximation for the set of nodes, drawn from the specification, would be of benefit. There is no escaping this complexity in general (it comes from the external well-foundedness conditions which are necessary to prove least-fixpoints).

3. Clearly we need to make a running oracle deterministic. If it were not for the $\langle\!\langle K \rangle\!\rangle$ modality and the $\vee$-disjunction, this would be a simple matter of accepting only if all final states accept. We can remove non-determinism by the standard mechanism of increasing the state set to be the powerset of the original state set.

3.5.1 Oracle States

We can interpret the proof of $\mu Y.\langle\!\langle n \rangle\!\rangle tt \vee [\!] Y \wedge \langle - \rangle tt$ as defining a number of (S, ψ)-pairs ((stateset,subformula)-pairs) where the states in the sets should satisfy the subformulae. These satisfaction relations are what the oracle must check.

We can optimise things slightly by not checking every (S, ψ)-pair. Where the tableau nodes are propositional breakdowns of formulae, checking nodes where the ψ are what we will call the *propositional-atomic* formulae are sufficient. Those are $[\![K]\!]\phi$, $\langle\!\langle K \rangle\!\rangle \phi$ as well as the trivial tt and ff. The truths of the other nodes can be inferred from the tableau rules and propositional reasoning, without reference to the transition relation. For the current formula, the subformulae of interest can be reduced to

$$1.\ \langle\!\langle n \rangle\!\rangle tt,\quad 2.\ [\!] U_0,\quad 3.\ \langle - \rangle tt$$

The complete proof for a μ-formula involves defining a *well-order* (partial order with no infinite chains) external to the tableau, on the set of states which are intended to satisfy the least-fixpoint formula. Thus, for a $\{s_i, s_j, s_k\} \vdash U_0$ node, a well-order must be discovered on the set $\{s_i, s_j, s_k\}$. We use the well-order to divide up the states satisfying U_0 into subsets, according to their maximum distance from the top of any chain in the order. Thus, the well-order imposes a maximum repetition count on a μ-formula. In the case here where μY is used to say that a property $P(\langle\!\langle n \rangle\!\rangle tt)$ eventually holds, the repetition count limits the

number of times that the system state can go through a loop

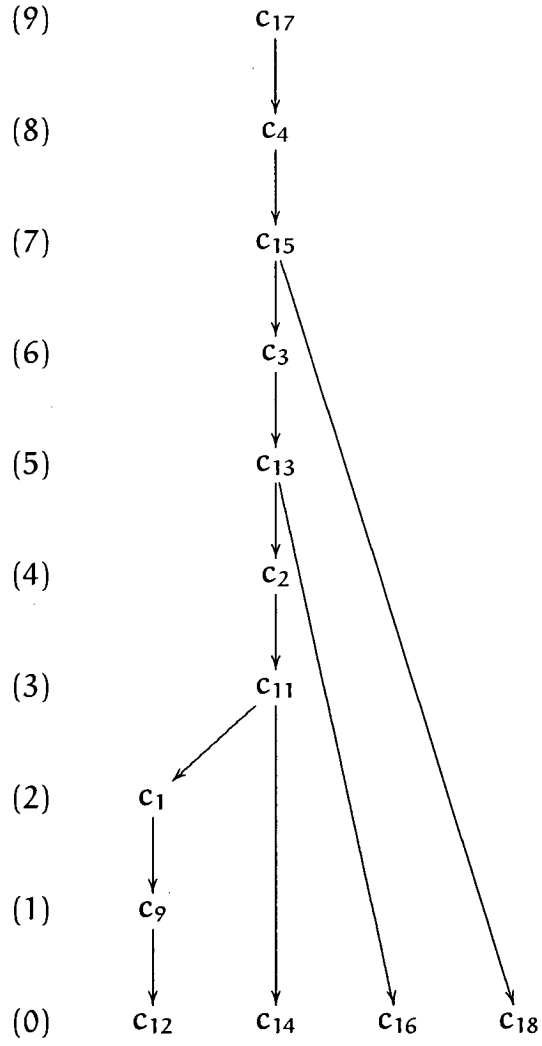
$$s \vdash [-]U_0, s \xrightarrow{a} s', s' \vdash [-]U_0, s' \xrightarrow{a} s'', s'' \vdash [-]U_0$$

before it reaches a terminating state s^α

$$s^{\alpha-1} \xrightarrow{a} S^\alpha, s^\alpha \vdash \langle in \rangle tt$$

In general more than one state will have the same number of repetitions allowed, because each is the same distance from the bottom of the most distant chain. These states can be dealt with as a set; the oracle simply has to allow them the defined number of repetitions to reach a state satisfying the terminal property P. For our example, the well-order (with maximum repetitions on the left) is shown in figure 3.4.

Figure 3.4 A well-order for $\mu Y.\langle in \rangle tt \vee ([-]Y \wedge \langle - \rangle tt)$



Discovering a well-order allows us to define the oracle states for the example formula and proof. Each state is a pair of formula, plus a count-tuple. The count

is empty if the formula does not contain least-fixpoint formulae as subformulae. Our example has a single μ-fixpoint(s) involved in some states; these states have a count for that fixpoint. The states are:

$$\begin{aligned}\{\ & (\langle \mathfrak{in} \rangle \mathrm{tt}, ()), (\langle - \rangle \mathrm{tt}, ()), \\ & (\vdash\!\!\!\dashv U_0, (1)), (\vdash\!\!\!\dashv U_0, (2)), (\vdash\!\!\!\dashv U_0, (3)), (\vdash\!\!\!\dashv U_0, (4)), \\ & (\vdash\!\!\!\dashv U_0, (5)), (\vdash\!\!\!\dashv U_0, (6)), (\vdash\!\!\!\dashv U_0, (7)), (\vdash\!\!\!\dashv U_0, (8)), \\ & (\vdash\!\!\!\dashv U_0, (9)) \\ \}\end{aligned}$$

We shall write $|s, \phi| = n$ when the repetition count for the state s, for the formula ϕ, is n; reading from the well-order of Figure 3.4, we have $|c_{17}, \vdash\!\!\!\dashv U_0| = 1$.

3.5.2 Oracle Transitions

Now we have the states of the oracle LTS, we obviously need to define a transition relation. We will develop this in stages. The first step is to ask what the relation stands for. Since we have decided that states stand for properties of LTS states it is natural that the transition relation stands for some kind of necessitation of a second property by a first, under the appropriate transition of an LTS state. So we have e.g.

$$(\phi, n) \xrightarrow{a} (\phi', n') \Leftrightarrow \exists s \in \{|\phi, n|\}, s' \in \{|\phi', n'|\} : s \xrightarrow{a} s'$$

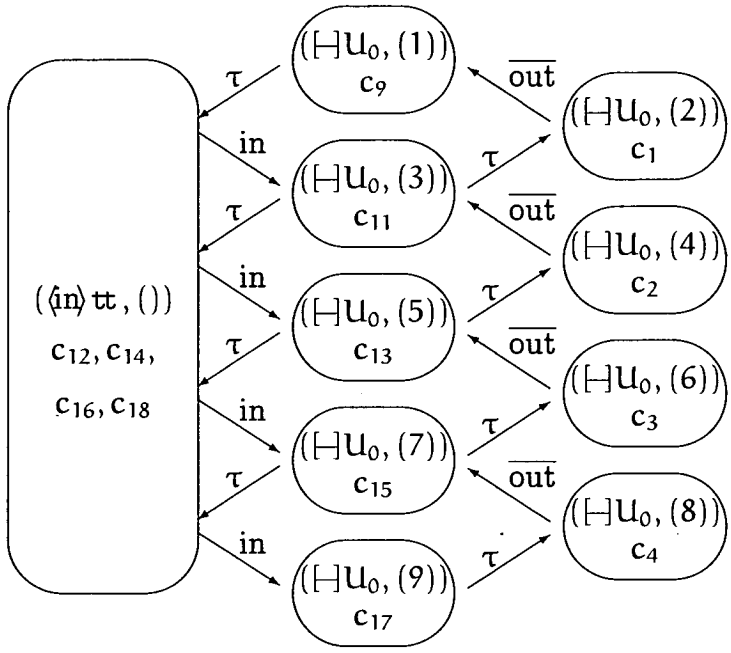
with

$$\{|\phi, n|\} \triangleq \{s : S \vdash \phi \text{ a tableau node, and } |s, \phi| = n\}$$

For now we are dealing only with formulae including at most one μ-fixpoint. Where there are no fixpoints, n will not be relevant. The case where there are multiple fixpoints is more complicated, and we omit it for now because a single fixpoint gives the flavour of the approach.

Figure 3.5 shows the oracle for the compressing server for the example formula. The arrows represent the transition relation as we have just explained it.

Figure 3.5 Oracle for compressing server for $\mu Y.\langle in\rangle tt \vee [-]Y \wedge \langle -\rangle tt$



The transition-system we have generated can very easily be used as the data for a checking program. Starting at a known initial state, and observing actions of the annotated implementation, we can at each stage infer that the system must be in one of a limited number of the oracular states, and consequently should satisfy particular formulae. Since some formulae turn out to be trivial to annotate, we can go one better and use the annotation to *declare* the formulae. The oracle can then check that all declared formulae are implied by the states of the oracle LTS. For example $\langle in\rangle tt$ is simple to confirm or refute, because it almost certainly corresponds to an input-processing module in the implementation.

Ensuring consistency of the oracle LTS and declarations from the annotation forms the basis of the verification mechanism.

3.5.3 Ambiguity

Figure 3.5 shows that the same transition can lead to different states. For example

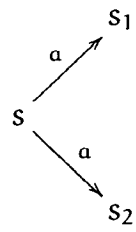
$$[-]U_0, (5) \xrightarrow{\tau} [-]U_0, (4)$$
$$[-]U_0, (5) \xrightarrow{\tau} \langle in\rangle tt, ()$$

This poses problems for the oracle. It must be a non-deterministic one, or practically we must define the oracle configuration as a more complex structure.

The problem combines choice, and the paucity of information conveyed by the transition name alone. Imagine the state s, satisfying

$$s \models [a](\psi_1 \vee \psi_2)$$

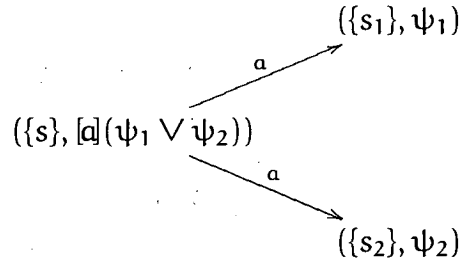
and having as its derivatives



Imagine that part of the tableau we use (and in general it may be the only correct tableau, so the problem cannot be finessed)

$$\frac{\dfrac{\{s\} \vdash [a]\psi_1 \vee \psi_2}{\{s_1, s_2\} \vdash \psi_1 \vee \psi_2}\,(\vdash)}{\{s_1\} \vdash \psi_1 \quad \{s_2\} \vdash \psi_2}\,(\vee)$$

Then the oracle has the pair of a-transitions



and of course it is impossible to determine which. Control must somehow keep track of the options; if the LTS state is s_1 then the oracle state is $(\{s_1\}, \psi_1)$, but if the LTS state is s_2 then the oracle state is $(\{s_2\}, \psi_2)$. The problem arises because to an observer, the transition system is non-deterministic. We resolve the problem in Chapter 6 by having annotation reveal the state to us.

3.5.4 Conjunction

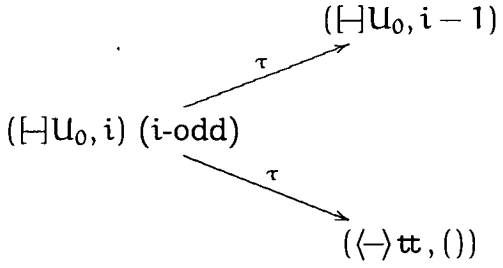
We have cheated a little in the presentation of the oracle transition system, by omitting the state $\langle - \rangle \mathrm{tt}$ and transitions into it. The full OTS has the state

$$(\langle - \rangle \mathrm{tt}, ())$$

and for every $([-]U_0, i)$

$$([-]U_0, i) \xrightarrow{\tau} (\langle - \rangle \mathrm{tt}, ())$$

The multiple τ-transitions



are parts of the same combined property. In the abstract

$$\dfrac{\dfrac{\{s\} \vdash [a]\psi_1 \wedge \psi_2}{\{s'\} \vdash \psi_1 \wedge \psi_2}\,([\vdash])}{\{s'\} \vdash \psi_1 \quad \{s'\} \vdash \psi_2}\,(\vee)$$

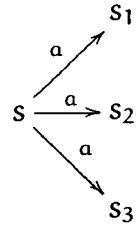
Clearly, in order to check the property, the oracle must make the transition to both states simultaneously, and verify both ψ_1 and ψ_2. This can result in a large blow up of the number of properties simultaneously to be checked, if the $\wedge$ is regenerated by a ν or μ formula and neither branch dies out quickly (as for instance $\langle\dashv\rangle tt$ dies out).

3.5.5 Resolving Things

The combination of ambiguities and conjunctions means that the oracle state can be very complicated. For example, the proposition that

$$s \vdash [a](\psi_1 \vee (\psi_2 \wedge (\psi_3 \vee \psi_4)))$$

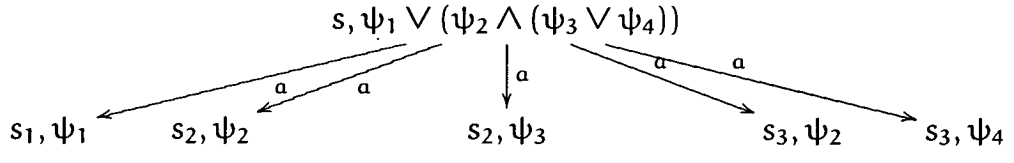
the LTS



and the tableau

$$\dfrac{\dfrac{\dfrac{\{s\} \vdash [a]\psi_1 \vee (\psi_2 \wedge (\psi_3 \vee \psi_4))}{\{s_1, s_2, s_3\} \vdash \psi_1 \vee (\psi_2 \wedge (\psi_3 \vee \psi_4))}\,([\vdash])}{\{s_1\} \vdash \psi_1 \qquad \{s_2, s_3\} \vdash \psi_2 \wedge (\psi_3 \vee \psi_4)}\,(\vee)}{\dfrac{\{s_2, s_3\} \vdash \psi_2 \quad \{s_2, s_3\} \vdash \psi_3 \vee \psi_4}{\{s_2\} \vdash \psi_3 \quad \{s_3\} \vdash \psi_4}\,(\wedge)}\,(\wedge)$$

means that subsequent to an a-transition on the oracle



the subformulae of interest are

- in state s_1, ψ_1
- in state s_2, ψ_2 and ψ_3
- in state s_3, ψ_2 and ψ_4

and the oracle mechanisms must reflect this. The best way to do this seems to be to represent the oracle state formally as a set of conditional (s, ψ)-pairs, in this case

$$\{(s_1, \psi_1), (s_2, \psi_2), (s_2, \psi_3), (s_3, \psi_2), (s_3, \psi_4)\}$$

The general principle must be to maintain the set of conditional (s, ψ) (read as if the LTS is in state s, then the formula ψ should hold), irrespective of what the true LTS state is. The set of formulae necessarily true of a particular LTS state is not in doubt, only which state the LTS modelling the implementation is really in. The price we pay is the possible large size of the conditional configuration, and we must then address how to keep the size manageable.

3.5.6 The Role of Declarations

We have mentioned in passing that in checking the compressing server we use the fact that the prop-atomic formula $\langle in \rangle tt$ can be declared (or equally, refuted), by the annotated system. In view of the problems of imperfectly determined states from actions, I find it it necessary to promote such proposition declaration to an equal prominence with transitions. The way to do it is to take the calculated set of configurations inferred from the OTS

$$\{(s_1, \psi_1), (s_2, \psi_2), (s_2, \psi_3), (s_3, \psi_2), (s_3, \psi_4)\}$$

and to prune it by any states concerned with refuted formulae. If we know $\neg\psi_2$, then s_2 and s_3 are denied, giving

$$\{(s_1, \psi_1)\}$$

For example in the compressing server, declaring $\neg\langle in \rangle tt$ in

$$\{(c_2, \boxdash U_0), (c_{12}, \langle in \rangle tt)\}$$

denies c_{12}, and we must conclude that the LTS is in state c_2. The denial of all possible states constitutes the discovery of an error, so that

$$\{(c_{12}, \langle in \rangle tt)\}, \neg\langle in \rangle tt \text{ declared}$$

represents an implementation of the compressing server which does not conform to the proof; declaring $\neg\langle in \rangle tt$ means that we have not yet reached the bottom

of the fixpoint descent for U_0, but we expect to have done so because the oracle believes we must be in c_{12}. The conclusion must be that the bounds on fixpoint descent have been exceeded.

A proof can be viewed as providing a set of conditional statements, for example:

- if the state is c_1, then ψ_1 should hold
- if the state is c_2, then ψ_2 and ψ_3 should hold
- if the state is c_3, then ψ_2 and ψ_4 should hold

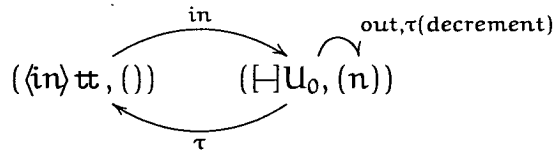
The effect of regular declarations in the behaviour of an implementation is to prune the set of conditional statements. This view copes with transitional ambiguities and conjunctions uniformly. Then declarations reduce the complexity so produced, also uniformly. The fundamental need for information to produce a small and useful oracle is attacked along a different information axis.

3.5.7 Collapsing Oracle States

In deriving an oracle from an LTS and a proof we move from a state-based system to a more abstract one, with oracle states as formulae satisfied by the original LTS states. This is reasonable because we are not throwing away the information that allows us to decide that a system is behaving incorrectly, as we would if we generated oracles purely from logical formulae. But it may be possible to go further towards making oracles smaller, while still retaining more information than in purely formula-derived oracles.

Consider what further simplifications might consist of; when we count down a fixpoint (i.e. n is not a trivial $()$ in a state) our only concern is really that we have some checkable upper bound by which number of loops any member state of the set satisfying the fixpoint formula will have exited the fixpoint loop. In the compressing server, this can be expressed by after any $\xrightarrow{\text{in}}$ action, resetting a counter of admissible transitions before another $\xrightarrow{\text{in}}$ occurs. The counter value will be reset to 9, as this is the length of the longest chain in the well-order discovered for the proof of $\mu Y.\langle \text{in} \rangle \text{tt} \vee [\text{-}]Y \wedge \langle \text{-} \rangle \text{tt}$, in the compressing server. Figure 3.6 is a sketch of how this version of the oracle looks.

Figure 3.6 Collapsed Compressing Server Oracle



Although the system of figure 3.6 is intuitively simpler than that of Figure 3.5, a version expressed just as a transition system would have just as many states. In any case, we have not shown a way of automatically deriving such an oracle. Methods for automatically collapsing oracles, possibly through translations of well-orders into ones which are as pessimistic but with less chains, should be investigated. Work in Chapter 8 looks at how counting-based shorthands can be applied to efficient representation of transition systems, and is applicable for representing oracles.

3.6 Conclusion

I have described how to take a system to be audited and have

1. Introduced CCS, and specified the example system.
2. Introduced μM, and presented a fundamental property of the example to prove.
3. Proved the property using a tableau.
4. Described the process of deriving an oracle from a tableau, discussed the structure of the resulting oracle, and raised some of the detailed questions of oracle implementation.

This chapter should convince the reader that the approach is practical for the small example considered. The subsequent Chapters 4, 5 and 6 reconstruct the argument of this chapter on a much deeper and more formal level; they are necessary reading both as theoretical background and for the detailed mechanics of how to implement automatic oracle generation. The reader more interested in application of the methodology can jump on to Chapter 7.

Chapter 4

Transition Systems and Logic

Summary

In this chapter I examine in detail the network of formalisms which I use as the framework for proof-based oracles. This helps to fill in details which have been elided in Chapter 3 for want of formal support. The chapter can be read as a review of a particular framework for doing useful proof work in concurrent systems; this corresponds to the first stage in the process of staged introduction of formal methods into a software development process, with auditioning as the second stage. Just as introducing auditioning in development builds on initial modelling and proving work, so auditioning itself requires the formal foundations of a modelling and proving methodology.

The components of a formal system for modelling and proof are

- A model. I recall the standard model for concurrent systems, the LTS.
- A specification language, which defines models.
- A logic, which expresses properties of systems, and defines their truth relative to models.
- A proof system, which connects logic and model and serves as a recipe for deriving the truth or falsehood of properties in the logic, on the model.

I devote a section to the discussion of each component, covering the formal definition of the CCS specification language, the modal-mu calculus (logic) and the tableau proof system.

4.1 Labelled Transition Systems

Definition 4.1 A labelled transition system (LTS) is a triple $\langle S, A, T \subseteq S \times A \times S \rangle$

S is a set of states

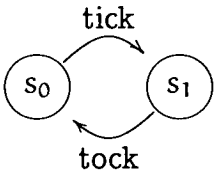
A is a set of action names

T is the transition-relation

and we write $s_1 \xrightarrow{a} s_2$ when $\langle s_1, a, s_2 \rangle \in T$

LTSs are often represented graphically as nodes S, with arcs A connecting those pairs of nodes related by T. For example, a grandfather clock looks like Figure 4.1.

Figure 4.1 A sonorous clock



Explicitly, the clock's transition system is

$$\langle \{s_0, s_1\}, \{\text{tick}, \text{tock}\}, \{\langle s_0, \text{tick}, s_1 \rangle, \langle s_1, \text{tock}, s_0 \rangle\} \rangle$$

System descriptions written in CCS[Mil89], TLA[Lam91] and Petri Nets[Pet62] all induce LTSs.

4.2 CCS

CCS [Mil89] is a particularly appealing language for the expression of concurrent behaviour. Milner uses the term *agents* to avoid commitment to how CCS entities are viewed. They can be either descriptive or prescriptive. In his model for specification a small CCS process prescribes the correct behaviour of a system, a complex CCS process describes the actual behaviour and an equivalence relation between the two captures conformance of implementation (actual behaviour) with specification (required behaviour).

A CCS system is defined from a set of actions A. The silent action τ plays a special role, and is distinct from any $a \in A$. The agent 0 (or null) is inert ($\neg\exists s \in S, a \in A : 0 \xrightarrow{a} s$). Agents have a simple syntax and their behaviour is based on their syntactic form. The most fundamental such form is the action-prefix. The agent $a.E$ prefixes a to E and behaves as the agent that can perform an a-action to become the agent E.

Definition 4.2 (Agents and Processes) Using $\alpha \in A$ to range over non-τ actions, and assuming all agent sets contain all inverses ($\alpha \in A$ iff $\overline{\alpha} \in A$) and $\overline{\overline{\alpha}} = \alpha$ we let $E, F, \ldots$ range over agents and $P, Q, \ldots$ over named processes

$$E \triangleq 0 \mid \alpha.0 \mid E + F \mid E \mid F \mid \Sigma_{i \in I} E_i \mid \Pi_{i \in I} E_i \mid \textbf{rec}\, \tilde{X} = \tilde{E}$$

$$P \triangleq N = E \text{ (where N stands for process names)}$$

Definition 4.3 CCS processes behave according to the set of SOS (structured operational semantics) rules[Plo81, BIM95] defined in Figure 4.2.

Figure 4.2 CCS SOS-rules

$\alpha.E$	prefix	$\alpha.E \xrightarrow{\alpha} E$
$E + F$	choice	$\frac{E \xrightarrow{\alpha} E'}{E + F \xrightarrow{\alpha} E'} \qquad \frac{F \xrightarrow{\alpha} F'}{E + F \xrightarrow{\alpha} F'}$
$\Sigma_{i\in I} E_i$	(generalised)	$\frac{E_j \xrightarrow{\alpha} E_j'}{\Sigma_{i\in I} E_i \xrightarrow{\alpha} E_j'}\, j \in I$
$E \mid F$	parallel	$\frac{E \xrightarrow{\alpha} E'}{E \mid F \xrightarrow{\alpha} E' \mid F} \qquad \frac{F \xrightarrow{\alpha} F'}{E \mid F \xrightarrow{\alpha} E \mid F'}$
$\Pi_{i\in I} E_i$	(generalised)	$\frac{E_j \xrightarrow{\alpha} E_j'}{\Pi_{i\in I} E_i \xrightarrow{\alpha} \Pi_{i\in I-\{j\}} E_i, E_j'}\, j \in I$
	interaction	$\frac{E \xrightarrow{\alpha} E' \quad F \xrightarrow{\overline{\alpha}} F'}{E \mid F \xrightarrow{\tau} E' \mid F'}$
	(generalised)	$\frac{E_j \xrightarrow{\alpha} E_j' \quad E_k \xrightarrow{\overline{\alpha}} E_k'}{\Pi_{i\in I-\{j,k\}} E_i, E_j, E_k \xrightarrow{\tau} \Pi_{i\in I-\{j,k\}} E_i, E_j', E_k'}$
$\mathbf{rec}_i\, \tilde{X}$	recursion*	$\frac{E_i[\mathbf{rec}\, \tilde{X} = \tilde{E}/X] \xrightarrow{\alpha} E'}{\mathbf{rec}_i\, \tilde{X} = \tilde{E} \xrightarrow{\alpha} E'}$
$E\backslash L$	restriction	$\frac{E \xrightarrow{\alpha} E'}{E\backslash L \xrightarrow{\alpha} E'\backslash L}\, \alpha \notin L$
$E[f]$	relabelling	$\frac{E \xrightarrow{\alpha} E'}{E[f] \xrightarrow{f(\alpha)} E'[f]}$
$P \triangleq E$	definition	$\frac{E \xrightarrow{\alpha} E'}{P \xrightarrow{\alpha} E'}\, P \triangleq E$

(*)The recursion rule says that the behaviour of any member of a family of simultaneous recursion equations is just the behaviour of the member's recursion expression substituted by all the definitions in the family, i.e. recursion works by unwinding

These are close to Milner's standard definitions, where the rec expressions serve to separate recursion and definition. Practically, letting all foregoing rules range over $P, Q, \ldots$ the single rule

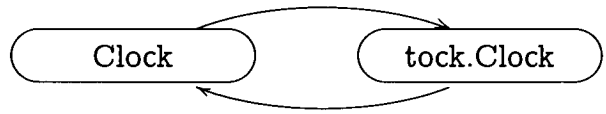
$$\text{definition and recursion} \qquad \frac{Q \xrightarrow{\alpha} Q'}{P \xrightarrow{\alpha} Q'} \; P \triangleq Q$$

has the same effect as definition plus recursion. We can write the clock example in CCS as

$$\text{Clock} \triangleq \text{tick.tock.Clock}$$

The definition and prefix rules are enough to infer the transition system of Figure 4.3.

Figure 4.3 Grandfather clock induced by CCS



The CCS transition rules of Definition 4.3 allow us to infer the presence of transitions. Given an agent we can sometimes close the set of reachable states and thus explicitly express an LTS. This is not always the case because we can express the halting problem in CCS. In general we therefore only have local knowledge of the system; luckily the proof system we use is inherently local (see Section 4.4.1).

Milner equips CCS with two process congruences, $\sim$ and $=$. The former (bisimilarity) equates processes when each can simulate the behaviour of the other, treating all actions the same.

Definition 4.4 (Strong bisimulation) is the largest relation $\sim$, such that $P \sim Q$. $P \sim Q$ if and only if $\forall a \in A$

$$P \xrightarrow{a} P' \Rightarrow \exists Q' : Q \xrightarrow{a} Q' \text{ and } P' \sim Q'$$
$$Q \xrightarrow{a} Q' \Rightarrow \exists P' : P \xrightarrow{a} P' \text{ and } P' \sim Q'$$

And there is a largest such relation [Mil89].

There are many other notions of process equality, but $\sim$ is sufficiently strong that there is little reason to distinguish processes equated by it. It is therefore common practice implicitly to quotient transition systems induced by CCS by the $\sim$ relation.

4.3 Specification in CCS

When CCS is used both as specification and as implementation, strong bisimulation distinguishes too many processes, and is thus unsuitable for relating implementation to specification. The $=$-relation relaxes the criteria for equality sufficiently to make it usable. The definition uses a subsidiary equivalence (which is not a congruence).

Definition 4.5 (Weak bisimulation) is the largest relation $\approx$, such that $P \approx Q$. $P \approx Q$ if and only if $\forall a \in A$

$$P \stackrel{s}{\Longrightarrow} P' \Rightarrow \exists Q' : Q \stackrel{s}{\Longrightarrow} Q' \text{ and } P' \approx Q'$$
$$Q \stackrel{s}{\Longrightarrow} Q' \Rightarrow \exists P' : P \stackrel{s}{\Longrightarrow} P' \text{ and } P' \approx Q'$$

where

$$P \stackrel{a_1, a_2, \cdots}{\Longrightarrow} P' \Leftrightarrow \exists i, P_1, P_2, \ldots, P_i : P \stackrel{\tau}{\longrightarrow}^* \stackrel{a_1}{\longrightarrow} \stackrel{\tau}{\longrightarrow}^* P_1 \stackrel{\tau}{\longrightarrow}^* \stackrel{a_2}{\longrightarrow} \stackrel{\tau}{\longrightarrow}^* P_2 \stackrel{\cdots}{\longrightarrow} P'$$

Definition 4.6 (Weak congruence (equality)) $P = Q$ if, for all a,

$$P \stackrel{a}{\longrightarrow} P' \Rightarrow \exists Q' : Q \stackrel{a}{\Longrightarrow} Q' \text{ and } P' \approx Q'$$
$$Q \stackrel{a}{\longrightarrow} Q' \Rightarrow \exists P' : P \stackrel{a}{\Longrightarrow} P' \text{ and } P' \approx Q'$$

4.3.1 Refinement within CCS

Where CCS is used to analyse both implementations and specifications, it is usual to relate them using the $=$-equivalence. The degree to which $=$ is insensitive to τ-actions seems to be enough to make this approach practical for some useful examples [Mil89]. But it is not at all clear that an equivalence is the right notion for specification. For instance,

$$I_1 = S, I_2 = S \Rightarrow I_1 = I_2$$

suggesting that where any two implementations satisfy the same specification, then they are in some sense the same as each other. The standard response is to define a refinement relationship which is a pre-order. We write $I \succ S$ when I is a refinement/ implementation of S. Formal development will produce a series of refinements

$$S \prec R_1 \prec R_2 \prec R_3 \prec I$$

but where $S \prec I_1$ and $S \prec I_2$, the alternative implementations have no necessary relationship to each other.

Attempts have been made to provide asymmetric process relationships within CCS [Bru94], but it seems that a canonical one may not exist. Perhaps a workable refinement relationship needs a more general way of relating actions or sequences of actions; or perhaps refinement is simply not universal, but

relative to the initial specification. If we take the view that a program implements a specification whenever it satisfies an appropriate set of properties, then the implements relationship changes as the set of properties changes and it is unsurprising that there is no perfect choice of implementation relationship.

In any case, it seems worthwhile to look at other notions of specification, both for their own sake and in order to see if they reflect light on specification in CCS. In the rest of this thesis I use CCS at one level only, that of the abstract implementation of auditioning. I write specifications in modal logic.

4.4 Logic

The logical approach to specification considers the meets relation in terms of a set of logical properties. Taking $S = \{\phi_1, \phi_2, \ldots, \phi_n\}$ we might consider one system as a refinement of another if it takes more properties from a chosen set, so:

$$R_1 \prec_S R_2 \Leftrightarrow \forall i. R_1 \models \phi_i \Rightarrow R_2 \models \phi_i$$

There are many possible approaches to defining refinement in terms of properties of intermediate systems. Each refinement step will add some implementation choices which extend the class of properties demanded of subsequent refinement steps. The choice of which properties are relevant and which accidental must then be clarified.

We are only concerned with proving properties of specifications, not with designing a refinement calculus. We merely wish to suggest that using separate logic and specification language is a more flexible approach than using specification language at all levels. We begin by recalling a simple logic with a close link to CCS.

Definition 4.7 (Hennessy-Milner Logic [HM85]) is defined inductively in terms of formulae ϕ

$$\phi \quad \triangleq \quad \mathtt{tt} \mid \mathtt{ff} \mid \langle\alpha\rangle\phi \mid [\alpha]\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2$$

This logic allows the specification of behaviours along finite paths.

Definition 4.8 (Satisfaction) of formulae ϕ by processes P

$$\begin{array}{rcl}
P \models \mathtt{tt} & & \text{always} \\
P \models \mathtt{ff} & & \text{never} \\
P \models \langle a\rangle\phi & \Leftrightarrow & \exists P'. P \xrightarrow{a} P' \text{ and } P' \models \phi \\
P \models [a]\phi & \Leftrightarrow & \forall P' : P \xrightarrow{a} P' \,.\, P' \models \phi \\
P \models \phi_1 \wedge \phi_2 & \Leftrightarrow & P \models \phi_1 \text{ and } P \models \phi_2 \\
P \models \phi_1 \vee \phi_2 & \Leftrightarrow & P \models \phi_1 \text{ or } P \models \phi_2
\end{array}$$

A connection between the logical and equational views of agents exists [HM85]

Definition 4.9 An LTS $\langle S, A, T\rangle$ is image-finite if and only if

$$\forall s \in S, \alpha \in A \,.\, \{s' : s \xrightarrow{\alpha} s'\} \text{ is finite.}$$

Fact 4.10 (Hennessy-Milner) Where the induced LTS is image-finite, processes are bisimilar if and only if they satisfy the same formulae

$$\exists \phi : P \models \phi \text{ and } Q \not\models \phi \Leftrightarrow P \not\sim Q$$

It is easy to see that HML cannot express infinite behaviours. Such properties as *always* and *eventually* are necessary for a practical logic.

4.4.1 Modal mu-calculus

Bradfield in his thesis [Bra91] presents a history of logic-based verification, while Stirling [Sti92] makes an exhaustive survey of the theory of modal and temporal logics. Much present day work takes the modal mu-calculus [Koz83] as the canonical logic for investigating model-checking and verification. It is regular and highly expressive [Bra91, Bra96] so that proper subsets are both interesting (in expressibility and complexity terms) and easily distinguished. Other commonly used logics such as CTL and CTL* can be encoded in μM [Dam94]. Perhaps the only drawback is the sense in which *natural* properties are often not expressible particularly succinctly; this can partly be addressed by a series of macros which translate common properties into raw μM.

We follow [Bra91] in the definition of μM. We work with the *positive normal form*(PNF).

Definition 4.11 (Modal mu-calculus, positive normal form) Given:

A model M, which is an LTS $\langle S, A, T\rangle$,

K stands for any subset of A, and $-K$ for $A - K$,

$X, Y, \ldots$ are variable names,

Z is a set of constant names,

$V : Z \longrightarrow 2^S$ is a valuation function taking names to subsets of S

A formula in the modal-mu calculus is of the form:

$$\phi \triangleq X \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid [K]\phi \mid \langle K\rangle\phi \mid \nu X.\phi \mid \mu X.\phi$$

We can introduce $\neg\phi$ as a derived operator. For any formula contaning $\neg$, we transform it to an equivalent PNF formula by applying the pnf function:

$$\begin{gathered}
\mathrm{pnf}(\neg(\phi_1 \wedge \phi_2)) = \mathrm{pnf}(\neg\phi_1) \vee \mathrm{pnf}(\neg\phi_2) \\
\mathrm{pnf}(\neg(\phi_1 \vee \phi_2)) = \mathrm{pnf}(\neg\phi_1) \wedge \mathrm{pnf}(\neg\phi_2) \\
\mathrm{pnf}(\neg[K]\phi) = \langle K\rangle \mathrm{pnf}(\neg\phi) \\
\mathrm{pnf}(\neg\langle K\rangle\phi) = [K]\mathrm{pnf}(\neg\phi) \\
\mathrm{pnf}(\neg\nu X.\phi) = \mu X.\mathrm{pnf}(\neg\phi) \\
\mathrm{pnf}(\neg\mu X.\phi) = \nu X.\mathrm{pnf}(\neg\phi) \\
\mathrm{pnf}(\neg X) = X,\ X \text{ a fixpoint variable} \\
\mathrm{pnf}(\neg Z) = \overline{Z}(\text{a new variable}),\ \ (V(\overline{Z}) = S - V(Z)) \\
\mathrm{pnf}(\neg \mathrm{ff}) = \mathrm{tt} \\
\mathrm{pnf}(\neg \mathrm{tt}) = \mathrm{ff}
\end{gathered}$$

These rules illustrate the dualities of the operators of μM, about which we will have more to say in Section 4.4.2.

The most significant way in which μM goes beyond HML is in the *fixpoint* or σ-formulae $\nu X.\phi$ and $\mu X.\phi$ (we write σ to range over ν and μ in some contexts). With σ-formulae the logic can express very complex repetition and termination properties in a very regular way.

4.4.2 Interpretation of μM Formulae

Formulae are interpreted over LTS models, and valuation functions which interpret propositional variables as the set of LTS states at which they are taken to be true. The meaning of any formula is the subset of the states of S for which the formula is true. Free variables can be free globally (propositional constants) or free only in subformulae (fixpoint variables), as for example X is free in $[-]X$ but bound in $\mu X.[-]X$. Fixpoint variables serve to define the semantics of σ-formulae. We write $[\![\phi]\!]_V$ for the interpretation of ϕ in the context of the valuation V, and more rarely $[\![\phi]\!]_V^M$ in the context of valuation V and model M when M is not clear from the context. $V[S/X]$ is a valuation function:

$$\begin{gathered}
(V[S/X])(Y) = V(Y) \text{ iff } Y \neq X. \\
(V[S/X])(X) = S
\end{gathered}$$

Definition 4.12 (Interpretation of μM formulae)

$$
\begin{array}{rcl}
[\![X]\!]_V & \triangleq & V(X) \\
[\![\phi_1 \wedge \phi_2]\!]_V & \triangleq & [\![\phi_1]\!]_V \cap [\![\phi_2]\!]_V \\
[\![\phi_1 \vee \phi_2]\!]_V & \triangleq & [\![\phi_1]\!]_V \cup [\![\phi_2]\!]_V \\
[\![[K]\phi]\!]_V & \triangleq & \{s : K(s) \subseteq [\![\phi]\!]_V\} \\
[\![\langle K\rangle\phi]\!]_V & \triangleq & \{s : \exists s' \in K(s) \,.\, s' \in [\![\phi]\!]_V\} \\
[\![\nu X.\phi]\!]_V & \triangleq & \bigcup\{S \in \mathcal{S} : S \subseteq f_{\phi_{V,X}} S\} \\
[\![\mu X.\phi]\!]_V & \triangleq & \bigcap\{S \in \mathcal{S} : f_{\phi_{V,X}} S \subseteq S\}
\end{array}
$$

where

$$
\begin{array}{rcl}
K(s) & \triangleq & \{s' : \exists a \in K \,.\, s \xrightarrow{a} s'\} \text{ and} \\
f_{\phi_{V,X}} & \triangleq & \lambda S \,.\, [\![\phi]\!]_{V[S/X]}
\end{array}
$$

We express the ν, μ definitions in terms of $f_{\phi_{V,X}}$, a function on states. This makes it clear through a series of standard definitions and results that what are known as *fixpoint* formulae are technically the least (μ) and greatest (ν) fixpoints of the function $f_{\phi_{V,X}} S$. First of all

Fact 4.13 $f_{\phi_{V,X}} S$ is monotonic in S [Koz83]

Proof is a routine induction on ϕ.

Fact 4.14 By the Knaster-Tarski fixpoint theorem [Tar55]

$$
\begin{array}{l}
[\![\mu X, \phi]\!]_V \text{ is the least fixpoint of } f_{\phi_{V,X}} \\
[\![\nu X, \phi]\!]_V \text{ is the greatest fixpoint of } f_{\phi_{V,X}}
\end{array}
$$

Proof Application of K-T is immediate from monotonicity of $f_{\phi_{V,X}} S$ □

This result allows us to show the equivalence of positive normal forms and the negation-based form.

The most important consequence of Knaster-Tarski is the ordinal approximant view of fixpoints [Koz83], which is a vital device for understanding and elucidating μM-formulae [BS92, Bra91]. Approximants motivate and justify Bradfield's proof system [Bra91].

4.4.3 Approximants

Approximant formulae, indexed by ordinals, are also vital for writing inductive proofs about μM. We shall have some call for this method later, so present a version of approximants [Bra91]. Only a basic understanding of ordinals [Pot90] is necessary for this. We follow convention in using $\alpha, \beta, \ldots$ to range over the ordinals.

Definition 4.15 (Approximants) We add terms $\sigma^\alpha X.\phi$ to μM and set

$$[\![\sigma^\alpha X.\phi]\!]_V \triangleq a^\alpha_\sigma\phi$$

with f standing for $f_{\phi_{V,X}}$ we take

$$\begin{aligned} a^\alpha_\mu &\triangleq f(a^{\alpha-1}_\nu) && \text{(when } \alpha \text{ is a successor ordinal)} \\ &\triangleq \bigcup_{\beta<\alpha} a^\beta_\nu && \text{(when } \alpha \text{ is a limit ordinal)} \\ a^\alpha_\nu &\triangleq f(a^{\alpha-1}_\mu) && \text{(when } \alpha \text{ is a successor ordinal)} \\ &\triangleq \bigcap_{\beta<\alpha} a^\beta_\mu && \text{(when } \alpha \text{ is a limit ordinal)} \end{aligned}$$

In particular, notice that

$$\begin{aligned} [\![\mu^0.\phi]\!]_V &= \{\} \text{ for any } \phi, V(= [\![\mathrm{ff}]\!]) \\ [\![\nu^0.\phi]\!]_V &= \mathcal{S} \text{ for any } \phi, V(= [\![\mathrm{tt}]\!]) \\ [\![\mu^1 X.\phi]\!]_V &= [\![\phi]\!]_{V[[\![\mu^0 X.\phi]\!]_V/X]} = [\![\phi[\{\}/X]]\!] \end{aligned}$$

whereupon we can make the key observation of the approximant view

Fact 4.16

$$\begin{aligned} \bigcup_{\alpha\in\mathsf{Ord}} a^\alpha_\mu &= [\![\mu X.\phi]\!]_V \\ \bigcap_{\alpha\in\mathsf{Ord}} a^\alpha_\nu &= [\![\nu X.\phi]\!]_V \end{aligned}$$

Proof is standard [Bra91, Koz83]. We'll show μ; ν has a dual argument. Let g be the least fixpoint of $f_{\phi_{V,X}}$ (g exists by K-T). By induction on the ordinal β, we show

$$\forall\beta \in \mathbf{Ord}, a^\beta_\mu \subseteq g$$

- The base case is $[\![a^\mu_0]\!] = \{\} \subseteq g$.
- For $\beta + 1$ (any successor ordinal) we have $a^{\beta+1}_\mu = f(a^\beta_\mu) \subseteq g = f(g)$ by monotonicity

$$\begin{array}{ccc} a^\beta_\mu & \subseteq & g \\ \downarrow f & \downarrow \mathrm{mon} & \downarrow f \\ a^{\beta+1}_\mu & \subseteq & g \end{array}$$

- For β a limit ordinal,

$$a^i_\mu, i < \beta \Rightarrow a^\mu_i \subseteq g, \text{so} \bigcup a^i_\mu : i < \beta \subseteq g$$

But $\bigcup_{\alpha\in\mathsf{Ord}} a^\alpha_\mu$ is a fixpoint of f just because it contains all ordinals; the above argument shows $\bigcup_{\alpha\in\mathsf{Ord}} a^\alpha_\mu = g$, the least fixpoint □

4.5 Tableau Proofs

Stirling [Sti87] has argued for using semantic tableaux to write μM proofs. Bradfield [Bra91], building on [SW91] introduced a general tableau system, usable for writing proofs about infinite systems. Tableau methods are local; rather than systems for adducing the truth or falsity of a formula in all states, they are systems for adducing the truth of a formula in a set of states by exploring subformulae on other states on demand. Tableaux for μM cannot be derived automatically; for a sufficiently powerful model language, correctness of μM-formulae is undecidable. Bradfield [Bra91] shows this for Petri Nets by encoding the halting problem as an instance of the μM decision problem, and the same thing can be done for CCS; Milner presents a CCS Turing Machine in [Mil89].

So tableaux must be constructed manually. This is no more than we should expect, and turns out to be a reasonable thing to do for someone who understands a system; a designer's knowledge is a very powerful tool for generating proofs. Then a tableau for us stands as a pleasant encoding of such knowledge, and more actively the goal-directed generation of a tableau is the easiest way to go about extracting this knowledge. All that I do later in this thesis can be seen as attempting to recode the same knowledge to a different purpose. Such a task of course demands that we ask questions about what the knowledge is that we encode.

4.5.1 Bradfield Tableaux

Bradfield's tableau system [Bra91] is a goal-directed (or top-down) system for validating sequents of the form $S \vdash_\Delta \phi$

S are sets of states of an LTS $M = \langle S, A, T \subseteq S \times A \times S \rangle$

ϕ are μM properties

Δ are lists of variable definitions introduced at fixpoints

Usually we start with a goal $S \vdash_{()} \phi$ in order to validate ϕ for S in the absence of enclosing fixpoints.

Definition 4.17 (Tableaux) are constructed according to the rules of Figure 4.4. These are the precise set of rules by which the tableaux of Section 3.4 are constructed. The rules are used to extend the tableau-in-progress at any leaf node until all leaves are of very simple type, having one of the forms

1. $S \vdash_\Delta X$ if X is free in ϕ of the root node
2. $S \vdash_\Delta \langle K \rangle \phi$ if $\exists s \in S : \neg s \xrightarrow{a \in K}$
3. $\{\} \vdash_\Delta \phi$
4. $S \vdash_{\Delta'} U$ where $S' \vdash_\Delta U$ is an ancestor and $S \subseteq S'$

It is possible to further extend the tableau when nodes are of the form $S \vdash_\Delta U$, and the rules for correct tableaux take such extensions into account.

Figure 4.4 Tableau construction rules

$$\frac{S \vdash_\Delta \phi_1 \wedge \phi_2}{S \vdash_\Delta \phi_1 \quad S \vdash_\Delta \phi_2} \wedge$$

$$\frac{S \vdash_\Delta \phi_1 \vee \phi_2}{S_1 \vdash_\Delta \phi_1 \quad S_2 \vdash_\Delta \phi_2} \vee \qquad S_1 \cup S_2 = S$$

$$\frac{S \vdash_\Delta [K]\phi}{S' \vdash_\Delta \phi} [K] \qquad S' = \{s' : \exists a \in K, s \in S \,.\, s \xrightarrow{a} s'\}$$

$$\frac{S \vdash_\Delta \langle K\rangle\phi}{S' \vdash_\Delta \phi} \langle K\rangle \qquad S' = \bigcup_{s\in S}\{f(s)\}, f : \forall s \,.\, f(s) \subseteq \{s' : a \in K, s \xrightarrow{a} s'\} \text{ and } f(s) \neq \{\}^*$$

$$\frac{S \vdash_\Delta \sigma X.\phi}{S \vdash_{\Delta'} U} \sigma = \nu, \mu \qquad \Delta' = \Delta[\sigma X.\phi/U], U \notin \mathrm{Dom}(\Delta)$$

$$\frac{S \vdash_\Delta U}{S \vdash_\Delta \phi[U/X]} (U) \qquad \Delta(U) = \sigma X.\phi \ (\sigma = \nu, \mu)$$

$$\frac{S \vdash_\Delta \phi}{T \vdash_\Delta \phi} \text{Thin} \qquad S \subseteq T$$

(*)The function f which generates S' for $\langle - \rangle$-rule is any function which generates at least one valid successor for each state s. Of course if no such function exists, the success rules will tell us (Definition 4.18) that the tableau fails.

Tableaux are successful if and only if all their leaves are successful:

Definition 4.18 (Leaf Criteria) (see Definition 4.17 for leaf forms)

Success
The leaf is type 1 ($S \vdash_\Delta X$) and $S \subseteq V(X)$
The leaf is type 3 (empty goal set)
The leaf is type 4 ($S \vdash_\Delta U$),$\Delta(U) = \nu X.\psi$ and ν-conditions succeed. (Section 4.5.1.1)
The leaf is type 4 ($S \vdash_\Delta U$),$\Delta(U) = \mu X.\psi$ and μ-conditions succeed. (Section 4.5.1.2)
Failure
The leaf is type 1 and $S \not\subseteq V(X)$
The leaf is type 2 ($S \vdash_\Delta \langle K \rangle \psi$ and $\exists s \in S : \neg s \xrightarrow{a \in K}$)
The leaf is type 4 ($S \vdash_\Delta U$),$\Delta(U) = \nu X.\psi$ and ν-conditions fail.
The leaf is type 4 ($S \vdash_\Delta U$),$\Delta(U) = \mu X.\psi$ and μ-conditions fail.

4.5.1.1 ν-conditions

A leaf node $S \vdash_\Delta U$, where $\Delta(U) = \nu X.\psi$ is successful iff it has an ancestor node $S' \vdash_{\Delta'} U$ and for that node, $S \subseteq S'$. The ancestor node expresses the goal that $S' \subseteq [\![\psi_{\Delta[S'/X]}]\!]$ and the tableau rules render one of the subgoals of that as $S \subseteq [\![X]\!]_{\Delta[S'/X]}$, so that the simple criterion for the leaf node demonstrates the admissibility of the fixpoint formula at the ancestor.

Informally, the leaf node reached through successive application of tableau rules to the ancestor has resulted in a goal set (S) for the formula U, no larger than the original goal set (S').

4.5.1.2 μ-conditions

The μ-fixpoint condition has extra complications beyond that of ν. First an ancestor node must be identified as for the ν-fixpoint; this corresponds to ν-fixpoints being subsets of μ-fixpoints.

In addition, we must trace the individual LTS states in the ancestor node ($s' \in S'$) through the tableau to states in the leaf node ($s \in S$) and back to ancestor nodes ($s \in S' \text{or} S''$) and verify the eventual termination of all such paths through the tableau. The definitions of [Bra91], which we replicate here, make this precise; they are not easy, and to justify them we must fall back on the not quite circular reasoning that least fixpoints capture termination well and that the tableau system with these rules is sound and complete; see Fact 4.22.

Definition 4.19 (Companion node) For a leaf node,n, of the form $S \vdash_\Delta U$, its companion is its nearest ancestor node, n',of the form $S' \vdash_{\Delta'} U$.

Thus leaf node and companion share a formula, U, which is an introduced variable standing for the fixpoint formula $\mu X.\psi$.

The additional correctness rule for the leaf node n is that there exists a particular ordering relationship, a well-order, on the elements of S'. A crucial

feature of a well-order is that it does not contain cycles. We call the well-order $\sqsubset$ and we write $s \sqsubset s'$ if s is related to s' in the well-order. The well-order must respect the relationship

Definition 4.20 $\longrightarrow$ is the transitive closure of the relationship

Definition 4.21 $(\rightarrow)$ $(s_p, n_i) \rightarrow (s_q, n_j)$, $n_i = S_i \vdash_\Delta \Delta_i \phi_i$, $n_j = S_j \vdash_{\Delta_j} \phi_j$ if and only if

either

1 $\dfrac{S_i \vdash_{\Delta_i} \phi_i}{\ldots S_j \vdash_{\Delta_j} \phi_j \ldots}$ is a fragment of the tableau, and

2 $s_p \in S_i$ and $s_q \in S_j$ and

3 $\phi_i = \langle [K] \rangle \phi_j$ and $s_i \xrightarrow{a \in K} s_j$ or

ϕ_i is any other formula, and $s_i = s_j$

or

1 $S_i \vdash_{\Delta_i} \phi_i$ is a leaf node, and

2 $S_j \vdash_{\Delta_j} \phi_j$ is its companion, and

3 $s_p = s_q$ and

4 n' is an ancestor of $S_j \vdash_\Delta \phi_j$

The relationship between well-order and $\longrightarrow$ is

$$(s_1, n') \longrightarrow (s_2, n') \Rightarrow s_2 \sqsubset s_1$$

Then the correctness condition for a least fixpoint leaf is that such a well-order on the state set S' of the leaf's companion node exists. Informally, the existence of a well order related to the possible traces of states in the set S' shows that behaviours described by the least-fixpoint formula will terminate. if they were to repeat then there could not be a well-order.

Where a human is developing a proof in the tableau notation, the complexity of the least-fixpoint correctness rule unsurprisingly causes the most difficulties in proving. It takes time to learn how to express even well-understood reasons for termination in terms of the appropriate well orders. But the tableau least-fixpoint rules can be viewed as meta-rules to apply to any modelling language, and as we discuss in Chapter 8, there are possibilities for automatic treatment of some families of models.

4.5.2 Tableau Facts

A tableau is successful iff all of its leaves are successful. The key result is that the system of tableaux constructed according to the rules of Figure 4.4 and with all leaves successful according to Definition 4.18 is sound and complete.

Fact 4.22 (Bradfield) A successful tableau (all leaves successful, μ-conditions met) exists for $S \vdash_{()} \phi$ iff $S \models_V \phi$ (iff $S \subseteq [\![\phi]\!]_V$) □

Fact 4.22 is sufficient for a human user of a tableau system to be confident of developing proofs of interesting properties. But in fact, tableaux of a particularly restricted form are enough:

Definition 4.23 A tableau is of degree n if no single fixpoint variable U is unfolded (by applying the U-rule to $S \vdash_\Delta U$) more than n times.

Fact 4.24 (Bradfield) The degree 1 tableau system is complete

Bradfield's completeness proof is actually more particular, and shows that only one application of the *Thin* rule is necessary per fixpoint-variable; his canonical tableau, which exists for each true formula, has such a form. The form of a canonical tableau is entirely determined by the formula in question. For example

$$S_0 \vdash \phi \triangleq \nu X.[A]\mu Y.\langle B\rangle X \vee [C]Y \vee P$$

has a canonical tableau formed by instantiating the S_i in the following *meta-tableau.*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{S_0 \vdash \nu X.[A]\mu Y.\langle B\rangle X \vee [C]Y}{S_0 \vdash U_0}(\text{Def})}{S_1 \vdash U_0}(\text{Thin})}{S_1 \vdash [A]\mu Y.\langle B\rangle U_0 \vee [C]Y}(\text{Un})}{S_2 \vdash \mu Y.\langle B\rangle U_0 \vee [C]Y}(\text{H})}{S_2 \vdash U_1}(\text{Def})}{S_3 \vdash U_1}(\text{Thin})}{S_3 \vdash \langle B\rangle U_0 \vee [C]U_1}(\text{Un})}{\dfrac{S_4 \vdash \langle B\rangle U_0}{S_5 \vdash U_0}(\langle\rightarrow\rangle) \quad \dfrac{S_6 \vdash [C]U_1}{S_7 \vdash U_1}(\text{H})}(\vee)$$

and the problem of proving ϕ for a particular S_0 becomes equivalent to finding a satisfactory instantiation of the S_i. $S_0 \vdash \phi$ just in the case where there is such an instantiation. In Chapter 8 we use the instantiation of canonical tableaux to develop a technique for computer-aided proof construction.

4.6 Conclusion

In this chapter I have described the components of a framework for making formal proofs about system specifications. This can be considered as just a technical necessity in order to have concrete systems in which to work.

- *CCS* to write specifications of concurrent systems

- *Modal-mu calculus* to define behavioural properties of systems
- *Tableaux* to develop and encode proofs of properties of systems

I have detailed the particular instance of each component which I choose to work with. None of the choices is meant to be particularly novel by itself. Rather it is the connected set of components which is of interest. We can view the proof of a property of a specification as an object, and with this perspective we can try, for instance, to use it as (or transform it into) a program testing device.

We may also be able to think of other uses for such a proof object. First, we look at proofs from another perspective, that of games, both for the sake of general insight and as an attempt to smooth the path to a nice representation of proof objects.

Chapter 5

Presenting Proofs

Summary

Tableaux are one formal device for representing proofs. They play a crucial role in proof *development*. However, other notions may be more natural in other contexts. Model-checking games [Sti97, Sti95] have been proposed as a useful way of explaining the meaning of μM properties. In particular, playing strategies can be presented as proofs or refutations of properties.

In this chapter I

- Describe the formal concept of a model-checking game, and the key idea of a strategy for a player to win a game.
- Argue for games as practical tools for explaining why properties hold.
- Take the connection between strategies for games, and proofs of properties, to motivate a strategy-like encoding of proofs. I call this encoding a verity.

Verities (with some refinement in Chapter 6) are what I use as the data for defining oracles.

5.1 Games

Games characterise the model-checking of μM formulae as playing an adversarial game. Games consider whether a single state satisfies a formula, and are closely linked to tableaux. Tableaux can be seen as a succinct encoding of why sets of states satisfy formulae. Then games complement this by providing a much more explicit analysis of why single states satisfy formulae. The complex least-fixpoint success conditions in tableaux (Section 4.5.1.2) suggest that explicit analysis of individual states is unavoidable.

The game based characterisation of the truth of μM formulae, particularly in the interaction of multiple nested fixpoints, turns out to be more comprehensible and thus helpful to understanding how fixpoints interact.

A game is also an effective weapon for explaining and supporting a tableau proof. Where the proof is in any way complicated, a doubter may be convinced by being challenged to take part in a game. A player armed with a correct proof can win a game whatever strategy is adopted by her opponent (Theorem 5.8). If the opponent uses her moves to express her doubts about the proof, asking "What happens if I make this move ?" then the player armed with the proof can answer "I still win, in this fashion..." and can keep refuting the doubter until a configuration is reached where the doubter has no questions left to ask.

The requirement for auditioning is to have a proof representation that can tractably be used as an oracle. Chapter 3 leads us to think that structuring the proof as a transition system is a good way to go about this, though the informal analysis there is unable to resolve how best to cope with nodeterministic choice of successor state which can occur. Games can be considered as operational encodings of proofs, so they ought to be a good starting point for a concrete transition-oriented oracle.

5.2 Formal Games

A game is a contest between 2 players over a sequence of *game configurations*. In a game there must be

- A clear way to establish which player has the next *move*. This can be based on the history of the game (you moved last, so it's my turn), solely on the current configuration, or some combination of both.
- Rules to determine when a game has finished. These might be when a particular configuration is reached, or is repeated, or when a player cannot move.
- Rules to establish who has won. When a game has finished, the configuration and the history is examined to determine a winner. Some games may be drawn. Frequently a player attempts to force the game into a particular configuration, whence it is fairly obvious that the player who succeeds in this is the winner.

5.2.1 Model-Checking Games

We'll now define the model-checking games of [Sti97].

Configuration is a combination of a state and a subformula.

Player to Move depends on the leading connective of the current subformula. Player 1 attempts to refute the formula, so for instance Player 1 has the move where the formula is a conjunction; she chooses the subformula

with which to challenge the other player. Player 2 attempts to support the formula, so has the move for instance at a disjunction.

Move is to select a successor state on a transition formula, or the same state when decomposing propositionally. The formula component of the new configuration is the selected subformula.

History is important. Repetitions determine whether a game has finished, and a fine analysis of history is needed to determine the winner.

Finish Atomic formulae or inability to move, plus infinite repetitions of fixpoints.

To describe a model-checking game configuration, we need an LTS $\langle S, A, T\rangle$, a formula ϕ of μM and states $s \in S$. We restrict ourselves to ϕ in which all binding names X of $\sigma X.\psi$ are distinct from each other and from the free names of ϕ. Any formula can be α-converted into this form. We also have a valuation function $V : X \to 2^S$ which gives valuations for the free names in ϕ. Configurations of the game are concerned with sub-formulae of ϕ.

Definition 5.1 [Sub formula] A formula ψ is a sub-formula of ϕ ($\psi \prec \phi$) iff

$$\begin{array}{ccc}
\psi \prec \psi & & (\phi = \psi) \\
\psi \prec \psi \wedge \psi_2 & \psi \prec \psi_2 \wedge \psi & (\phi = \psi \wedge \psi_2 \text{ or } \psi_2 \wedge \psi) \\
\psi \prec \psi \vee \psi_2 & \psi \prec \psi_2 \vee \psi & (\phi = \psi \vee \psi_2 \text{ or } \psi_2 \vee \psi) \\
\psi \prec [K]\psi & & (\phi = [K]\psi) \\
\psi \prec \langle K\rangle\psi & & (\phi = \langle K\rangle\psi) \\
\psi \prec \nu X.\psi & & (\phi = \nu X.\psi) \\
\psi \prec \mu X.\psi & & (\phi = \mu X.\psi)
\end{array}$$

or they are related by transitive closure of the above rule:

$$\psi \prec \phi \text{ if } \exists \phi' : \psi \prec \phi' \text{ and } \phi' \prec \phi$$

Because all names are chosen to be distinct, we can distinguish any ψ s.t. $\sigma X.\psi \prec \phi$ by the binding-name X. We shall refer to the ψ as ψ_X. We are now in a position to describe how to play the game.

Definition 5.2 (Player to Move) A state and sub-formula (together a [game] configuration) are transformed to another configuration according to the following rules; the indicated player chooses which of the successor configurations will be selected. A number of moves are completely forced (there is necessarily exactly one successor configuration), and these can arbitrarily be assigned to either

player to fit them into the standard game scheme. We prefer to present them separately.

Player 1	Player 2
$(s, \phi_1 \wedge \phi_2) \rightarrow (s, \phi_{i=1 \text{ or } 2})$	$(s, \phi_1 \vee \phi_2) \rightarrow (s, \phi_{i=1 \text{ or } 2})$
$(s, [K]\phi) \rightarrow (s', \phi) : s \xrightarrow{a \in K} s' \in T$	$(s, \langle K \rangle \phi) \rightarrow (s', \phi) : s \xrightarrow{a \in K} s' \in T$

Forced
$(s, \nu X.\phi_X) \rightarrow (s, \phi_X)$
$(s, \mu X.\phi_X) \rightarrow (s, \phi_X)$
$(s, X) \rightarrow (s, \phi_X)$

Definition 5.3 (Play) is a sequence of game configurations where the transformation from the n-th to the $n+1$-st configuration in the sequence (for all n within the length of the sequence) constitutes a valid move according to the move rules.

A game finishes when a move transforms a play into a **won** play. For simple connectives a play may be won according solely to the final configuration, but for fixpoints previous configurations must be considered in order to determine whether a play has in fact yet won, and if so who the winner is; this is like checking a chess game for previous instances of the same board layout, which determines a draw by repetition.

Definition 5.4 (Simple Wins)

Configuration	Winner
$(s, [K]\phi)$ and $\neg \exists a \in K, s' \in S \,.\, s \xrightarrow{a} s'$	Player 2
$(s, \langle K \rangle \phi)$ and $\neg \exists a \in K, s' \in S \,.\, s \xrightarrow{a} s'$	Player 1
(s, tt) (abbreviates $\nu X.X$)	Player 2
(s, ff), (abbreviates $\mu X.X$)	Player 1
(s, X) with X free in ϕ and $s \in V(X)$	Player 2
(s, X) with X free in ϕ and $s \notin V(X)$	Player 1

In order to present fixpoint termination rules we must imagine games which continue to infinity.

Lemma 5.5 (Stirling [Sti97]) Given the partial order defined by $\prec$, there is a greatest ψ_X which forms the ϕ-component of a configuration infinitely often in an infinite play. We use *top* to define a dominant formula, and *infoft* to express infinite an inifinitely recurring dominant formula.

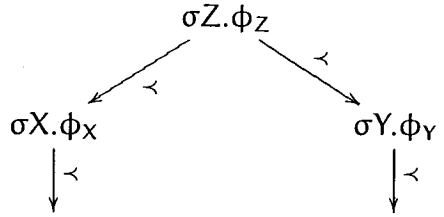
$$\mathrm{infoft}(X) \Leftrightarrow \forall m \in \mathrm{Nat}, \exists n \in \mathrm{Nat} : n > m \text{ and } \phi_n = \sigma X.\phi_X$$
$$\mathrm{top}(X) \Leftrightarrow \forall Y : \mathrm{infoft}(Y)\ ,\ \sigma Y.\phi_Y \prec \sigma X.\phi_X$$

Theorem 5.6 There is a unique X for which top(X) holds (call it $\hat{X}$).

Proof There are only finitely many ϕ to take part in configurations. So at least one must occur infinitely often in an infinite play. ($\exists X$:infoft(X)). Now the set

$$\{X : \text{infoft}(X)\}$$

has a top element, because of the tree structure of formulae



and the fact that configuration formulae are calculated by breaking down one connective at a time:

$$\text{infoft}(X),\ \text{infoft}(Y), X \not\preceq Y \text{ and } Y \not\preceq X \Rightarrow \exists Z : X \prec Z, Y \prec Z \text{ and infoft}(Z)$$

So we have a $\hat{X}$ □

Definition 5.7 (Complicated wins) The winning condition for infinite plays is

$\hat{X}$	Winner
$\nu\hat{X}.\psi$ ($\hat{X}$ denotes g.f.p.)	Player 2
$\mu\hat{X}.\psi$ ($\hat{X}$ denotes l.f.p.)	Player 1

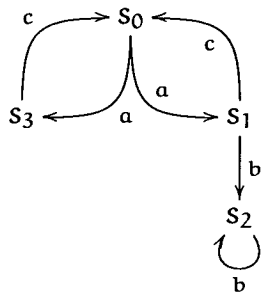
Example Game

Consider the formula

$$\nu X.[a](\mu Y.[b]Y \vee \langle c\rangle X)$$

which in English tells us that any a-action leads to a state where a c-action returning to the beginning is possible immediately or after some number of b-actions (Clearly, once the formal calculus is understood it is much more precise and unamiguous than prose).

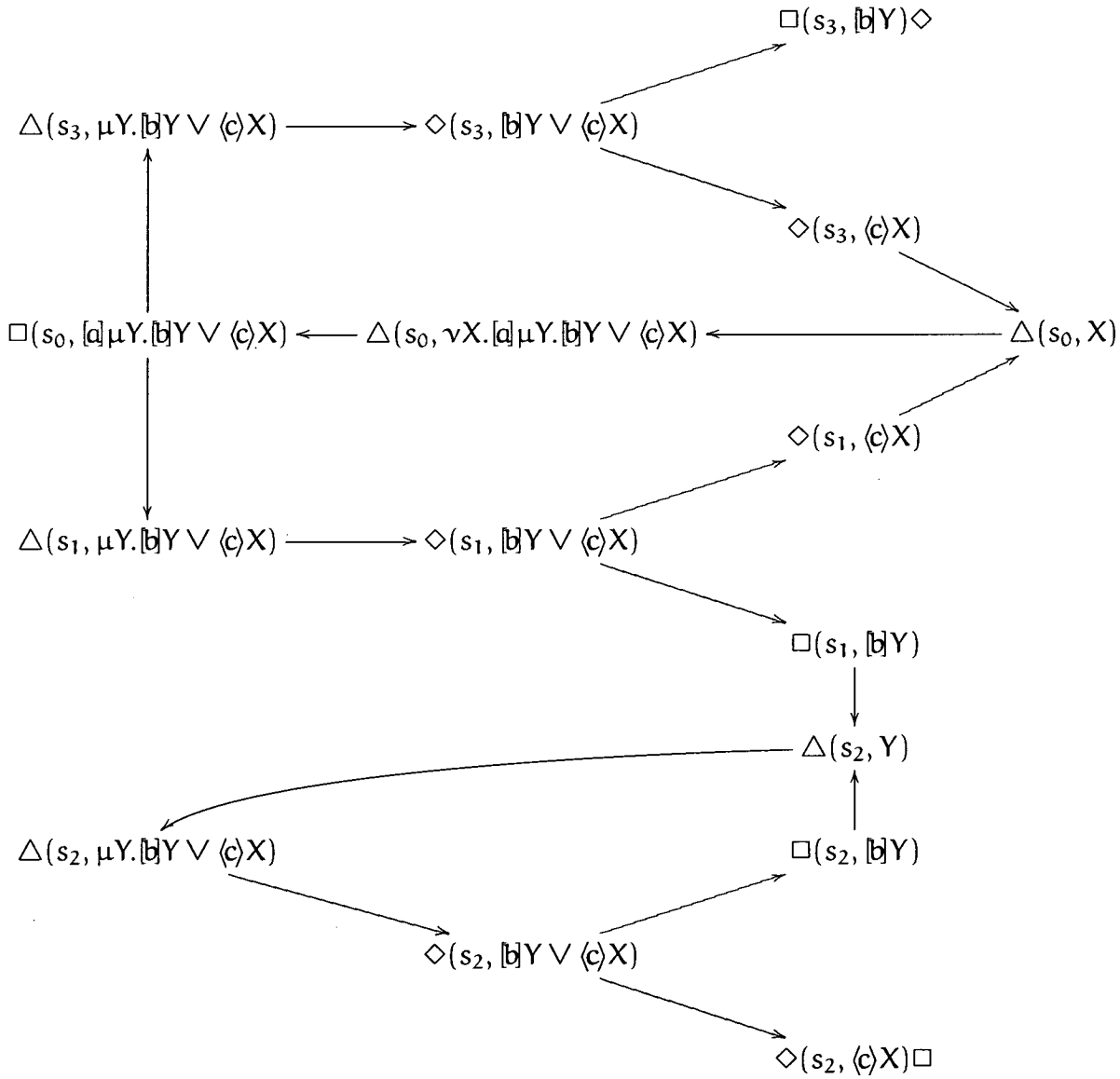
We interpret the formula on the model:



We prefix the game graph states with symbols to show which player makes choices at which node. We use the symbols as suffixes to denote winners.

$\Box$ Player 1 $\quad \Diamond$ Player 2 $\quad \triangle$ A forced move

For example, $\Box(s, \mathrm{tt})\Diamond$ denotes that Player 1 chooses in this state and that it is a terminating state in which Player 2 wins. Then the game graph is



Observe that there are a few terminating entries, where the winner is directly known. Observe also that the formula is true for s_0. Let's play along to see this:

-At $(s_0, \nu X)$ the move to $(s_0, [a]\mu Y)$ is forced

-At $(s_0, [a]\mu Y)$ Player 1 can choose $(s_1, \mu Y)$ or $(s_3, \mu Y)$

-These force moves to $(s_1, [b]Y \vee \langle c\rangle X)$ and $(s_3, [b]Y \vee \langle c\rangle X)$ respectively

We shall ignore $(s_3, [b]Y \vee \langle c\rangle X)$ as it's even easier to show than is $(s_1, [b]Y \vee \langle c\rangle X)$

-At $(s_1, [b]Y \vee \langle c\rangle X)$ Player 2 (her choice at $\vee$) chooses to move to $(s_1, \langle c\rangle X)$

-At $(s_1, \langle c\rangle X)$ Player 2 (her choice at $\langle - \rangle$) chooses to move to (s_0, X)

-At (s_0, X) the move to $(s_0, \nu X)$ is forced

Clearly Player 2 can force the game to return to $(s_0, \nu X)$ as long as she makes the right responses to Player 1's moves. So she can always make $\hat{X}$ be $\nu X\psi$ rather than $\mu Y.\psi$, for some ψ. In contrast notice that if we removed the transition

$$s_1 \xrightarrow{c} s_0$$

then Player 1 could enforce a play where $\hat{X}$ is $\mu Y \ldots$ and thus win.

Characterisation of Truth as Strategy

The characterisation theorem is now presentable, if we take a strategy to be a function which tells us (s', ψ') whenever Player 2 must move and has a choice of $(s', \psi') : (s, \psi) \rightarrow (s', \psi')$.

Theorem 5.8 (Stirling [Sti97])

$$s \models \phi \Leftrightarrow \text{Player 2 has a winning strategy for the game } (s, \phi)$$

Proof Sketch This can be proven by showing, using the approximant version of μ-fixpoints, that Player 2 can force all μ-fixpoints *above* the top ν-fixpoint to be traversed to reach a mode where this top ν-fixpoint is repeated indefinitely (and Player 2 wins) or Player 1 allows the ν-fixpoint to be traversed, whereupon Player 2 forces the game down again to the next ν-fixpoint, and inductively to a trivial win □

The fact that Player 2 can make sure there is no possible $\hat{X}$ which denotes a μ-fixpoint (P_2 having a winning strategy) corresponds to a tableau which satisfies the μ-success condition, and is also of course intimately connected to termination of tracing (Theorem 6.44), where we make a fully-detailed approximant-based proof.

5.3 Game Graphs and Fixpoints

We can use game graphs to take a different view of the interpretation of formulae and of the expression of algorithms for checking proofs or calculating denotations. The standard (exponential) algorithm for calculating fixpoints on models

with finite statesets has as its basic step a monolithic function on statesets, the application of

$$f_{\phi V,X} \text{ to } S_i \text{ yielding } S_{i+1}$$

in the iteration process from $\{\}$ to calculate the least fixpoint or from $\mathcal{S}$ to calculate the greatest fixpoint. The justification for the algorithm's correctness is Fact 4.16.

Instead, we consider the game graph of M, ϕ.

Definition 5.9 (Model-checking Game Graph) $G = \langle V, E \rangle$ is a directed graph with vertex set

$$V = \{(s, \psi) : s \in S, \psi \prec \phi\}$$

and edge set

$$E = \{(s, \psi), (s', \psi') : (s, \psi) \rightarrow (s', \psi')\}$$

To use the game graph we need to be able to assign validity at each node, thus we must understand the meaning of *open* formulae ψ whenever $\psi \prec \phi$. The *closed* formula which allows this is just the root formula ϕ of original interest. We extend $[\![-]\!]$ to open formulae (ψ with free names) by interpreting them in the context of closed formulae $\phi : \psi \prec \phi$. Closed formulae are interpreted in the context of a model $M = \langle S, A, T \rangle$ and a valuation function on propositional names X, $V(X) \longrightarrow 2^S$ to give

$$[\![\phi]\!]^M_V$$

To interpret open formulae we inductively defined a series of sets which are the denotations of closed formulae, and which stand to solve each of the n fixpoint variables $X_i \preceq \phi$ in the open version:

Definition 5.10

$$\begin{aligned} S_1 &= [\![\sigma X_1.\psi_1]\!]_V \\ S_2 &= [\![\sigma X_2.\psi_2]\!]_{V[S_1/X_1]} \\ &\cdots \\ S_n &= [\![\sigma X_n.\psi_n]\!]_{V[S_1/X_1, S_2/X_2, \ldots, S_{n-1}/X_{n-1}]} \end{aligned}$$

where

$$\sigma X_i.\psi_i \prec \phi \text{ and } i < j \Rightarrow \sigma X_i.\psi_i \not\prec \sigma X_j.\psi_j$$

so that S_i definitions are closed applications of $[\![-]\!]$

Now we make the open definitions using the S_i as a base; we build a context ρ from the S_i

$$\rho \triangleq \langle M, V, S_1, S_2, \ldots, S_n \rangle$$

The idea is simply to calculate the true values of the valid set for each fixpoint subformula, and to use these values to define $[\![-]\!]$ for open subformulae.

Definition 5.11 (Open $[\![-]\!]$)

$$\begin{aligned}
[\![Z]\!]\rho &= V(Z) \\
[\![X_i]\!]\rho &= S_i \\
[\![\psi_1 \wedge \psi_2]\!]\rho &= [\![\psi_1]\!]\rho \cap [\![\psi_2]\!]\rho \\
[\![\psi_1 \vee \psi_2]\!]\rho &= [\![\psi_1]\!]\rho \cup [\![\psi_2]\!]\rho \\
[\![[K]\psi]\!]\rho &= \{s : a \in K \text{ and } s \xrightarrow{a} s' \Rightarrow s' \in [\![\psi]\!]\rho\} \\
[\![\langle K\rangle\psi]\!] &= \{s : \exists a \in K, s' \in S \,.\, s \xrightarrow{a} s' \text{ and } s' \in [\![\psi]\!]\rho\} \\
[\![\nu X_i.\psi]\!]\rho &= S_i \\
[\![\mu X_i.\psi]\!]\rho &= S_i
\end{aligned}$$

We write $s \models_\rho \psi$ as shorthand for $s \in [\![\psi]\!]\rho$. In order to ensure the validity of the extension to open formulae, we need to see that it coincides with the original definition of $[\![-]\!]$ for closed formulae.

Theorem 5.12 $[\![\psi]\!]\rho = [\![\psi]\!]^M_V$ if ψ is closed.

Proof We show by induction on open formulae (ψ) that

$$[\![\psi]\!]\rho = [\![\psi]\!]^M_{V[S_j/X_j:\psi \prec \sigma X_j.\psi_j]}$$

i.e. the fully open semantics (with ρ) coincides with the semantics where ψ itself is closed, and only its references to containing fixpoints are defined in the valuation set.

$[X_i]$	$[\![X_i]\!]\rho = S_i$
	$[\![X_i]\!]^M_{V[S_j/X_j:\psi \prec \sigma X_j.\psi_j]} = S_i$(by S_i definition)
$[\wedge, \vee, [\dashv], \langle\dashv\rangle]$	trivially
$[\nu X_i.\psi_i]$	$[\![\nu X_i.\psi_i]\!]\rho = S_i$
	$[\![\nu X_i.\psi_i]\!]^M_{V[S_j/X_j:\psi \prec \sigma X_j.\psi_j]}$
	$= [\![\nu X_i.\psi_i]\!]^M_{V[S_1/X_1,\ldots,S_{i-1}/X_{i-1}]}$
	$= \bigcup\{S : S \subseteq [\![\psi_i]\!]_{V[S_1/X1,\ldots,S_{i-1}/X_{i-1}][S/X_i]}\}$
	$= S_i$ (by S_i definition)
$[\mu X_i.\psi_i]$	exactly as for ν □

In order to use this to check the soundness of the game moves we need a small lemma for traversing fixpoints

Lemma 5.13 $[\![\nu X_i.\psi_i]\!]\rho = [\![\psi_i]\!]\rho$

Proof

$$\begin{aligned}
[\![\nu X_i.\psi_i]\!]\rho &= S_i \text{ (defn)} \\
[\![\psi_i]\!]\rho &= [\![\psi_i]\!]^{M}_{V[S_1/X_1,\ldots,S_i/X_i]} \text{ (Theorem 5.12)} \\
&= (\lambda S.[\![\psi_i]\!]^{M}_{V[S_1/X_1,\ldots,S_{i-1}/X_{i-1}][S/X_i]})S_i \\
&= S_i \text{ as } S_i \text{ is a fixpoint of the function } \square
\end{aligned}$$

Defining $[\![-]\!]$ for open formulae allows us to check the soundness of the game-playing rules.

Theorem 5.14 (Game-Soundness - Stirling [Sti97])

True Player 2 configurations $[\psi = \vee, \langle K \rangle]$ have a true successor

$$s \models \psi \Rightarrow \exists (s', \psi') \,.\, (s, \psi) \rightarrow (s', \psi') \text{ and } s' \models \psi'$$

False Player 1 configurations $[\psi = \wedge, [K]]$ have a false successor

$$s \not\models \psi \Rightarrow \exists (s', \psi') \,.\, (s, \psi) \rightarrow (s', \psi') \text{ and } s' \not\models \psi'$$

Forced configurations $[\psi = \rho X.\psi_X, X]$ have a unique and equivalent successor

$$s \models \psi \Leftrightarrow \exists! (s', \psi') \,.\, (s, \psi) \rightarrow (s', \psi') \text{ and } s' \models \psi'$$

We can rephrase this with some dualisation to see how all games connect to one proof

Corollary 5.15

Player 2 $[\vee, \langle K \rangle, \mathtt{ff}]$

$$s \models \psi \Leftrightarrow \exists (s', \psi') \,.\, (s, \psi) \rightarrow (s', \psi') \;,\; s' \models \psi'$$

Player 1 $[\wedge, [K], \mathtt{tt}]$

$$s \models \psi \Leftrightarrow \forall (s', \psi') \,.\, (s, \psi) \rightarrow (s', \psi') \;,\; s' \models \psi'$$

Forced $[\sigma, X]$

$$s \models \psi \Leftrightarrow \exists! (s', \psi') \,.\, (s, \psi) \rightarrow (s', \psi') \;,\; s' \models \psi'$$

These results motivate the choice of a structure to represent proofs in a way that is more applicable to the auditioning process.

5.4 Verities

Winning strategies for Player 2 demonstrate the truth of properties (Theorem 5.8). We introduce the concept of a graph which is a sub-graph of the game graph, constrained to only explore the game space within the bounds to which Player 2 can restrict it by playing with a particular strategy. So we define a verity to be a true, closed subgraph.

A verity captures the subformulae on which truth of a formula depend in the same way as a tableau proof does. It provides an explicit and neutral encoding of the information in a proof, so that when we are being abstract about proofs and say *proof* we can concretely substitute *verity*.

Definition 5.16 (Verity) A verity is a graph (a pair of a vertex set V and an edge set E) together with a context ρ (which we often omit), thus $\langle V, E\rangle$ or $\langle V, E, \rho\rangle$, and subject to the following properties

$$
\begin{aligned}
&(s,\psi) \in V \Rightarrow\ s \models_\rho \psi \\
&\quad (s,\psi), \psi \text{ a Player 1 formula } [\Box, \wedge, \mathtt{tt}] \Rightarrow \\
&\qquad \forall (s',\psi') : (s,\psi) \rightarrow (s',\psi') \,.\, (s',\psi') \in V\ ,\ ((s,\psi),(s',\psi')) \in E \\
&\quad (s,\psi), \psi \text{ a Player 2 formula } [\Diamond, \vee, \mathtt{ff}] \Rightarrow \\
&\qquad \exists (s',\psi') : (s,\psi) \rightarrow (s',\psi') \,.\, (s',\psi') \in V\ ,\ ((s,\psi),(s',\psi')) \in E \\
&\quad (s,\psi), \psi \text{ a Forced formula } [\sigma, X] \Rightarrow \\
&\qquad \exists! (s',\psi') : (s,\psi) \rightarrow (s',\psi') \,.\, (s',\psi') \in V\ ,\ ((s,\psi),(s',\psi')) \in E
\end{aligned}
$$

A $\langle V, E, \rho\rangle$ is a candidate verity until it is shown to have these properties.

Theorem 5.17 (Verity Existence) Every true formula is contained in some verity, formally:

$$\forall s, \phi, \rho : s \models_\rho \phi \,.\, \exists V, E : (s,\phi) \in V\ ,\ \langle V, E, \rho\rangle \text{ is a verity}$$

Proof By Theorem 5.15, $\langle \{(s,\psi) : \psi \prec \phi \text{ and } s \models \psi\}, \{((s,\psi),(s',\psi')) : (s,\psi) \rightarrow (s',\psi')\}\rangle$ is a verity. □

Theorem 5.15 is also the justification that a verity contains all the necessary information to encapsulate the proof of a property. We can rephrase it more directly as

Corollary 5.18

$$
\begin{aligned}
&(s,\psi) \in V \text{ and} \\
&(\forall (s',\psi') : ((s,\psi),(s',\psi')) \in E \,.\, s' \models \psi')
\end{aligned}
\quad \Rightarrow\ \ s \models \psi
$$

From here we will use $(s,\psi) \rightarrow (s',\psi')$ to denote $((s,\psi),(s',\psi')) \in E$, as we shall have little need to refer further to games.

5.5 Conclusion

In this chapter I have introduced verities. Verities are the central formal concept of the thesis. Verities are the most neutral concept of proof object which I have been able to define. They are most easily seen as subgraphs of game graphs, although they can also be considered as translations from tableaux; this is necessary because we tend to write proofs using tableaux.

The chapter has discussed model checking games in order to show where verities have developed from; in addition games are a fresh and insightful way of looking at how μM properties relate to systems.

I now proceed to apply verities by making them into oracles.

Chapter 6

Formalising Oracles

Summary

The verities defined in the last chapter allow us to make formal the oracles described in Chapter 3. In this chapter I show how to turn verities into oracles.

- I compare verities by defining trace languages for them.
- I give traces the power to declare acceptable sets of states. This forces annotations to take account of states, but it is an effective if brutal method of allowing oracles to spot invalid traces.
- I define verity configurations which contain possible model states and properties which must hold of particular model states. I show how these configurations track the system by defining their derivatives under trace actions.
- I modify traces to declare properties rather than states, and use the properties declared by traces to prune verity configurations, removing components of configurations which are contradicted by the declared properties.
- With the notion of verity configurations pruned by declared properties pinned down, I can define my notion of an oracle, as a combination of verity, and model LTS. I define the language of oracles analogously to that of verities, prove that the language of a verity is the same as that of the translated oracle, and work out a hierarchy of oracles. I present the oracle for the compressing server example of Chapter 3.
- Finally I prove two properties fundamental to auditioning based oracles. The first is a safety property such as any oracle, intensional or extensional, should have; that traces from correct implementations are not rejected. The second is the formal expression of the extra power of intensional oracles; that they can reject traces because a condition has failed to occur after a certain amount of behaviour has gone on.

6.1 Understanding Verities

To begin with we take the language of an implementation to be the set of action traces which it can exhibit whilst behaving correctly. We use an oracle to monitor the implementation and try to detect traces which expose incorrect behaviours. We can view a verity as defining a language of the traces which it accepts; then an oracle derived from the verity should define the same language, and it must be a superset of the implementation language. The usefulness of an oracle depends on how close its language comes to the implementation language.

Viewing an oracle as a machine which rejects invalid traces, we consider its language to be the set of traces which it does not reject. This permits the inclusion of infinite traces; even if we could never reject an infinite trace there is no reason why we can't say as much. We will define languages for verities, after which we can show derived oracles to be correct for the verities. We arrange the languages defined by verities for a particular formula (ϕ) into a hierachy, and demonstrate bounds for these. The first language to define is the one for a particular verity at a particular state.

In what follows, we restrict all $\rho X.\psi_X$ fixpoints to be guarded.

Definition 6.1 (Guardedness) Exactly the following formulae are guarded for X

$Y \neq X$	is guarded for X	
$[K]\psi$	is guarded for X	
$\langle K\rangle\psi$	is guarded for X	
$\psi_1 \wedge \psi_2$	is guarded for X	iff ψ_1 and ψ_2 are
$\psi_1 \vee \psi_2$	is guarded for X	iff ψ_1 and ψ_2 are
$\sigma Y.\psi, Y \neq X$	is guarded for X	iff ψ is
$\sigma X.\psi$	is guarded for X	

and a formula is *guarded* unqualified, if it is guarded for all variable names.

Once we have the shorthand tt and ff natural formulae tend to be guarded in any case, because we are always concerned with observable behaviour. And any unguarded formula can anyway be converted to a guarded one which has the same meaning; we lose no expressive power by insisting on guardedness.

We can now state exactly when a trace is a member of the language.

Definition 6.2 (Simple Traces)

$$t \triangleq \epsilon \mid a.t, a \in \text{Act}$$

Definition 6.3 (Simple Trace Language T) For a trace t from a point (s, ψ) in a verity V,

$$t \in T(V, (s, \psi)) \Leftrightarrow$$

$t = \epsilon$

or (choosing any subformula ψ' in the verity)

$\exists (s', \psi') \in V_V : (s, \psi) \rightarrow (s', \psi') \in E_V$,

$\psi \neq [K]\psi'$, $\psi \neq \langle K \rangle \psi'$ and $t \in T(V, (s', \psi'))$

or (following any transition from s in the verity)

$\exists (s', \psi') : (s, \psi) \rightarrow (s', \psi')$,

$\psi = [K]\psi'$ or $\psi = \langle K \rangle \psi', s \xrightarrow{a} s' \in M_T, t = a.t'$ and $t' \in T(V, (s', \psi'))$

Using guardedness of all ψ, Definition 6.3 is well-founded according to the measure $(|t|, \lfloor \psi \rfloor)$ which is a combination of the length of the trace and the maximum number of steps to decompose a formula to one where the leading connective is modal or the formula is a tautology.

Definition 6.4 The major component of the measure is the length of the trace:

$$|\epsilon| = 0$$
$$|a.t| = |t| + 1$$

and the minor component of the measure is the number of steps for the decomposition to the point which demands that another action from the trace be followed:

$$\begin{aligned} \lfloor [K]\psi \rfloor, \lfloor \langle K \rangle \psi \rfloor, \lfloor \mathrm{ff} \rfloor, \lfloor \mathrm{tt} \rfloor &= 0 \\ \lfloor \psi_1 \wedge \psi_2 \rfloor &= \max(\lfloor \psi_1 \rfloor, \lfloor \psi_2 \rfloor) + 1 \\ \lfloor \psi_1 \wedge \psi_2 \rfloor &= \max(\lfloor \psi_1 \rfloor, \lfloor \psi_2 \rfloor) + 1 \\ \lfloor \sigma X.\psi_X \rfloor &= \lfloor \psi_X \rfloor + 1 \\ \lfloor X \rfloor &= \lfloor \sigma X.\psi_X \rfloor + 1 \end{aligned}$$

The simple trace language does not represent the set of traces which should be accepted by an oracle. Rather it represents the traces which are *interesting* in the context of the verity.

Definition 6.5 (Interesting Traces) Simple traces are interesting just in case they induce paths through the verity from (s, ψ) to some (s', ψ').

Rather clearly then, simple traces are prefix closed. If $abcd$ is an interesting trace then abc must have been interesting to make it so.

We can relate verities by their simple trace languages. Interesting traces identify states for the derivatives of which some non-vacuous properties should hold. The size of the simple trace language varies with the verity. Larger verities have larger trace languages and can therefore be used to identify more potentially checkable states.

Definition 6.6 We make the convention that a verity V, on the formula ϕ and the model M is written $V^{\phi,M}$. We qualify the vertex and edge sets in the same way, so

$$V^{\phi,M} = \langle V_V^{\phi,M}, E_V^{\phi,M} \rangle$$

We make the natural comparison of verities by strict containment of respective vertices and edges

Definition 6.7 ($\leq$, Verities)

$$V_1 \leq V_2 \Leftrightarrow V_{V1} \subseteq V_{V2} \text{ and } E_{V1} \subseteq E_{V2}$$

Theorem 6.8 (Simple Trace Containment) Smaller verities have smaller simple trace languages

$$V_1 \leq V_2 \Rightarrow T(V_1) \subseteq T(V_2)$$

Proof We prove $t \in T(V_1) \Rightarrow t \in T(V_2)$ by induction on the length of membership inference, i.e.

$$t \in T((V_1, (s, \psi)) \text{ by inference of length } n \Rightarrow$$
$$t \in T((V_2, (s, \psi)) \text{ by inference of length } n$$

Case analysis of the inference rules gives

Rule (1) $t = \epsilon$ so $t \in T(V_2, (s, \psi))$

Rule (2) $t \in T(V_1, (s, \psi))$ so $\exists (s', \psi') \in V_{V1}, (s, \psi) \rightarrow (s', \psi') \in E_{V1}$
by $V_1 \subseteq V_2, (s', \psi') \in V_{V2}, (s, \psi) \rightarrow (s', \psi') \in E_{V2}$
and $t \in T(V_2, (s', \psi'))$ by I.H.

Rule (3) by a similar argument □

So the size of verities matters. And it's clear from the structure of verities that

Theorem 6.9 For any ϕ, M there is a largest verity

Proof Theorem 5.17 is proven constructively, and the construction contains all true (s, ψ)-pairs and all acceptable $(s, \psi) \rightarrow (s', \psi')$ □

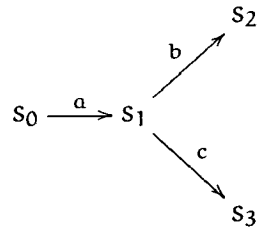
Definition 6.10 Call the verity of Theorem 5.17 $V_{\models}$. Let $T_{\models}$ be the trace language of the verity.

Corollary 6.11

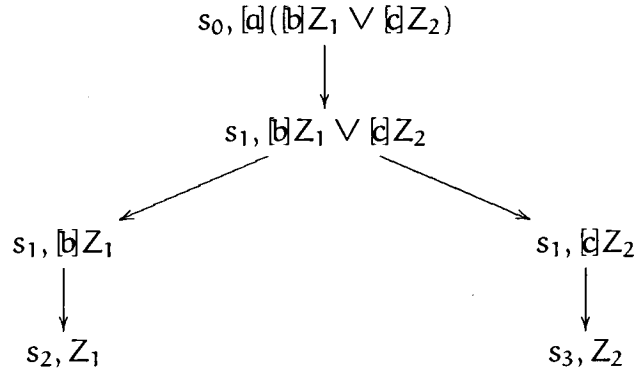
$T_{\models}$ is the largest language of a verity for ϕ, M

Proof This is immediate from Theorem 6.8 □

There is a hierarchy of verities, and by Theorem 6.8 there is also a hierarchy of trace languages. $V_{\models}$ and $T_{\models}$ are the respective maxima, but there are no minima in general. To see this, consider the model:



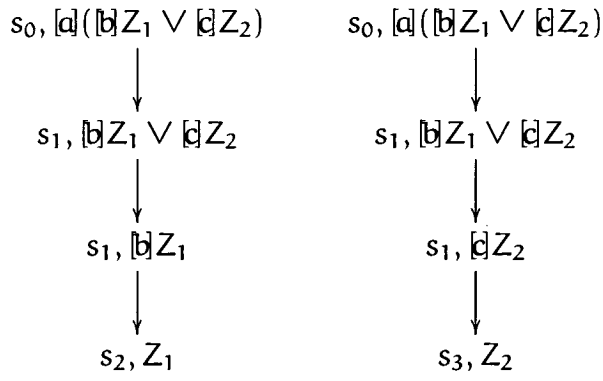
and the verity (V_0)



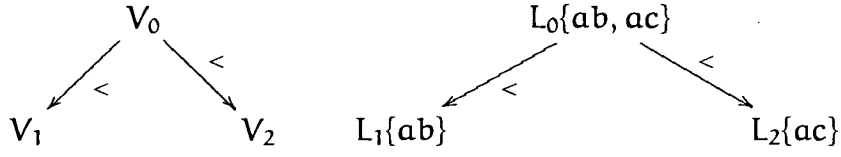
in the case where

$$s_2 \models Z_1 \text{ and } s_3 \models Z_2$$

The following verities (V_1 and V_2 respectively) come from different choices at $\vee$, are incomparable, and neither can be further reduced.



The relationships between verities and languages is consequently seen graphically as:



A trace being in a verity's trace language tells us that the state of the system after generating a t trace is subject to certain non-trivial properties. The hierarchy tells us that larger verities generate larger trace languages (Theorem 6.8), but in order to exploit this we need the traces to give us information about the implementation state.

6.2 Abstract Traces, States and Properties

The simple trace language can be viewed as completely abstract. Its actions make no reference to the states of the abstract implementation; it reveals no more about the state of the abstract implementation than can be inferred from the transitions. Making the best possible use of a verity requires explicit knowledge of the abstract implementation (AI) state being simulated; this knowledge can be reflected back at the implementation in reducing the set of transitions which are admissible. We can place such knowledge on a continuum by defining a set of possible AI states; then our knowledge is greater as the size of the set reduces towards a singleton.

Imagine that the implementation knows about the abstract implementation, and knows which state of the AI it is *simulating* at any time. Then we can make checks on implementation behaviour which depend on transitions being valid in the known current state. Where we only know that the current state is one of a set of states, we must admit as a valid transition any transition which is valid in any member state of the set of possible states.

Definition 6.12 (Explicit Traces)

$$\begin{aligned} t &\triangleq \epsilon \\ t &\triangleq a.S.t', t' \text{ an explicit trace,} \\ &\quad\; S \subseteq \mathcal{S} \text{ (a set of AI states)} \end{aligned}$$

Traces from the language which did not declare states can be defined to implicitly declare all states

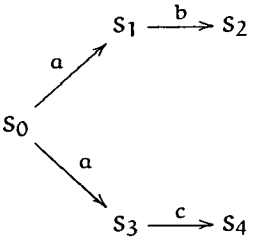
$$a.t \triangleq a.\mathcal{S}.t$$

The concept of having a greater or lesser amount of knowledge about the possible states can be expressed by a partial-order relationship on these traces

Definition 6.13 (Trace Sharpness) Trace t_1 is sharper than trace t_2 ($t_1 \leq t_2$) just in case

$$t_1 = t_2 = \epsilon, \text{ or}$$
$$t_1 = a.S_1.t_1', t_2 = a.S_2.t_2', S_1 \subseteq S_2 \text{ and } t_1' \leq t_2'$$

Consider the example



It can generate explicit traces $a.s_1.b.s_2$ and $a.s_3.c.s_4$. The definition of sharpness gives us

$$a.s_1.b.s_2 \leq a.b \text{ and } a.s_3.c.s_4 \leq a.c$$

Clearly the sharper traces (and consequently their blunter counterparts) conform to the model. But consider the traces $a.s_1.c.s_2$ and $a.s_3.b.s_4$. Sharpness gives us

$$a.s_1.c.s_2 \leq a.c \text{ and } a.s_3.b.s_4 \leq a.b$$

but the crucial difference here is that the sharper traces do not conform to the model, whereas their blunter counterparts do conform. So state declaration allows us to rule out some behaviours as inconsistent with the verity.

To attempt to add some of the information provided using state, without constraining the implementation, we choose to declare properties in traces, whence the set of admissible states can be restricted to those consistent with the properties. The importance of proof is therefore revealed as providing the connection between the declaration of property and the declaration of state; the confection of verities is exactly the expression of proofs convenient to the translation from property to state and back.

6.3 Interpolating Definitions and Traces

We seek to define an oracle which, whilst remaining satisfyingly abstract, allows as many errant behaviours as possible to be rejected. If we use traces to assert properties rather than states, then we isolate dependence on the abstract implementation in the oracle; the degree to which traces identify errors becomes a question of the explicit formulae they express, together with the richness of the verities which are used to relate states and formulae. As we have shown with the compressing server, it is possible to decide *some* non-trivial properties of the model within the implementation. One example is the decision of $\langle \mathrm{in} \rangle \mathrm{tt}$ by the presence of control in the input acceptance module of the compressing server. On the basis of these ideas we extend traces to make declarations of (sets of) properties, so they become of the form:

$$a_1\{\psi_{11}, \psi_{12}, \ldots\}a_2\{\psi_{21}, \psi_{22}, \ldots\}\ldots$$

Definition 6.14 ((Declarative) Traces)

$$t = \epsilon \mid a\,\Psi\,t \text{ (Ψ a set of formulae)}$$
$$\text{where } a \in M_{Act} \text{ and } \forall \psi \in \Psi \,.\, \psi \prec \phi$$

We now refer to declarative traces unqualifiedly as traces, since (to reveal the plot entirely) we find them a suitable basis for the rest of our work. Recall that these definitions are always in the context of a model M and a particular closed formula ϕ. But in what follows our results are all implicitly universally quantified over these variables; there is no loss of generality.

When we use traces to select configurations of verities, we want the announcements of properties to exclude some of these configurations, and so to reduce the uncertainty about which state the system is in. The easy dualisation of μM-formulae means that each formula in a trace announcement in fact announces the absence of its dual; where a configuration asserts that a property must hold of an LTS-state, the contradiction of the property means that the LTS-state cannot be the state of the system. For example, the trace $a_1\{[in]ff\}$ announces that we are not in the input acceptance module.

Definition 6.15 (Verity Configuration) We call an (s, ψ)-pair in V_V a verity state, and we call a set of (s, ψ)-pairs a verity configuration. It expresses a state of knowledge about the system as a set of conditions on LTS states.

The next stage is to define the *derivative* of a configuration under an implementation action; that is, the successor configuration.

If a configuration of a verity contains (s, ψ_1) and an announcement is made of $\neg\psi_1$, then the contradiction is very easy to check syntactically. Notice however that $\neg\psi_1$ certainly contradicts $\psi_1 \wedge \psi_2$, although this is less easy to verify syntactically. To overcome this, and to reduce the class of terms we must ultimately deal with to the prop-atomic, we define a kind of *normal form*, where an (s, ψ) in which ψ is not necessarily prop-atomic is represented by a set of successor (s, ψ') from the verity in which all ψ' are prop-atomic. Because in any case a configuration has to be a set of verity states, this does not complicate the rest of our treatment.

Definition 6.16 (Representative Set) A representative set of verity states is a set of verity states in which all formulae are prop-atomic, and which taken together declares the same information as any single verity state (in which the formula may not be prop-atomic).

$$\llcorner s, \psi \lrcorner \triangleq \{(s, \psi)\} \text{ if } \psi \text{ is prop-atomic}$$
$$\llcorner s, \psi \lrcorner \triangleq \bigcup_{(s',\psi'):(s,\psi)\to(s',\psi')} \llcorner s', \psi' \lrcorner \text{ otherwise}$$

This really just distributes the state s through the formula, as we never cross LTS transitions in $\llcorner - \lrcorner$ For example,

$$\llcorner s, ([a]\psi_1) \wedge (\langle b\rangle\psi_2 \vee [c]\psi_3)\lrcorner = \{(s, ([a]\psi_1)), (s, \langle b\rangle\psi_2), (s, [c]\psi_3)\}$$

We intend that $\llcorner - \lrcorner$ be clearly representative of all the possible formulae which are interesting for s, so we make a definition which includes all the intermediate stages to reaching the (s, ψ)'s prop-atomic subformulae which retain all modalities (don't pass state transitions).

Definition 6.17

$$\begin{aligned} \ulcorner s, \psi \urcorner &\triangleq \{(s, \psi)\} \text{ if } \psi \text{ is prop-atomic} \\ \ulcorner s, \psi \urcorner &\triangleq \{(s, \psi)\} \cup \bigcup_{(s', \psi') : (s, \psi) \to (s', \psi')} \ulcorner s', \psi' \urcorner \end{aligned}$$

and the example above becomes

$$\ulcorner s, ([a]\psi_1) \wedge (\langle b\rangle\psi_2 \vee [c]\psi_3) \urcorner = \left\{ \begin{array}{l} (s, ([a]\psi_1) \wedge (\langle b\rangle\psi_2 \vee [c]\psi_3)), \\ (s, \langle b\rangle\psi_2 \vee [c]\psi_3), \\ (s, ([a]\psi_1)), (s, \langle b\rangle\psi_2), (s, [c]\psi_3) \end{array} \right\}$$

Now we see how one configuration leads to another under first of all normalisation (the generation of a representative) set, followed by transition (the generation of successor verity states for each verity state in the configuration).

Definition 6.18 (Derivative, Δ) We use Γ to stand for a configuration, so $\Gamma = \{(s_1, \psi_1), (s_2, \psi_2), \ldots\}$ and we apply an action to a configuration with the derivative function Δ. First we have two supporting functions which interpolate configurations and trace them over transitions:

$$\begin{aligned} \Delta_{\text{inter}}(\Gamma) &\triangleq \bigcup_{(s,\psi)\in\Gamma} \ulcorner s, \psi \urcorner \\ \Delta_{\text{trans}}(a, \Gamma) &\triangleq \bigcup_{(s,\psi)\in\Gamma} \{(s', \psi') : (s, \psi) \to (s', \psi') \ , \\ &\qquad \psi = [K] \text{ or } \psi = \langle K\rangle \ , \ a \in K \ , \ s \xrightarrow{a} s' \in M_T\} \end{aligned}$$

Δ for an empty trace should just fill out the configuration by adding prop-atomic derivatives. For an action it should fill out the configuration, calculate the successor configuration and fill that out, so we put:

$$\begin{aligned} \Delta(\epsilon, \Gamma) &\triangleq \Delta_{\text{inter}}(\Gamma) \\ \Delta(a, \Gamma) &\triangleq \Delta_{\text{inter}}(\Delta_{\text{trans}}(a, \Delta_{\text{inter}}(\Gamma))) \end{aligned}$$

Then the configuration becomes data for the oracle to determine whether there is a failure:

> For each $(s, \psi) \in \Gamma$, if s is the current LTS state, then no declaration should contradict ψ.

Using this interpretation we can remove from Γ verity states involving a contradicted LTS state. Here it becomes clear why we have used $\ulcorner - \urcorner$ to fill out configurations at each stage; it gives the most flexibility to the precise form of declaration; the announcement of the contradiction of any ψ' where $(s, \psi') \in \ulcorner s, \psi \urcorner$ is enough to exclude s from the possible LTS states.

Definition 6.19 (Restriction to Consistent States) For $\Psi = \{\psi_1, \psi_2, \ldots\}$, we can excise from Γ all verity states, the LTS state of which has been ruled out. The LTS state is ruled out if it is a component of a verity state with a contradicted property. This looks a bit strange because we're really removing LTS states from the configuration, but the configuration is stored as state-and-property pairs.

$$\Gamma \downarrow \psi_i \triangleq \{(s, \psi) \in \Gamma : (s, \neg\psi_i) \notin \Gamma\}$$
$$\Gamma \downarrow \Psi \triangleq \Gamma \downarrow \psi_1 \downarrow \psi_2 \ldots$$

It is enough to only exclude on direct matching of formulae ($=$), because $\ulcorner - \urcorner$ is large enough to encompass a breakdown of any non prop-atomic ψ:

$$(s, \psi_1 \vee \psi_2) \in \Gamma \;\Rightarrow\; (s, \psi_1) \in \Gamma \text{ or } (s, \psi_2) \in \Gamma$$

Consequently, declaring $\neg\psi_2$ would be enough to exclude s as an LTS state, as

$$\{(s, \psi_1 \vee \psi_2), (s, \psi_2)\} \downarrow \neg\psi_2 = \{\}$$

which is the case where $s \models \psi_2$ is used to prove $s \models \psi_1 \vee \psi_2$. In the case where $s \models \psi_1$ is used, we don't exclude s:

$$\{(s, \psi_1 \vee \psi_2), (s, \psi_1)\} \downarrow \neg\psi_2 = \{(s, \psi_1 \vee \psi_2), (s, \psi_1)\}$$

In the first case, the verity configuration makes two assertions about a state of the LTS. One of these assertions has been contradicted, and the question is what inference should we draw from the contradiction ? The formal components of our framework have made sure that the verity configuration contains only valid assertions about LTS states. By the assumption of a valid mapping from specification to implementation, the properties must also be true of the implementation state analogue of the current LTS state. But some property is contradicted, so the current LTS state, if the mapping is indeed correct, cannot be s. Of course this is fine as long as it can be some state; when it cannot be any state we have discovered a flaw in the mapping, and auditioning has served its purpose.

6.3.1 Tracing Always

What we need to declare an outright error in the system under auditioning is that every LTS state is excluded. Such a conclusion is consistent when every

state reachable by the observed trace is represented in Γ and every verity state (or assertion) $(s, \psi) \in \Gamma$ is a valid assertion. We already know that verities contain only true assertions. We need only prove that the extension preserves this, and that it means that reachable states are always represented.

As it stands, restricting a configuration to be empty is not sufficient guarantee of error. This is because a non-interesting trace may lead to an acceptable state. We must exclude the possibility of reaching an empty configuration because the verity does not examine a particular set of states. We need to add something to the verity to do this. It doesn't hurt to assert everywhere that $\mathtt{tt}$ holds, and then wherever a state and derivative aren't reflected by a verity transition, we can add the trivial one.

Definition 6.20

$$\begin{aligned} V'_V &\triangleq V_V \cup \{(s, \mathtt{tt}) : s \in M_S\} \\ V'_E &\triangleq V_E \cup \{((s, \langle K \rangle \psi) \rightarrow (s', \mathtt{tt})) : \exists a \in M_{Act} \,,\, s \xrightarrow{a} s' \in M_T\} \\ &\quad \cup \{((s, \mathtt{tt}) \rightarrow (s', \mathtt{tt})) : \exists a \in M_{Act} \,,\, s \xrightarrow{a} s' \in M_T\} \end{aligned}$$

The next theorem ensures that the extension covers all transitions.

Theorem 6.21

$$(s, \psi) \in V'_V \,,\, s \xrightarrow{a} s' \in M_T \Rightarrow \exists \psi' : (s, \psi) \rightarrow (s', \psi') \in V'_E$$

Proof is a case-analysis on all possible formulae ψ

$[K]$	$s \xrightarrow{a} s' \in M_T \,,\, \psi = [K]\psi' \Rightarrow (s, \psi) \rightarrow (s', \psi') \in V_E$
$\langle K \rangle$	by addition of V'_E transitions
$\mathtt{tt}$	by addition of V'_E transitions
$\mathtt{ff}$	$(s, \mathtt{ff}) \notin V_V$
$\wedge, \vee, \sigma, X$	trivially

□

Theorem 6.22

$$s \xrightarrow{a} s' \in M_T \;\Rightarrow\; \forall \psi : (s, \psi) \in V_V, \exists \psi' : (s', \psi') \in \Delta(a, (s, \psi))$$

Proof By Theorem 6.21, first $\llcorner s, \psi \lrcorner \neq \{\}$ and second

$$\forall (s', \psi') \in \llcorner s, \psi \lrcorner \,.\, \exists (s^2, \psi^2) : (s', \psi') \rightarrow (s^2, \psi^2)$$

and by definition of $\ulcorner - \urcorner$, $(s^2, \psi^2) \in \ulcorner (s^2, \psi^2) \urcorner$, and if it needs saying, $\ulcorner - \urcorner$ is monotonic. □

Thus we have reached the point where with Γ we have a way of testing the validity of the auditioned implementation. It is worth reinforcing that the meaning of an empty Γ is that no possible state of the model LTS can be

consistent with the behaviour observed, and with the information supplied by the proof; the correct conclusion is that the implementation is faulty. Now we can extend the notion of Γ to traces of actions and declarations, and look at how the resultant configurations change given smaller or large verities.

The first step is simply to extend Δ to deal with the traces which an implementation, extended to property declarations, can supply.

Definition 6.23

$$\Delta(a\Psi, \Gamma) \triangleq \Delta(a, \Gamma) \downarrow \Psi$$
$$\Delta(a\Psi\, t, \Gamma) \triangleq \Delta(t, \Delta(a\, \Psi, \Gamma))$$

And the extension to the language of verities is the set of acceptable traces with declarations, that is those which never pass through an empty configuration.

Definition 6.24 (Verity Language)

$$t \in L(V, \Gamma) \Leftrightarrow$$

$$t = \epsilon\ ,\ \Gamma \neq \{\}$$
$$\text{or}\quad t = a\Psi\, t'\ ,\ t' \in L(V, \Delta(a, \Gamma) \downarrow \Psi)$$

6.4 The Oracle Transition System

A verity contains all the information necessary to define the oracle's transition system, when used in conjunction with the LTS from the model. But it is still more abstract than we would like it to be.

- Game transitions are atomic in terms of formula manipulations rather than model transitions; the two do not always correspond.
- The transitions which do correspond are not labelled in the verity, because the particular transition has no special relevance to the game, where anyway the configuration is fully known. In the *real world* of auditioning the name of the LTS transition helps us to determine the possible successor states of the verity.

With these points in mind we develop an LTS for the oracle (and we call it the OTS) where all transitions reflect model LTS transitions, and are labelled. We can do this because sets of prop-atomic formulae are sufficient to represent general formulae. The OTS stands by itself as the object with which to run the oracle in-situ.

From a verity $V = \langle V_V, E_V \rangle$ and a specification/abstract implementation $M = \langle S, A, T \rangle$ we create an oracle transition system

Definition 6.25 (OTS) An OTS O is again a graph $\langle V_O, E_O\rangle$ (vertex set V_O and edge set E_O). We write V_O and E_O to distinguish these from the vertex sets and edge sets of verities. An OTS is derived from a verity as representatives of verity states and transitions between pairs of representatives:

$$\begin{array}{lcl}(s',\psi') \in V_O & \Leftrightarrow & \exists(s,\psi) \in V_V \,.\, (s',\psi') \in \llcorner s,\psi \lrcorner \\ (s,\psi) \xrightarrow{a} (s',\psi') \in E_O & \Leftrightarrow & (s,\psi) \in V_O, (s',\psi') \in V_O\,, \\ & & \exists(s_1,\psi_1) \in V_V : \\ & & \quad (s,\psi) \xrightarrow{a} (s_1,\psi_1) \in E_V\,, \\ & & \quad (s',\psi') \in \llcorner s_1,\psi_1 \lrcorner\end{array}$$

Once the work of $\llcorner - \lrcorner$ has been applied to creating the OTS, Δ of course becomes simpler:

Definition 6.26 (Derivatives on OTSs)

$$\Delta(a, O, \Gamma) \triangleq \{(s',\psi') : \exists(s,\psi) \in \Gamma \,.\, (s,\psi) \xrightarrow{a} (s',\psi') \in E_O\}$$

Now we can apply the same definition as previously applied to verities (Definition 6.24) to define a language for oracles

Definition 6.27 (Oracle Language with Declarations)

$$t \in L(O,\Gamma) \Leftrightarrow \begin{array}{ll} & t = \epsilon \text{ and } \Gamma \neq \{\} \\ \text{or} & t = a\Psi\, t', t' \in L(O, \Delta(a, O, \Gamma) \downarrow \Psi)\end{array}$$

The transformation from verities to oracles leaves languages unchanged; when we generate an OTS from a verity and model then the respective language definitions yield the same language.

Theorem 6.28 A trace containing only prop-atomic declarations is a member of the verity language iff it is a member of the oracle language.

$$t \in L(V,\Gamma) \Leftrightarrow t \in L(O, \llcorner \Gamma \lrcorner)$$

Proof is by induction on the lengths of traces t

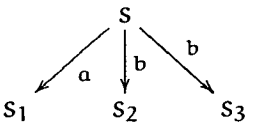
$$\begin{array}{lll}[t = \epsilon] & \Gamma = \{\} \Leftrightarrow \llcorner \Gamma \lrcorner = \{\} & \text{(obvious)} \\ [t = a\Psi\, t'] & \Delta(a, V, \Gamma) \downarrow \Psi = \llcorner \Delta(a, O, \llcorner \Gamma \lrcorner) \downarrow \Psi \lrcorner & \text{(from respective } \Delta \text{ definitions)}\end{array}$$

6.5 Example

Regarding the property

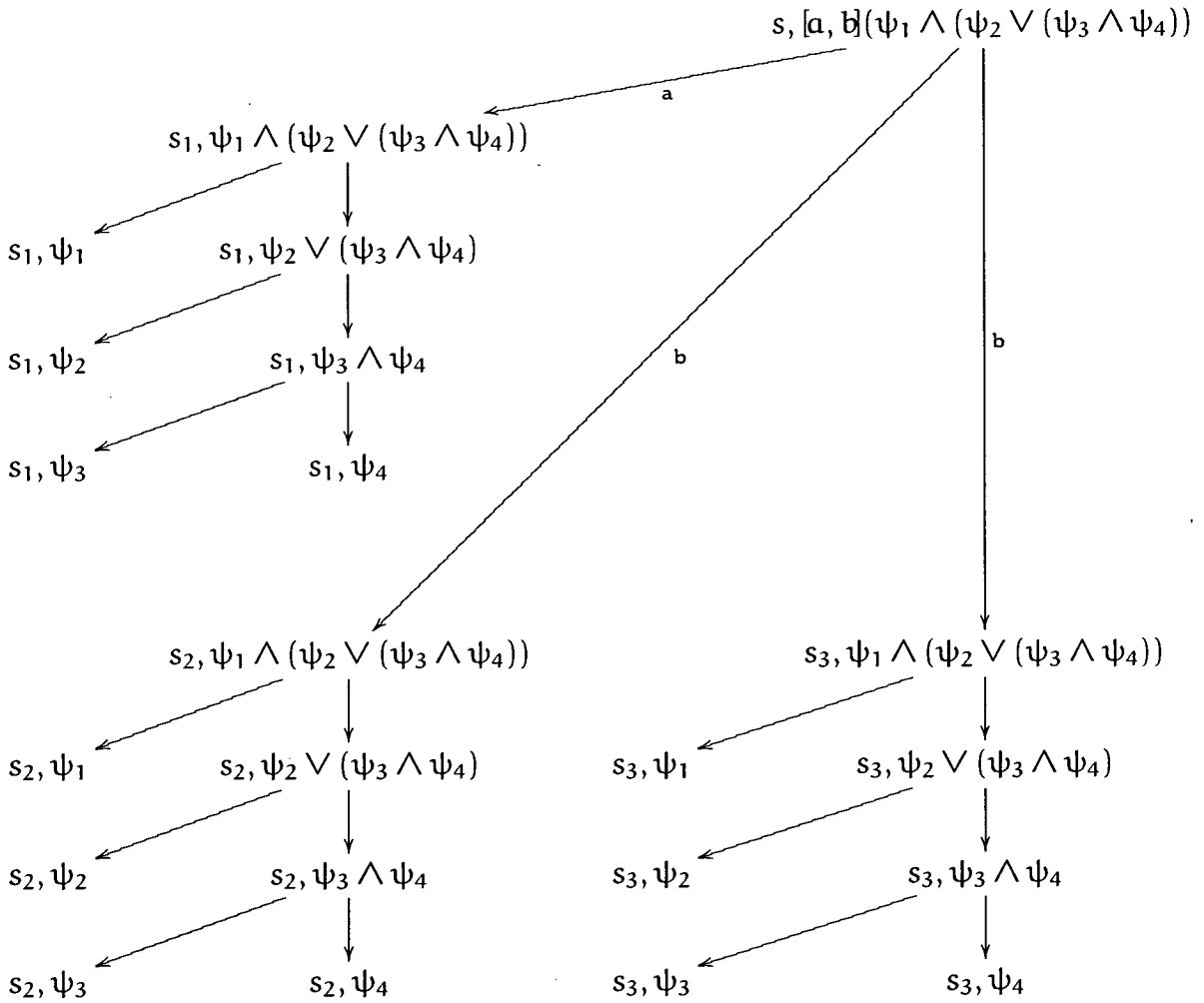
$$s \vdash \psi, \psi = [a, b](\psi_1 \wedge (\psi_2 \vee (\psi_3 \wedge \psi_4)))$$

where the interesting part of the model's transition system has the shape of



and one possible verity, modified with annotated transitions has the shape shown in Figure 6.1

Figure 6.1 A Verity



This verity requires more than any minimum set of true sub-propositions; we could also have a verity where only one choice is tested for at a disjunction; though there is no reason to do this when the source proof happens to validate both branches.

6.5.1 The Compressing Server OTS

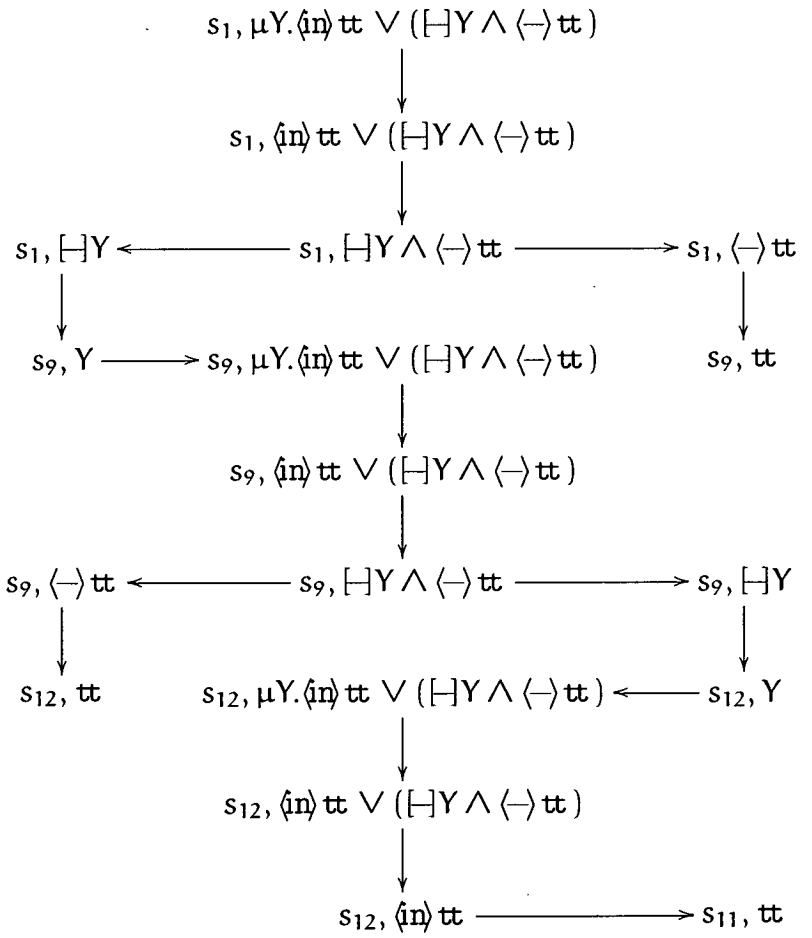
In Chapter 3 we introduced the compressing-server example in order informally to demonstrate the oracle transition systems which we have now formally de-

fined. Now we can derive the formal version of the compressing server OTS. Recall in particular that we proved, for the states when the buffers are full, that input eventually becomes possible

$$\{\ \mathrm{IO}|\mathrm{C4} \setminus \{\mathrm{put}, \mathrm{get}\}, \mathrm{I1}|\mathrm{C4} \setminus \{\mathrm{put}, \mathrm{get}\}\ \} \vdash \mu Y.\langle \mathrm{in}\rangle \mathrm{tt} \vee [\!\!-\!\!]Y \wedge \langle\!\!-\!\!\rangle \mathrm{tt}$$

A fragment of the verity resulting from the tableau is shown in Figure 6.2

Figure 6.2 Compressing Server Verity

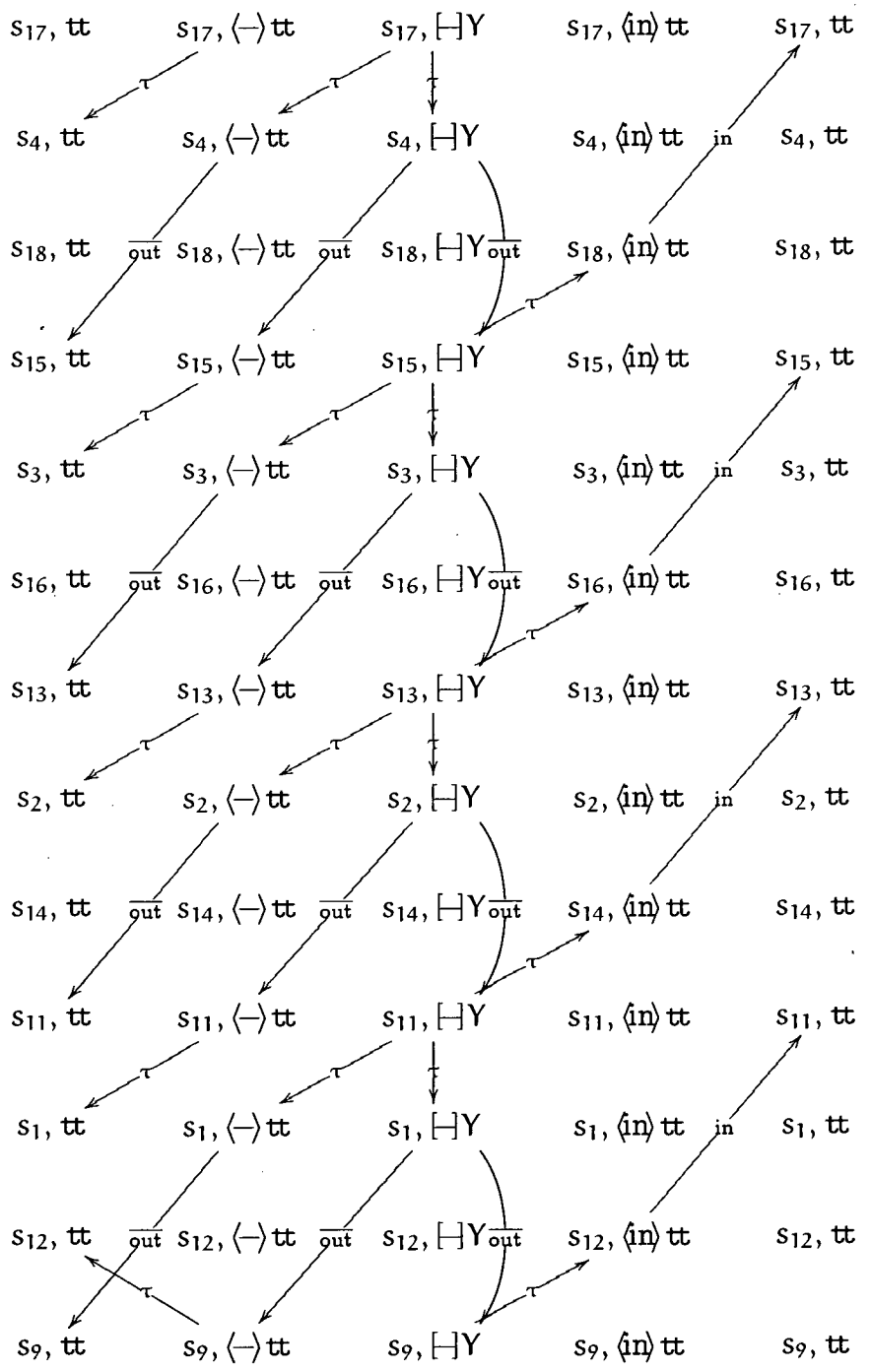


The property contains only the prop-atomic formulae

$$\langle \mathrm{in}\rangle \mathrm{tt},\ [\!\!-\!\!]Y,\ \langle\!\!-\!\!\rangle \mathrm{tt} \ \text{ and } \ \mathrm{tt}$$

so that the OTS induced by the model and verity turns out to be quite small (see Figure 6.3)

Figure 6.3 Compressing Server OTS



6.6 Hierarchies of Oracle, Model and Language

We would like to see some structure to the relationship between languages, and oracles, for the same Φ. For trace languages we have

$$V_1 \leq V_2 \Rightarrow T_1 \subseteq T_2$$

Filling out verities means that the declaration-free language for an OTS includes all possible traces of the original model. Checking with such traces is useless, and the motivation for adding declarations is to construct a system where some traces can be rejected. Ideally, stronger OTSs (larger ones) will have stronger languages (smaller ones) because the aim of a stronger OTS is to be more selective about acceptable traces; a larger OTS makes more assertions of the form

$$s \models \psi \text{ whenever } (s, \psi) \in O_V$$

First let's make the observation that our languages are still prefix closed.

Theorem 6.29

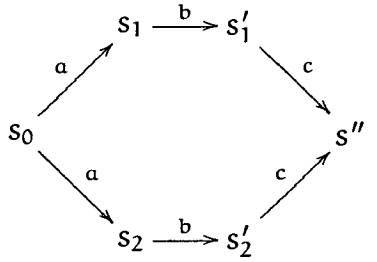
$$\text{if } t \notin L \text{ and } t \text{ a prefix of } t' \text{ then } t' \notin L$$

Proof is trivial. Once an oracle configuration is empty, no application of Δ or $\downarrow$ can generate oracle states □

We would like to be able to show that larger oracles reject more traces (thus have smaller languages). Unfortunately the hoped for

$$O_1 \leq O_2 \Rightarrow L(O_2) \subseteq L(O_1)$$

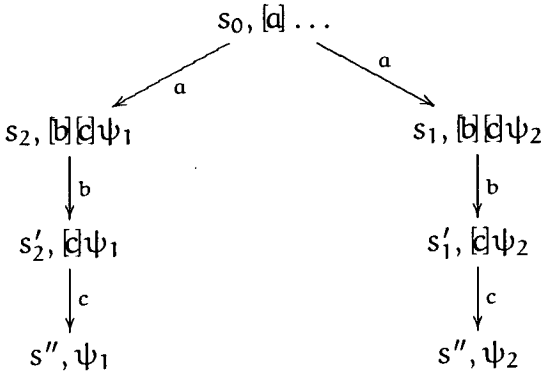
is not true. The reason is that if we exclude from a configuration all oracle states involving a particular model state, we remove the potential to exclude oracle states involving derivatives of the model state, later. Such a state of affairs should only occur if there are multiple errors in the connection between model and implementation, but this is by no means unlikely. An example makes the problem clearer; we'll show a trace t and oracles $O_1 \leq O_2$ where t is a counter-example to $L(O_2) \subseteq L(O_1)$. Consider the following LTS:



Assuming that s'' holds for the arbitrary formulae ψ_1, ψ_2 and ψ_3 then we can prove

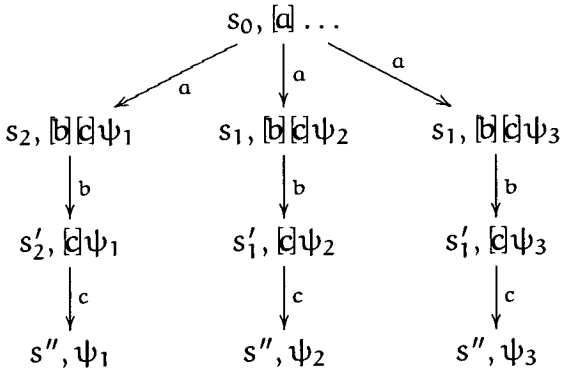
$$s_0 \models [a]([b][c]\psi_1 \vee ([b][c]\psi_2 \vee [b][c]\psi_3))$$

And testing the trace $t = a\{\}b\{\neg[c]\psi_3\}c\{\neg\psi_2\}$ on the oracle O_1:



$$\Gamma_1 \triangleq \Delta(a, O_1, \{(s_0, [a]\ldots)\})$$
$$= \{(s_2, [b][c]\psi_1), (s_1, [b][c]\psi_2)\}$$
$$\Gamma_2 \triangleq \Delta(b, O_1, \Gamma_1)$$
$$= \{(s_2', [c]\psi_1), (s_1', [c]\psi_2)\} \downarrow \neg[c]\psi_3 \quad = \{(s_2', [c]\psi_1), (s_1', [c]\psi_2)\}$$
$$\Gamma_3 \triangleq \Delta(c, O_1, \Gamma_2)$$
$$= \{(s'', \psi_1), (s'', \psi_2)\} \downarrow \neg\psi_2 = \{\}$$

'so that $t \notin L(O_1)$. But testing t for membership of the larger oracle (O_2)



$$\Gamma_1 \triangleq \Delta(a, O_2, \{(s_0, [a]\ldots)\})$$
$$= \{(s_2, [b][c]\psi_1), (s_1, [b][c]\psi_2), (s_1, [b][c]\psi_3)\}$$
$$\Gamma_2 \triangleq \Delta(b, O_2, \Gamma_1)$$
$$= \{(s_2', [c]\psi_1), (s_1', [c]\psi_2), (s_1', [c]\psi_3)\} \downarrow \neg[c]\psi_3 \quad = \{(s_2', [c]\psi_1)\}$$
$$\Gamma_3 \triangleq \Delta(c, O_2, \Gamma_2)$$
$$= \{(s'', \psi_1)\} \downarrow \neg\psi_2 \qquad = \{(s'', \psi_1)\}$$

so that $t \in L(O_2)$, contradicting $L(O_2) \subseteq L(O_1)$ □.
The exclusion of the redundant $(s_1, [b][c]\psi_3)$ path from O_2 means that s_1 is not removed from consideration by the first declaration, after which its successor conflicts with $\neg\psi_2$ and thus rules out all states.

6.6.1 A Stricter Language of Oracles

In order to restore the natural structure whereby larger oracles reject more traces, we must modify derivates. The remedy which we employ is to have the derivative notion take account of all possible paths through the oracle, even those resulting from states which we have decided that the system could not possibly be in. The only problem with doing this is a philosophical one; ruling out a state because it must satisfy a contradicted property is reasonable, but using the possibility that we are in a contradicted state in order to contradict others seems perverse. To resolve this we appeal once again to a fundamental assumption on which auditioning is based, that inconsistencies between model and implementation indicate problems with the implementation. Should the implementation contradict a state s by declaring $\neg\psi$ when the oracle has (s, ψ) then we conclude that the implementation *may* be mistaken. With the appropriately strengthened definitions of $L(O)$, we will show first of all that the function from oracles to languages is inversely monotonic. Then we show that even this strongest-so-far notion of the oracle language is nonetheless consistent with the safety of auditioning.

We will term the states which have been removed by applying $\downarrow$ to configurations *trace inconsistent.* In order to take account of trace inconsistent states we find it easiest to enrich a configuration to become a pair of a set of possible LTS states and a set of conditional requirements on states (the previous notion of verity configuration). We return to verity configurations because the encapsulation of the LTS in the new configuration remove the need to relate the (s, ψ) by transitions. It will require further thought to attempt to reconcile things into a single transition system again, but it is surely worthwhile, as we would like oracles to only maintain state information local to the truth of the property, rather than for the entire transition system.

Definition 6.30 (Oracle Configuration) is $\langle S_M \subseteq M_S, \Gamma_O \subseteq V_O \rangle$ where

- S_M is the set of LTS (model) states
- M_S is the model state set

The derivative function now accepts an LTS and OTS:

Definition 6.31 (Strict Δ) $\Delta(a, O, M, \langle S_M, \Gamma_O \rangle) \triangleq$

$$\langle \{s' : \exists s \in S_M \,.\, s \xrightarrow{a} s'\},\ \{(s', \psi') : \exists (s, \psi) \in \Gamma_O \,.\, (s, \psi) \xrightarrow{a} (s', \psi')\} \rangle$$

The restriction to uncontradicted *states* uses the OTS requirements to reduce the LTS state set. Monotonicity works because the OTS requirements are not reduced.

Definition 6.32 (Strict $\downarrow$) $\langle S_M, \Gamma_O\rangle \downarrow \Psi \triangleq$

$$\langle \{s \in S_M : \neg\exists(s,\psi_i) \in \Gamma_O \,.\, \neg\psi_i \in \Psi\}, \Gamma_O\rangle$$

From this point, the definition of L is routine. Its structure is the same as for previous versions:

Definition 6.33 (Language of Strict Oracles) $t \in L(O, M, \langle S_M, \Gamma_O\rangle) \Leftrightarrow$

$$\begin{array}{ll} [t = \epsilon] & S_M \neq \{\} \\ [t = a\Psi\, t'] & t' \in L(O, M, \Delta(a, O, M, \langle S_M, \Gamma_O\rangle) \downarrow \Psi) \end{array}$$

We can recover a language of a verity configuration by extracting the set of mentioned LTS states at the start

Definition 6.34 (Strict Verity Language)

$$L(O, M, \Gamma_O) \triangleq L(O, M, \langle \{s : \exists(s,\psi) \in \Gamma_O\}, \Gamma_O\rangle)$$

Finally we can state and prove the inverse ordering

Theorem 6.35 (Language Containment)

$$O_1 \leq O_2 \Rightarrow L(O_2, M, \langle S_M, \Gamma_{O2}\rangle) \subseteq L(O_1, M, \langle S_M, \Gamma_{O1}\rangle)$$

Proof is of a generalisation to independent sets of LTS states, by induction on traces:

$$\begin{array}{c} O_1 \leq O_2, S_{M2} \subseteq S_{M1}\ ,\ \Gamma_{O1} \subseteq \Gamma_{O2}\ ,\ t \in L(O_2, M, \langle S_{M2}, \Gamma\rangle) \\ \Rightarrow t \in L(O_1, M, \langle S_{M1}, \Gamma\rangle) \end{array}$$

By cases

$$\begin{array}{l} [t = \epsilon] \text{ trivial} \\ [t = a\Psi\, t'] \text{ as follows} \end{array}$$

Let

$$\begin{array}{l} \langle S'_{M2}, \Gamma'_{O2}\rangle \triangleq \Delta(a, O_2, \langle S_{M2}, \Gamma_{O2}\rangle) \\ \langle S'_{M1}, \Gamma'_{O1}\rangle \triangleq \Delta(a, O_1, \langle S_{M1}, \Gamma_{O1}\rangle) \end{array}$$

then

$$\begin{array}{ll} [S'_{M2} \subseteq S'_{M1}] & s' \in S'_{M2} \Rightarrow \exists s : s \xrightarrow{a} s'\ ,\ s \in S_{M2} \\ & \text{but } S_{M2} \subseteq S_{M1}, \text{ so } s \in S_{M1}, \text{ hence } s' \in S'_{M1} \\ [\Gamma'_{O1} \subseteq \Gamma'_{O2}] & (s', \psi') \in \Gamma'_{O1} \Rightarrow \exists(s,\psi) : (s,\psi) \xrightarrow{a} (s', \psi')\ ,\ (s,\psi) \in \Gamma_{O1} \\ & \text{but } \Gamma_{O1} \subseteq \Gamma_{O2}, \text{ so } (s,\psi) \in \Gamma_{O2}, \text{ hence } (s', \psi') \in \Gamma'_{O2} \end{array}$$

Now let

$$\begin{array}{l} \langle S''_{M2}, \Gamma''_{O2}\rangle \triangleq \langle S'_{M2}, \Gamma'_{O2}\rangle \downarrow \Psi \\ \langle S''_{M1}, \Gamma''_{O1}\rangle \triangleq \langle S'_{M1}, \Gamma'_{O1}\rangle \downarrow \Psi \end{array}$$

then

$$[S''_{M2} \subseteq S''_{M1}] \quad s \in S''_{M2} \Leftrightarrow s \in S'_{M2} \text{ and } \neg\exists(s,\psi) \in \Gamma'_{O2} : \neg\psi \in \Psi$$
$$\text{but } S'_{M2} \subseteq S'_{M1}, \text{ so } s \in S'_{M1}$$
$$\text{and } \Gamma'_{O1} \subseteq \Gamma'_{O2}, \text{ so } \neg\exists(s,\psi) \in \Gamma'_{O1} : \neg\psi \in \Psi$$
$$\text{hence } s \in S''_{M1}$$

so we can apply the inductive hypothesis

$$t' \in L(O_2, M, \langle S''_{M2}, \Gamma''_{O2}\rangle) \text{ so } t' \in L(O_1, M, \langle S''_{M1}, \Gamma''_{O1}\rangle)$$

And substituting $S_{M1} = S_{M2} = S_M$ and $\Gamma = \Gamma_{O1} = \Gamma_{O2}$ gives us what we want □

6.7 Safety

An oracle is only *safe* if it never rejects a trace from a correct implementation. We demand safety at a minimum for showing that the oracle generation methodology is sound. Because we have designed our oracles to enforce a form of intensional equality between the implementation and the specification, we can only show safety for models closely related to the abstract implementation from which the oracle is derived. We express safety as containment of languages; an oracle is safe for a model if the oracle language contains the model language. If this holds then the oracle will never reject a trace generated by the model. First we must define the output language of the model, in other words, ask what are the expected behaviours of a model with no associated verity or oracle ?

Definition 6.36 (Languages of Models) For the model $M = \langle S, A, T\rangle$, the language of a state of the model is the set of traces which are possible transition sequences of the model, together with property declarations consistent with properties of the model.

$$t \in L(M, s) \Leftrightarrow$$

$$\begin{aligned} & t = \epsilon \\ \text{or} \quad & t = a\Psi t' \ , \ \exists s' : s \xrightarrow{a} s' \ , \\ & \forall s' : s \xrightarrow{a} s' \in M_T, \forall \psi \in \Psi \ . \ s' \models \psi \text{ and } t' \in L(M, s') \end{aligned}$$

Then the question we ask about safety becomes a simple one of language containment:

Definition 6.37 (Safety) For the oracle O derived from a particular proof about a model M, O is safe for a model I just in the case

$$\forall s, \psi \ . \ L(I, s) \subseteq L(O, M, \{(s, \psi)\})$$

Taking the model to stand for its implementation, we can prove that any oracle is safe for the model from which it is drawn.

Theorem 6.38 (Oracles are Safe)

$$L(M, s) \subseteq L(O, M, \{(s, \psi)\})$$

Proof For any oracle O containing $(s, \psi) \in O_V$, we show that

$$t \in L(M, s) \Rightarrow t \in L(O, M, \{(s, \psi)\})$$

by induction on the length of t

$[t = \epsilon]$ direct by definitions

$[t = a\,\Psi t']$ $a\Psi\, t' \in L(M, s)$

$\exists s' : s \xrightarrow{a} s'$ thus $S'_M \neq \{\}$,

$\forall s' : s \xrightarrow{a} s', \forall \psi \in \Psi \,.\, s' \models \psi$

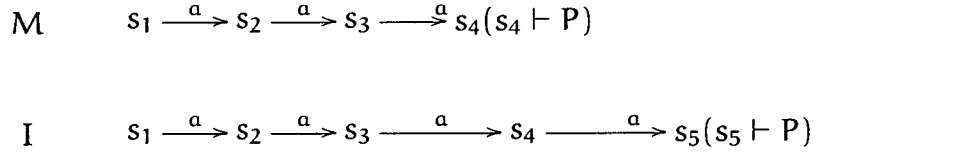
then by verity fundamentals, and Δ, $\langle S'_O, S'_M \rangle \downarrow \Psi = \langle S'_O, S'_M \rangle$

and I.H. gives $t' \in L(O, M, \langle S'_M, S'_O \rangle)$

ergo $a\Psi\, t' \in L(O, \{(s, \psi)\})$ □

Where $s \models \psi$ in a different implementation it is no longer necessarily true that $L(I, s) \subseteq L(O, M, (s, \psi))$. This is the formal consequence of intensionality, and we can demonstrate it for the simple termination example of Figure 6.4, which shows two models where the property P becomes true after a different number of transitions from the initial state.

Figure 6.4 Model and implementation differ



Consider in both models that $s_1 \models \mu X.[a]X \vee P$, but derive an oracle O from M. Using the trace $t = a\{\}a\{\}a\{\neg P\}$, for the respective languages we have

$$t \notin L(O, \{(s_1, [a]X)\}$$
$$t \notin L(M, s_1)$$
$$t \in L(I, s_1)$$

We contend that such a result is a positive benefit of our approach. The oracle in this case can identify, if it observes the trace t, that the implementation is not terminating as quickly as the original proof must have claimed it did. This suggests there is a bug in the implementation, or at least a misunderstanding of how it behaves on the part of its implementor.

6.8 Termination

One of the most important advantages of auditioning over logic-based tracing is the ability to bound termination through using intensional data. Verities should therefore allow us to check assertions that properties should eventually hold (e.g. termination of loops). For example, a correctly constructed n-place buffer satisfies

$$\nu X.\mu Y.([\dashv]Y \vee \langle in \rangle tt) \wedge [\dashv]X$$

because it cannot forever output records without coming to the point of having a free slot into which it may input. But we can only know a particular bound for a particular implementation, and unless we know a bound we cannot know it has been exceeded.

Tableau proofs of μ-properties show how certain μ properties eventually hold. Bradfield's completeness proof [Bra91] defines a notion of a signature of an (s, ψ)-pair (when $s \models \psi$) which defines how far up the approximant hierarchy we must search to see $s \models \psi$. We use the same idea to analyse how verities can be used to check correct termination, and come to a very similar proof for constructing termination-guaranteeing verities. We examine a limited but important case where something must eventually happen, and show that a verity can be trusted to detect non-termination in this case. Not all μ-fixpoints involve guaranteed termination though. Often they present only the possibility of termination, and in that case we look at what failures we can still hope to detect with what oracles.

6.8.1 Signatures in Open Formulae

In order to show termination in a verity, we must have a measure on a verity state which is well founded and which we can show reduces as the verity state evolves. This is where we need approximants (Section 4.4.3). The measure of a verity state (s, ψ) becomes the least vector defining values for approximants which makes ψ true for s, which we call its signature. A series of extensions to definitions are necessary to do this. We extend the open formulae of Definition 5.11 to add approximants. With these we can define signatures. The first step is to make approximant-indexed versions of the S_i sets to stand for the respective $\sigma X_i.\psi_i$ as the basis for an open context ρ. We only approximate the μ-fixpoints because $s \models \psi$ and we seek to approach $[\![\psi]\!]$ from below.

Definition 6.39 We write β to stand for the lexicographically ordered list of ordinals $(\alpha_1, \alpha_2, \ldots, \alpha_n)$ and $\beta \downharpoonright i$ to stand for $(\alpha_1, \alpha_2, \ldots, \alpha_i)$, $i < n$

Definition 6.40 The basis sets are

$$\begin{aligned}
S_i^{\beta} &\triangleq [\![\nu X_i.\psi_i]\!]^{M}_{V[S_1^{\beta\restriction 1}/X_1, S_2^{\beta\restriction 2}/X_2, \ldots, S_{i-1}^{\beta\restriction i-1}/X_{i-1}]} \quad (\sigma = \nu) \\
S_i^{(\alpha_1,\ldots,\alpha_{i-1},\lim)} &\triangleq \bigcup_{\alpha_i<\lim} S_i^{(\alpha_1,\ldots,\alpha_{i-1},\alpha_i)} \quad (\sigma = \mu) \\
S_i^{(\alpha_1,\ldots,\alpha_{i-1},\alpha_i+1)'} &\triangleq [\![\psi_i]\!]^{M}_{V[S_1^{\beta\restriction 1}/X_1, S_2^{\beta\restriction 2}/X_2, \ldots, S_i^{(\beta\restriction i+1)+(\alpha_i)}/X_i]} \quad (\sigma = \mu)
\end{aligned}$$

an approximant-based context becomes

$$\rho^{\beta} \triangleq \langle M, V, S_1^{\beta\restriction 1}, S_2^{\beta\restriction 2}, \ldots, S_n^{\beta\restriction n}\rangle$$

And $[\![\psi]\!]\rho^{\beta}$ follows Definition 5.11, in particular recall that

$$[\![\sigma X_i.\psi_i]\!]\rho = [\![X_i]\!]\rho = S_i$$

Definition 6.41 (Signature) is the lexicographically least β in which ψ holds in the state s:

$$|s,\psi| \triangleq \beta : s \in [\![\psi]\!]\rho^{\beta} \ , \ \neg\exists \beta' < \beta \ . \ s \in [\![\psi]\!]\rho^{\beta'}$$

and we can consider how this varies as we proceed on paths through verities. First we need to make precise the notion of following a trace in an oracle. This is analogous to the notion of a reachable configuration in a game, reading the verity as some constraints on strategies.

Definition 6.42 ($\rightsquigarrow$)

$$(s,\psi) \rightsquigarrow (s',\psi') \Leftrightarrow \exists s_1,\psi_1,s_2,\psi_2,\ldots : (s,\psi) \rightarrow (s_1,\psi_1) \rightarrow \ldots \rightarrow (s',\psi')$$

Then we refine $\rightsquigarrow$ to proceeding on a path in which every formula is part of a subformula.

Definition 6.43 ($\overset{\psi_0}{\rightsquigarrow}$)

$$(s,\psi) \overset{\psi_0}{\rightsquigarrow} (s',\psi') \Leftrightarrow$$

$$\begin{aligned}
\exists s_1,\psi_1,s_2,\psi_2,\ldots : \ & (s,\psi) \rightarrow (s_1,\psi_1) \rightarrow \ldots \rightarrow (s',\psi') \\
& \text{and } \forall \psi_i \ . \ \psi_i = \psi_0 \text{ or } \psi_i \prec \psi_0 \\
& \text{and } \psi = \psi_0 \text{ or } \psi \prec \psi_0 \\
& \text{and } \psi' = \psi_0 \text{ or } \psi' \prec \psi_0
\end{aligned}$$

We can show that for a least-fixpoint formula $\mu X_i.\psi_i$, the number of configurations of a verity on a path under $\overset{\mu X_i.\psi_i}{\rightsquigarrow}$ can be bounded. Thus the verity encodes a point at which *eventually* becomes *now*. The declaration of properties in traces allows the oracle to check that the implementation has reached fixpoint termination by the time it ought to have. For the eventuality property $\mu X.\psi \vee [\!]X$, the implementation has to continually declare $\neg\psi$, which is not

a contradiction until the oracle determines that the verity should have terminated the loop. The output from the implementation is a trace of the form $a\{\neg\psi\}b\{\neg\psi\}c\{\neg\psi\}\ldots$. If the implementation has still not reached termination when the oracle knows it should have, declaring $\neg\psi$ becomes a contradiction, and an error is detected.

Theorem 6.44 (Terminating Verity) There is a verity which for a property $\psi \prec \nu X_i.\psi_i$ of a state s, has a bounded path from the verity state (s,ψ) to a verity state beyond the fixpoint. For $s \models \psi$, $\psi \preceq \mu X_i.\psi_i$ we can construct a verity where, on all paths

$$(s,\psi) \overset{\mu X_i.\psi_i}{\rightsquigarrow} (s',\psi') : X_i \not\prec \psi'$$

and there are at most α_i configurations on the path

$$(s_j, \mu X_i.\psi_i) \text{ where } \alpha_i \text{ comes from } |s,\psi| = (\alpha_1, \alpha_2, \ldots, \alpha_i, \ldots)$$

Proof We construct a verity for (s,ψ), inductively from (s_j,ψ_j) such that

$$\begin{array}{l}
\text{if } \psi \neq \mu X_i.\psi_i \\
\quad \text{then } \forall (s_{j+1},\psi_{j+1}) : (s_j,\psi_j) \rightarrow (s_{j+1},\psi_{j+1}) \;.\; |(s_{j+1}.\psi_{j+1})| \lfloor i \leq |(s_j.\psi_j)| \lfloor i \\
\text{if } \psi = \mu X_i.\psi_i \\
\quad \text{then } \forall (s_{j+1},\psi_{j+1}) : (s_j,\psi_j) \rightarrow (s_{j+1},\psi_{j+1}) \;.\; |(s_{j+1}.\psi_{j+1})| \lfloor i < |(s_j.\psi_j)| \lfloor i
\end{array}$$

This proceeds by analysis of ψ_j

$[\psi = \psi_1 \wedge \psi_2]$	$s_j \in [\![\psi_j]\!]\rho^\beta$ so $s_j \in [\![\psi_1]\!]\rho^\beta$ and $s_j \in [\![\psi_2]\!]\rho^\beta$
	covering both possible verity paths
$[\psi = \psi_1 \vee \psi_2]$	$s_j \in [\![\psi_i]\!]\rho^\beta$ for $k = 1$ or $k = 2$
	choose to build the verity with (s_j,ψ_k)
$[[K]\psi, \langle K\rangle\psi]$	all necessary paths work for $\vdash$
	and we choose which path to build into the verity for $\langle\rightarrow\rangle$
$[X]$	$(s_j, X) \rightarrow (s_j, \sigma X.\psi_X)$ and $[\![X]\!] = [\![\sigma X.\psi_X]\!]$
$[\nu X_m.\psi_m]$	$S_m^{\alpha_1,\ldots,\alpha_m}$ is a fixpoint of ψ_X for some set of $S_p^{\alpha_1,\ldots,\alpha_m,\ldots,\alpha_p}$
	so $|(s_j,\psi_m)| = (\alpha_1,\ldots,\alpha_i,\ldots,\alpha_m,\ldots,\alpha_{p1},\alpha_{p2},\ldots)$,
	and $\lfloor i$ gives $(\alpha_1,\ldots,\alpha_i)$
$[\mu X_m.\psi_m, m > i]$	$|(s_j,\psi_m)| = (\alpha_1,\ldots,\alpha_i,\ldots,\alpha_m - 1,\ldots,\alpha_{p1},\alpha_{p2},\ldots)$,
	and $\lfloor i$ gives $(\alpha_1,\ldots,\alpha_i)$
$[\mu X_i.\psi_i]$	$|(s_j,\psi_i)| = (\alpha_1,\ldots,\alpha_i - 1,\ldots,\alpha_{p1},\alpha_{p2},\ldots))$
	and α_i must be a successor ordinal,
	so $|(s_j,\psi_i)| \lfloor i = (\alpha_1,\ldots,\alpha_i - 1)$ $\square$

So indeed there is a verity for which the appropriate μ-fixpoint's signature reduces through each transition of the verity. Thus there is a point at which the verity reaches a termination state for the fixpoint (let this state be (s', ψ'), and where the declaration of $\neg\psi'$ is contradicted; oracles derived from verities can check termination properties of implementations.

6.8.2 Termination Example

One of the commonest forms of properties of programs is **Eventually**(ψ), in one form or another. We consider the μM formula

$$\mu X.\psi_P \vee [K]X$$

The compressing-server example is of this form, and states that eventually the system will be able to perform input. But ψ_P, and implicitly tt, are the only subformulae ψ of $\mu X.\psi_P \vee [K]X$ where $X \not\prec \psi$, and Theorem 6.44 shows us that all traces must reach such an (s, ψ). Finite-branching systems have all closure ordinals less than ω [Bra91, Lar90] and we can therefore impose a finite limit in this case. It's therefore direct that we can identify a failing trace:

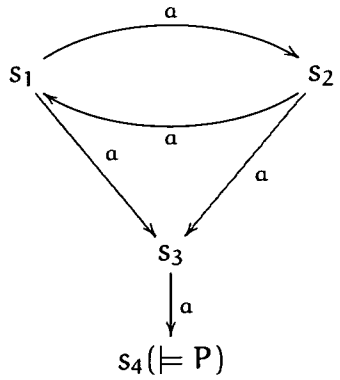
Theorem 6.45 (Finite Error Detection) For a particular model M

$$\exists \text{ an oracle } O, n \in \text{Nat} : (a \in K\{\neg\psi_P\})^n \notin L(O, M, \langle\{s\}, \{(s, \mu X.\psi_P \vee [K]X)\}\rangle)$$

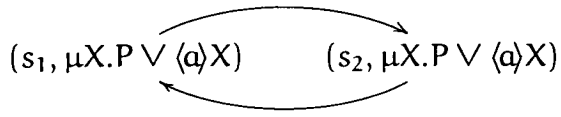
6.8.3 Using the Right Verity

A verity is only as good as the proof from which it is derived. This is positive; we only dedicate resource to checking for what is important. But it is vital to be aware that one must start off by defining and proving appropriate properties.

For instance, to check termination one needs a verity derived from a fixpoint proof. In the LTS:



then clearly $s_1 \models \mu X.P \vee \langle a\rangle X$ and $s_2 \models \mu X.P \vee \langle a\rangle X$, but the verity



is acceptable as a verity in terms of the verity rules. In this case, the declaration of $\neg\langle a \rangle X$ in the state s_4 would leave an empty oracle configuration, so that subsequent states would not be subject to analysis. However, a verity derived from a proof of

$$\{s_1, s_2\} \models \mu X.P \vee \langle a \rangle X$$

would allow such analysis.

6.9 Conclusion

This chapter concludes the formal part of the thesis.

- I have analysed verities as language acceptors, and described traces enriched with properties as the language by which annotated implementations describe their behaviour and oracles check the conformance of that behaviour.

- I have compared oracles based on the languages they accept, and shown that under some adaptations to the original definition, larger oracles are better at rejecting invalid behaviours. This allows an incremental approach to oracle-based checking; proofs of conjunctions can be built up as conjunctions of proofs, yielding oracles which check multiple properties simultaneously.

- Finally I have shown that intensional oracles really are strictly more powerful than extensional ones, by proving that they can be used to detect the violation of termination properties. This constitutes a strong formal justification of the premise outlined in Chapter 2.

Having formalised how oracles are constructed, I now want to look at the practical side of using oracles. I want to show that some real-life systems could practically be used as subjects for auditioning; and I want to show that tools can be constructed which make the practical use of auditioning much easier.

I believe though, that the industrial case studies described in the next chapter would have benefited from the use of auditioning; they have interesting termination properties which I have been able to prove within the framework of μM and tableaux.

Chapter 7

Case Studies

Summary

In the preceding chapters I have motivated and detailed how to use proofs to create oracles. One characteristic of auditioning is that the methodology is particularly suitable for industrial application because it can be incrementally introduced into existing development processes.

Attempting to apply auditioning in all its stages is outwith the scope of a single thesis, but I can at least show the work which goes into developing a set of proofs, which is the area in which the user is most likely to be stretched. And this will test the assertion that a system can reasonably be annotated.

This chapter describes the specification of two different systems, explaining the work necessary to prepare them for auditioning. The first example is a tape streaming system, part of the backup mechanism of a filesystem. For this first example I examine each stage of the process in detail. I,

- Outline the system
- Present a formal model. This model is in CCS.
- Derive some interesting properties
- Prove some properties
- Describe the annotation process

I then more briefly describe a second example, a file locking protocol, by way of demonstrating that the methodology can just as well be applied using a different specification formalism, in this case Leslie' Lamport's TLA [Lam91].

7.1 Background

Both examples described in this chapter are components of a commercial operating system, part of a development project by Digital Equipment Co. aimed at

updating the VMS Operating System [GBD96, WBW96] with a new filesystem.

The design and software were sufficiently complex that formal methods of specification and proof were thought useful for the work. As part of this, the company supported the CASE (Co-Operative Award in Science and Engineering) Studentship under which I wrote this thesis.

7.2 A Tape Streaming System

A file system uses a raw disk I/O interface to implement the abstraction of files (persistent named sequences of bytes) in what is normally nowadays a hierarchical name space. As with most such services, the abstract view of it is quite simple. But complexities are introduced in many different ways.

Distribution Many computers in a common privilege domain expect to see and access the same files. If this is to work efficiently, complicated cacheing and locking mechanisms must be designed.

Reliability An implicit but fundamentally important feature of filesystems is the persistence of data over computer shutdowns and crashes. This becomes more of a problem when it must be reconciled with performance demands.

The result is that although an abstract implementation can be simple, and a concrete implementation can obviously be seen to implement the *same* thing, it may be orders of magnitude larger because of concerns which are orthogonal to how we define the abstract behaviour. From the viewpoint of the chosen abstraction these implementation details are intensional, but at the implementor's level they are a tradeoff which respects a number of requirements which are not even expressible in the same abstract language (how do we assess the performance of a CCS process ?).

Backup is an important part of the solution of the reliability problem for filesystems. It entails making copies of the state of all or part of the filesystem at regular intervals (often nightly) and storing these copies so that in the event of complete loss of the running system, the latest backup copy can be installed in its place, and the amount of work lost can be restricted to the time since the last backup. Earlier designs of operating system called for backups to be done at quiet times, while user access to the filesystem was prohibited. But a class of systems called transaction systems has grown up which demand constant user access to data, so that backups must be created at the same time as normal work is going on. These systems also typically have huge quantities of data to be backed up, and so the backup system must be very fast.

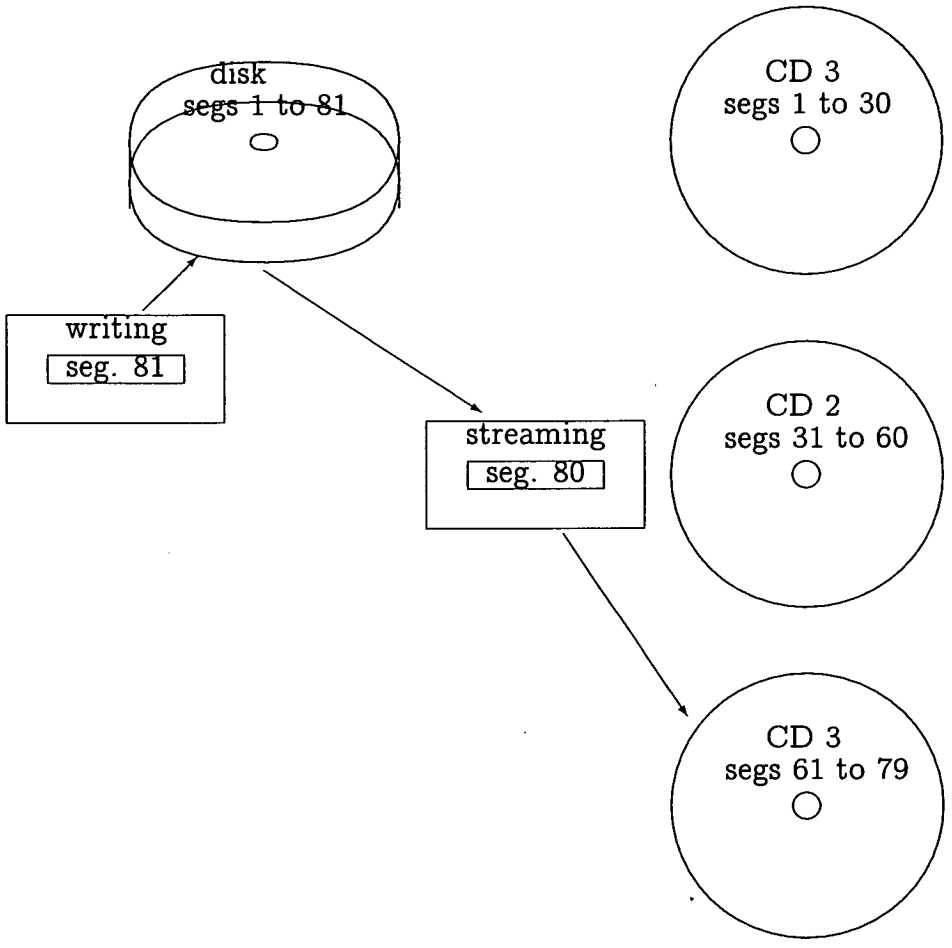
The filesystem design, based on that of [RO92], uses an abstraction called a segment. This is a large, fixed-size (perhaps 5Mb) section of disk which contains parts of many files, plus control information about its contents (what

parts of which files it contains). Segments are allocated with ever-increasing index values, and the complete collection of segments from 1 to infinity allows access to the state of the filesystem at any time in its history. The *now* state of the filesystem is kept quickly accessible by holding on disk the segments which contain non-obsolete parts of files, and a daemon process copies non-obsolete parts forward to limit the number of these segments. All segments are stored on backup media (tape, CD-Recordable,...) by the backup system, and restoration after the disk is lost is achieved by copying segments from tape back to disk.

The upshot of all of this is that the core of a high performance backup/restore system is a collection of segment transfer operations in several directions (disk-to-disk, disk-to-tape, tape-to-disk and even tape-to-tape) using memory buffers as a staging post. In normal operation segments are copied to tape immediately after they are filled on disk, so that recovery to a very recent point in time is always possible.

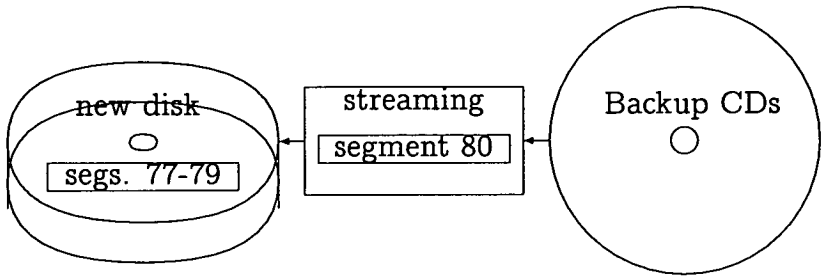
Figure 7.1 shows a disk in use with 80 segments filled and the 81st being used to store new data. Meanwhile, tape streaming has filled 2 recordable CDs with copies of the earlier segments, and is filling a 3rd with the latest segments to have been written. As permanent copies of all segments exist on CD, *time-travel* access to all data is possible.

Figure 7.1 Continuously backing up a segment-based filesystem



The segment streaming backup operation goes on in the background while normal filesystem updates proceed, but it requires minimal locking of segments and it uses efficient block-copying mechanisms so that it does not hog resources. Restoration from backup to disk after disk loss uses the same segment transfer abstractions but is the only user of the system and exploits the efficiency of segment transfers to reduce the amount of time it takes to restart the system.

Figure 7.2 Restoring the latest snapshot



The streaming system, which carries out all these transfers, was identified as a candidate for specification and auditioning.

- It must be reliable, because the correct operation of backup/restore, in terms of never corrupting data, is absolutely vital
- because this correctness must be achieved while system performance is still the maximum allowed by the physical (disk, tape, CD) systems.
- because the streaming system is sufficiently small to be specified and verified by one person in the space of a few months.

7.3 The Model

The discussion of what is important in the design of the streaming system is the first stage in deciding what abstraction we will use. But our views of other subsystems with which the streaming interacts colours the choice of abstraction too.

The major decision is to abstract away from the underlying byte-transfer mechanisms (wire/network abstractions) and assume that these behave perfectly. Then our model need not be aware of the contents of segments. On the other hand, since the streamer encapsulates our design for fast segment transfer, we choose to expose the way in which input to and output from the streamer's buffers is controlled; we model the receipt of transfer completion acknowledgements from the transfer mechanism, and how access to the newly freed buffer is mediated. We identify two symptoms of correct behaviour, specify and prove them for our model, and show they can be auditioned.

1. Segment output order reflects input order; backup expects segment number n to be in position n. Thus streaming must enforce it, and segments input in the order $[1,2,3,4,5]$ must be output in the order $[1,2,3,4,5]$ rather than $[1,3,2,5,4]$

2. Buffers must be used *properly*. A segment must be copied in completely before permission is given to copy it out. This imposes a cyclic ordering on the handshakes around any particular buffer, and we express the cyclicity as a logical property.

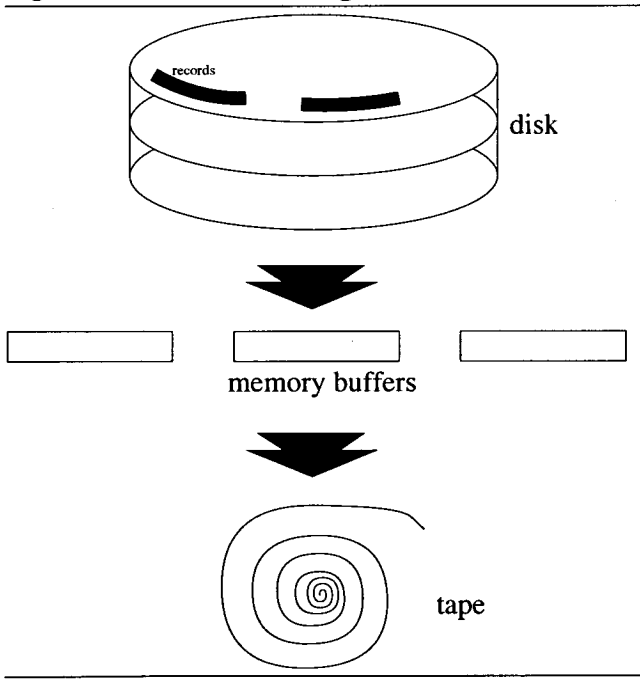
Design

We parameterise our model as one which streams r records (we will use this more neutral term instead of segments) and has c buffers at its disposal. Thus the model is parameterised by the constants r and c. Note also the shorthand (i%c) for i mod c.

Figure 7.3 represents the model. It shows input into buffers from a disk, and output from these buffers to a tape. The model abstracts the disk as the In process and the tape or CDR as the Out process. The control of buffers is represented by a pair of processes, Read and Write, which co-operate to ensure that buffer data is not trampled on. The processes interact using these actions:

sr(v)	represents *starting* to *read* a record of value v
cr(v)	represents the *completion* of a *read*
sw(v)	represents the starting of a write
cw(v)	represents the completion of a write

Figure 7.3 The streaming model



Input

The input device has the capability to put the next record (ν) in whichever buffer it's asked. It then becomes a process willing to input $\nu+1$ and willing to acknowledge completion of the input of ν. So it enforces record reading in tape order (ν) but allows the completion of outstanding reads to be acknowledged whenever is convenient to its client process. Once all records have been read, end-of-file is signalled.

We add a parameter to input which represents the index of the input record in the input order. This becomes the model's view of the record data (recall that we have abstracted away from the real data) and this index allows the simpler expression of ordering properties.

$$\begin{array}{lcll} \mathrm{In}_r(\nu) & \triangleq & \sum_{i=0}^{c-1} \overline{\mathrm{sr}_i}(\nu).(\mathrm{Ip}(i,\nu) \mid \mathrm{In}_r(\nu+1)) & \text{where } \nu < r \\ & & \mathrm{eof.nil} & \text{where } \nu = r \\ & & \mathrm{nil} & \text{where } \nu > r \\ \mathrm{Ip}(i,\nu) & \triangleq & \overline{\mathrm{cr}_i}(\nu).\mathrm{nil} & \end{array}$$

Output

The output device is simpler, in not having to deal with end-of-file. It is willing to see a write initiated from any buffer. The initiation is followed by the commencement of output ($\xrightarrow{\mathrm{out}}$) and it is these actions we hope to see occurring in strict order.

$$\begin{array}{lcl} \mathrm{Out} & \triangleq & \sum_{i=0}^{c-1} \mathrm{sw}_i(\nu).\mathrm{Out}'(i,\nu) \\ \mathrm{Out}'(i,\nu) & \triangleq & \mathrm{Out}_i(\nu).(\mathrm{Op}(i,\nu) \mid \mathrm{Out}) \\ \mathrm{Op}(i,\nu) & \triangleq & \mathrm{cw}_i(\nu).\mathrm{nil} \end{array}$$

Reading

The reading process policy is to rotate through all buffers in order, checking that the write out of the previous record using the buffer (if any) has finished, then initiating the read-in of the next record which the input has available.

$$\begin{array}{lcll} \mathrm{Read}_c(i) & \triangleq & \mathrm{sr}_{i\%c}(\nu).\mathrm{Read}_c(i+1) + \mathrm{Ending}_c(i) & (\text{where } i < c) \\ & \triangleq & \mathrm{cw}_{i\%c}(w).\mathrm{Read}'_c(i) & (\text{where } i \geq c) \\ \mathrm{Read}'_c(i) & \triangleq & (\mathrm{sr}_{i\%c}(\nu).\mathrm{Read}_c(i+1) + \mathrm{Ending}_c(i)) & \end{array}$$

Completion processing replaces Read processing when the end of input is detected. The problem is to catch exactly the writes which have been initiated, which involves a little bit of arithmetic.

$$
\begin{array}{lcll}
\mathrm{Ending}_c(i) & \triangleq & \overline{\mathrm{eof}}.\mathrm{End}_{r,c}(i+1) & \\
\mathrm{End}_{r,c}(i) & \triangleq & \mathrm{nil} & \text{where } i \geq r+c \\
 & & \mathrm{cw}_{i\%c}(w).\mathrm{End}_{r,c}(i+1) & \text{where } i \geq c \wedge i < r+c \\
 & & \tau.\mathrm{End}_{r,c}(i+1) & \text{otherwise (i.e. } i < c \wedge i < r+c)
\end{array}
$$

Writing

Writing mirrors the structure of reading. It waits for the buffer to fill, initiates its emptying and moves on to the next buffer. Note that *Write* doesn't have to do any special case processing at the beginning or the end, since we don't care about the writer process waiting around after streaming has finished.

$$
\begin{aligned}
\mathrm{Write}_c(i) &\triangleq \mathrm{cr}_{i\%c}(v).\mathrm{Write}'_c(i,v) \\
\mathrm{Write}'_c(i,v) &\triangleq \overline{\mathrm{sw}_{i\%c}}(v).\mathrm{Write}_c(i+1)
\end{aligned}
$$

Now the initial state of a streaming system with c buffers and a device with r records is

$$
\mathrm{Init}_{r,c} \triangleq (\mathrm{In}_{c,r}(v) \mid \mathrm{Out}_c \mid \mathrm{Read}_c(0) \mid \mathit{Write}_c(0))\backslash\{\mathrm{sr}_*, \mathrm{cr}_*, \mathrm{sw}_*, \mathrm{cw}_*\}
$$

The model will perform a series of internal (τ) actions and a series of $\xrightarrow{\mathrm{out}}$ actions. If we chose to do a bisimulation proof we would strongly expect

$$
\mathrm{Init}_{r,c} \approx \mathrm{Outputter}_{r,c}(0)
$$

where

$$
\begin{array}{lcll}
\mathrm{Outputter}_{r,c}(i) & \triangleq & \text{if } i < r & \text{then } \mathrm{Out}_{i\%c}(i).\mathrm{Outputter}_{r,c}(i+1) \\
 & & & \text{else nil}
\end{array}
$$

7.4 Properties

The particular properties which we find useful can be classified more generally. Indeed, it is a question deserving of further research whether a practical and general classification scheme can be devised for μM. We use the following types of property:

Ordering Ensuring, for instance, that the output of buffer i precedes that of buffer $i+1$. We use the infix $\ll$ to say "precedes".

Never Is used to prohibit bad things from happening. In particular, understanding that the algorithm cycles through buffers means that record v is transferred through buffer $v \bmod c$ so we can assert that it is never possible to output record v through any other buffer.

Exactly Once If the same record is transferred twice, something is likely to be wrong. We thus wish to say, for such actions, that they occur exactly once.

7.4.1 Ordering

For the actions $\xrightarrow{a}$ and $\xrightarrow{b}$ we express $a \ll b$ by

$$\nu X.[b]\mathrm{ff} \wedge [\vdash a]X$$

saying

> No $\xrightarrow{b}$ ever happens until an $\xrightarrow{a}$ has happened.

The ordering property we demonstrate is

Theorem 7.1 (Output Ordering)

$$\forall i \,.\, \mathrm{out}_{i\%c}(i) \ll \mathrm{out}_{i+1\%c}(i+1)$$

which expresses that record outputs are initiated in tape order. We can devise a schematic tableau for $\ll$ properties.

$$\dfrac{\dfrac{\dfrac{\dfrac{S_0 \vdash \nu X.[b]\mathrm{ff} \wedge [\vdash a]X}{S_0 \vdash U_0}(\mathrm{Def})}{S_1 \vdash U_0}(\mathrm{Thin})}{S_1 \vdash [b]\mathrm{ff} \wedge [\vdash a]U_0}(\mathrm{Un})}{\dfrac{S_1 \vdash [b]\mathrm{ff}}{S_2 \vdash \mathrm{ff}}([\vdash]) \quad \dfrac{S_1 \vdash [\vdash a]U_0}{S_3 \vdash U_0}([\vdash])}(\wedge)$$

and we are left with instantiating the set S_1 such that

1. $S_0 \subseteq S_1$ (Thin rule)
2. S_2 ($\xrightarrow{b}$-derivatives of S_1) $= \{\}$
3. S_3 ($\xrightarrow{-a}$-derivatives of S_1) $\subseteq S_1$ (ν-terminal rule)

(where S_0 is the state set {Init}) in order to construct a successful tableau and thus have a proof of the property. The next step is to find solutions for the S_i.

7.4.2 Invariant Sets

The key to proving **ordering** is to define a solution set for S_1. The tableau will keep us honest about proving that the set is closed under the necessary transitions. The way in which we have designed the model to behave influences the set we write down to try to serve for S_1.

Our first step is to define a set called Inv which encompasses the complete set of states reachable from Init. To define Inv we break down states of the model into families with the same *relative* states of concurrent processes. Thus for instance we have the states $RWO_{r,c}(i,j)$ which represent the reader available to read, the writer ready to write, the input with input available and the output ready to accept output. Further, some of the buffers have data in them. Each of the components of Inv we describe represents a different permutation of the cycles exhibited by each of the four processes.

In order to solve a particular $a \ll b$ we choose the subset of Inv for which $\xrightarrow{a}$ has not happened, and fill in the tableau by showing that this set is closed under transition ($S_3 \subseteq S_1$) and is not capable of any $\xrightarrow{b}$ actions ($S_2 = \{\}$). By separating Inv into families we can look at each in turn and define the bounds of indices for which the appropriate a-action is yet to occur, to solve $a \ll b$.

Here is our family-based representation of the invariant; in what follows we omit subscripting by the constants r and c.

$$\mathrm{Inv} \triangleq$$

$$\bigcup \left\{ \begin{array}{l} \prod_{k=max(i-c,0)}^{j-1} \mathrm{Op}(k\%c, k) \mid \mathrm{Out} \mid \mathrm{Write}(j) \mid \\ \prod_{k=j}^{i-1} \mathrm{Ip}(k\%c, k) \mid \mathrm{In}(i) \mid \mathrm{Read}(i) \\ \backslash\{sr_*, cr_*, sw_*, cw_*\} : i \leq r \wedge j \leq i \leq j + c \end{array} \right\} \tag{7.1}$$

(we'll call particular states $RWO_{r,c}(i,j)$)

$$\bigcup \left\{ \begin{array}{l} \prod_{k=i-c+1}^{j-1} \mathrm{Op}(k\%c, k) \mid \mathrm{Out} \mid \mathrm{Write}(j) \mid \\ \prod_{k=j}^{i-1} \mathrm{Ip}(k\%c, k) \mid \mathrm{In}(i) \mid \mathrm{Read}'(i) \\ \backslash\{sr_*, cr_*, sw_*, cw_*\} : c \leq i \leq r \wedge j \leq i < j + c \end{array} \right\} \tag{7.2}$$

(we'll call particular states $R'WO_{r,c}(i,j)$)

$$\bigcup \left\{ \begin{array}{l} \prod_{k=max(i-c,0)}^{j-1} \mathrm{Op}(k\%c, k) \mid \mathrm{Out} \mid \mathrm{Write}'(i,j) \mid \\ \prod_{k=j+1}^{i-1} \mathrm{Ip}(k\%c, k) \mid \mathrm{In}(i) \mid \mathrm{Read}(i) \\ \backslash\{sr_*, cr_*, sw_*, cw_*\} : i \leq r \wedge j < i \leq j + c \end{array} \right\} \tag{7.3}$$

(we'll call particular states $RW'O_{r,c}(i,j)$)

$$\bigcup \left\{ \begin{array}{l} \prod_{k=i-c+1}^{j-1} \mathrm{Op}(k\%c, k) \mid \mathrm{Out} \mid \mathrm{Write}'(i,j) \mid \\ \prod_{k=j+1}^{i-1} \mathrm{Ip}(k\%c, k) \mid \mathrm{In}(i) \mid \mathrm{Read}'(i) \\ \backslash\{sr_*, cr_*, sw_*, cw_*\} : c \leq i \leq r \wedge j < i < j + c \end{array} \right\} \tag{7.4}$$

(we'll call particular states $R'W'O_{r,c}(i,j)$)

$$\bigcup \left\{ \begin{array}{l} \prod_{k=max(i-c,0)}^{j-1} \mathrm{Op}(k\%c, k) \mid \mathrm{Out}'(j\%c; j) \mid \mathrm{Write}(j+1) \mid \\ \prod_{k=j+1}^{i-1} \mathrm{Ip}(k\%c, k) \mid \mathrm{In}(i) \mid \mathrm{Read}(i) \\ \backslash\{sr_*, cr_*, sw_*, cw_*\} : i \leq r \wedge j < i \leq j + c \end{array} \right\} \tag{7.5}$$

(we'll call particular states $RWO'_{r,c}(i,j)$)

$$\bigcup \left\{ \begin{array}{l} \prod_{k=i-c+1}^{j-1} Op(k\%c,k) \mid Out'(j\%c,j) \mid Write(j+1) \mid \\ \prod_{k=j+1}^{i-1} Ip(k\%c,k) \mid In(i) \mid Read'(i) \\ \backslash\{sr_*, cr_*, sw_*, cw_*\} : c \le i \le r \wedge j < i < j+c \end{array} \right\} \tag{7.6}$$

(we'll call particular states $R'WO'_{r,c}(i,j)$)

$$\bigcup \left\{ \begin{array}{l} \prod_{k=max(i-c,0)}^{j-1} Op(k\%c,k) \mid Out'(j\%c,j) \mid Write'(i,j+1) \mid \\ \prod_{k=j+2}^{i-1} Ip(k\%c,k) \mid In(i) \mid Read(i) \\ \backslash\{sr_*, cr_*, sw_*, cw_*\} : i \le r \wedge j+1 < i \le j+c \end{array} \right\} \tag{7.7}$$

(we'll call particular states $RW'O'_{r,c}(i,j)$)

$$\bigcup \left\{ \begin{array}{l} \prod_{k=i-c+1}^{j-1} Op(k\%c,k) \mid Out'(j\%c,j) \mid Write'(i,j+1) \mid \\ \prod_{k=j+2}^{i-1} Ip(k\%c,k) \mid In(i) \mid Read'(i) \\ \backslash\{sr_*, cr_*, sw_*, cw_*\} : c \le i \le r \wedge j+1 < i < j+c \end{array} \right\} \tag{7.8}$$

(we'll call particular states $R'W'O'_{r,c}(i,j)$)

$$\bigcup \left\{ \begin{array}{l} \prod_{k=max(i-c,0)}^{j-1} Op(k\%c,k) \mid Out \mid Write(j) \mid \\ \prod_{k=j}^{r-1} Ip(k\%c,k) \mid End(i) \\ \backslash\{sr_*, cr_*, sw_*, cw_*\} : j \le r < i \le r+c \wedge i \le j+c \end{array} \right\} \tag{7.9}$$

(we'll call particular states $EWO_{r,c}(i,j)$)

$$\bigcup \left\{ \begin{array}{l} \prod_{k=max(i-c,0)}^{j-1} Op(k\%c,k) \mid Out \mid Write'(i,j) \mid \\ \prod_{k=j+1}^{r-1} Ip(k\%c,k) \mid End(i) \\ \backslash\{sr_*, cr_*, sw_*, cw_*\} : j < r < i \le r+c \wedge i \le j+c \end{array} \right\} \tag{7.10}$$

(we'll call particular states $EW'O_{r,c}(i,j)$)

$$\bigcup \left\{ \begin{array}{l} \prod_{k=max(i-c,0)}^{j-1} Op(k\%c,k) \mid Out'(j\%c,j) \mid Write(j+1) \mid \\ \prod_{k=j+1}^{r-1} Ip(k\%c,k) \mid End(i) \\ \backslash\{sr_*, cr_*, sw_*, cw_*\} : j+1 < r < i \le r+c \wedge i \le j+c \end{array} \right\} \tag{7.11}$$

(we'll call particular states $EWO'_{r,c}(i,j)$)

$$\bigcup \left\{ \begin{array}{l} \prod_{k=i-c}^{j-1} Op(k\%c,k) \mid Out'(j\%c,j) \mid Write'(i,j+1) \mid \\ \prod_{k=j+2}^{r-1} Ip(k\%c,k) \mid End(i) \\ \backslash\{sr_*, cr_*, sw_*, cw_*\} : j < r < i \le r+c \wedge i \le j+c \end{array} \right\} \tag{7.12}$$

(we'll call particular states $EW'O'_{r,c}(i,j)$)

Now we prove that the set Inv is closed. In μM this is $Inv \vdash \nu X.[\dashv]X$.

Theorem 7.2 $Inv \vdash \nu X.[\dashv]X$

Proof

$$\dfrac{\dfrac{\dfrac{\mathsf{Inv} \vdash \nu X.\boxdot X}{\mathsf{Inv} \vdash U_0}\,(\mathrm{Def})}{\mathsf{Inv} \vdash \boxdot U_0}\,(\mathrm{Un})}{S_1 \vdash U_0}\,(\boxdot)$$

The only obligation here is to show S_1 $(\{s' : \exists s \in \mathsf{Inv} \,.\, s \rightarrow s'\}) \subseteq \mathsf{Inv}$. This requires a tedious but mechanical analysis on Inv states under any action. Here is the complete breakdown of all possible actions from states, plus the preconditions for their being enabled, based on the families defined in Inv. We drop subscripts again, saying $RW(i,d)$ for $RW_{r,c}(i,d)$ and leaving r and c implicit.

We can see that for all destinations s', $s' \in \mathsf{Inv}$.

$RWO(i,j)$

action	destination	condition	why
$\xrightarrow{\tau}$	$RWO(i+1,j)$	$(i < c \wedge i < j+c \wedge i < r)$	(sr)
	$R'WO(i,j)$	$(i \geq c \wedge i < j+c)$	(cw)
	$EWO(i+1,j)$	$(i = r \wedge i < c)$	(eof)
	$RW'O(i,j)$	$(j < i)$	(cr)

$R'WO(i,j)$

action	destination	condition	why
$\xrightarrow{\tau}$	$RWO(i+1,j)$	$(i < r)$	(sr)
	$R'W'O(i,j)$	$(j < i)$	(cr)
	$EWO(i+1,j)$	$(i = r)$	(eof)

$RW'O(i,j)$

action	destination	condition	why
$\xrightarrow{\tau}$	$R'W'O(i,j)$	$(i < j+c \wedge i \geq c)$	(cw)
	$RW'O(i+1,j)$	$(i < c \wedge i < r)$	(sr)
	$RWO'(i,j)$	(none)	(sw)
	$EW'O(i+1,j)$	$(i < c \wedge i = r)$	(eof)

$R'W'O(i,j)$

action	destination	condition	why
$\xrightarrow{\tau}$	$RW'O(i+1,j)$	$(i < r)$	(sr)
	$R'WO'(i,j)$	(none)	(sw)
	$EW'O(i+1,j)$	$(i = r)$	(eof)

$RWO'(i,j)$

action	destination	condition	why
$\xrightarrow{\mathrm{out}_{j\%c}(j)}$	$RWO(i,j+1)$	(none)	
$\xrightarrow{\tau}$	$R'WO'(i,j)$	$(i \geq c \wedge i < j+c)$	(cw)
	$RW'O'(i,j)$	$(j+1 < i)$	(cr)
	$RWO'(i+1,j)$	$(i < c \wedge i < j+c \wedge i < r)$	(sr)
	$EWO'(i+1,j)$	$(i < c \wedge i < j+c \wedge i = r)$	(eof)

	action	destination	condition	why
$R'WO'(i,j)$	$\xrightarrow{out_{j\%c}(j)}$	$R'WO(i,j+1)$	(none)	
	$\xrightarrow{\tau}$	$R'W'O'(i,j)$	$(j+1<i)$	(cr)
		$RWO'(i+1,j)$	$(i<r)$	(sr)
		$EWO'(i+1,j)$	$(i=r)$	(eof)

	action	destination	condition	why
$RW'O'(i,j)$	$\xrightarrow{out_{j\%c}(j)}$	$RW'O(i,j+1)$	(none)	
	$\xrightarrow{\tau}$	$R'W'O(i,j)$	$(i\geq c\wedge i<j+c)$	(sw)
		$RW'O'(i+1,j)$	$(i<c\wedge i<j+c\wedge i<r)$	(sr)
		$EW'O'(i+1,j)$	$(i<c\wedge i<j+c\wedge i=r)$	(eof)

	action	destination	condition	why
$R'W'O'(i,j)$	$\xrightarrow{out_{j\%c}(j)}$	$R'W'O(i,j+1)$	(none)	
	$\xrightarrow{\tau}$	$RW'O'(i+1,j)$	$(i<r)$	(sr)
		$EW'O'(i+1,j)$	$(i=r)$	(eof)

	action	destination	condition	why
$EWO(i,j)$	$\xrightarrow{\tau}$	$EWO(i+1,j)$	$(i<r+c\wedge i<j+c)$	(cw or τ)
		$EW'O(i,j)$	$(j<r)$	(cr)

Note that here $EWO(r+c,r)$ is deadlocked

	action	destination	condition	why
$EW'O(i,j)$	$\xrightarrow{\tau}$	$EW'O(i+1,j)$	$(i<r+c\wedge i<j+c)$	(cw or τ)
		$EWO'(i,j)$	(none)	(sw)

	action	destination	condition	why
$EWO'(i,j)$	$\xrightarrow{out_{j\%c}(j)}$	$EWO(i,j+1)$	(none)	
	$\xrightarrow{\tau}$	$EWO'(i+1,j)$	$(i<r+c\wedge i<j+c)$	(cw or τ)
		$EW'O'(i,j)$	$(j+1<r)$	(cr)

	action	destination	condition	why
$EW'O'(i,j)$	$\xrightarrow{out_{j\%c}(j)}$	$EW'O(i,j+1)$	(none)	
	$\xrightarrow{\tau}$	$EW'O'(i+1,j)$	$(i<r+c\wedge i<j+c)$	(cw) $\square$

We note for later reference that the only state $\in$ Inv from which an action is impossible is $EWO(r+c,r)$.

7.4.3 Proving Ordering

Now guided by the generic tableau presented earlier, we are very easily led to the set S_1 for $out_{k\%c}(k) \ll out_{k+1\%c}(k+1)$:

Definition 7.3 (Before Output)

$$\text{Before}(k) \triangleq \{S(i,j) \in \text{Inv} : j \leq k\}$$

S standing for any of the 12 components of Inv

And returning to our proof obligations for Theorem 7.1:

Proof

1. $S_0 \subseteq S_1$. But $S_0 = \text{Init}$ just contains the state $RWO(0,0)$, so $S_0 \subseteq \text{Before}(k)$.

2. S_2 ($\xrightarrow{out_{k+1\%c}(k+1)}$-derivatives of Before(k)) $= \{\}$. From the preceding exhaustive recitation of the transitions of Inv.

3. $S_3 \subseteq \text{Before}(k)$ (ν-terminal rule). Again just examine the transitions of Inv. Only the action $\xrightarrow{out_{k\%c}(k)}$ leads from within to without Before(k). S_3 is the set of Before(k) derivatives except for $\xrightarrow{out_{k\%c}(k)}$ ($[\![-out_{k\%c}(k)]\!]$), so all those are in Before(k) □.

7.4.4 Never

The modal-mu formula saying $\xrightarrow{a}$ never happens is $\nu X.[a]\,\mathrm{ff} \wedge [\![-]\!]X$, and the tableau for this is

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{S_0 \vdash \nu X.[a]\,\mathrm{ff} \wedge [\![-]\!]X}{S_0 \vdash U_0}\,(\text{Def})}{S_1 \vdash U_0}\,(\text{Thin})}{S_1 \vdash [a]\,\mathrm{ff} \wedge [\![-]\!]U_0}\,(\text{Un})}{\dfrac{S_1 \vdash [a]\,\mathrm{ff}}{S_2 \vdash \mathrm{ff}}\,([\![-]\!])\quad \dfrac{S_1 \vdash [\![-]\!]U_0}{S_3 \vdash U_0}\,([\![-]\!])}\,(\wedge)$$

Our instantiation of never is

Theorem 7.4 $\forall j,k : k \neq j\%c$. $\text{Never}(out_k(j))$

The tableau generates the obligations

1. $S_0 \subseteq S_1$
2. $S_2 \vdash \mathrm{ff}$
3. $S_3 \subseteq S_1$

Proof

1. From discussion of Inv
2. S_2 (the $\text{out}_k(j) : k \neq j\%c$ derivatives of S_1) $= \{\}$, by inspection of Inv transitions.
3. We have already seen that $\forall s \in \text{Inv}, a \in \text{Act} : s \xrightarrow{a} s' \,.\, s' \in \text{Inv}$ □

7.4.5 Exactly Once

Now we meet our only proof of a μ-formula. In order to prove that an action will happen, we must prove $\mu X.[a]X \wedge \langle - \rangle tt$, which results in a tableau:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{S_0 \vdash \mu X.[a]X \wedge \langle - \rangle tt}{S_0 \vdash U_0}\,(\text{Def})}{S_1 \vdash U_0}\,(\text{Thin})}{S_1 \vdash [a]U_0 \wedge \langle - \rangle tt}\,(\text{Un})}{\dfrac{S_1 \vdash [a]U_0}{S_2 \vdash U_0}\,([\dashv])\quad \dfrac{S_1 \vdash \langle - \rangle tt}{S_3 \vdash tt}\,(\langle \dashv \rangle)}\,(\wedge)$$

The additional complexity of the proof of a μ-formula comes in having to provide a well-founded measure m on the state set of every node which is the companion of some μ-terminal ([Bra91]). There is only one such terminal here ($S_2 \vdash U_0$), its companion is $S_1 \vdash U_0$ and we thus need

$$\forall s_1 \in S_1, s_2 \in S_2 : s_1 \xrightarrow{a} s_2 \Rightarrow m(s_1) \sqsupset m(s_2)$$

In fact we can provide such a measure for the set Inv previously introduced, under all possible actions, so that we could prove that eventually the system can do no action:

$$\text{Inv} \vdash \mu X.[\dashv]X$$

A measure for the entirety of Inv under all actions is necessarily also good for any subset of Inv under any subset of the possible actions.

The measure on states in Inv is

$$(r + c - i, r - j, \text{state-class})$$

where we assign state classes so:

1. $EW'O'$	2. EWO'	3. $EW'O$	4. EWO
5. $R'W'O'$	6. $R'WO'$	7. $R'W'O$	8. $R'WO$
9. $RW'O'$	10. RWO'	11. $RW'O$	12. RWO

The measure encapsulates a sense of progress towards having transferred all the records. Its most significant component accounts for the number of records remaining to be read into the buffers. Next is the number of records which have not yet been written out. Third is a specific ordering of state families which is consistent with progress in the other two measures; if writing is nearly done and outputting is nearly done, the major part of the count is inevitably about to drop, so that $EW'O'$ has an index of 1 to indicate that it is further *down* the measure. RWO has the highest index because a count has just dropped and other stages must be gone through before the major components of the measure drop again.

We express the fact that something occurs exactly once by saying that it occurs eventually, and thereafter never again. Generically, that $\xrightarrow{a}$ occurs exactly once is expressed as

$$\mu X.[\vdash a]X \wedge \langle\dashv\rangle tt \wedge [a](\nu Y.[a]ff \wedge [\dashv]Y)$$

and the consequent generic tableau is:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{S_0 \vdash \mu X.[\vdash a]X \wedge \langle\dashv\rangle tt \wedge [a]\nu Y.[a]ff \wedge [\dashv]Y}{S_0 \vdash U_0}(\text{Def})}{S_1 \vdash U_0}(\text{Thin})}{S_1 \vdash [\vdash a]U_0 \wedge \langle\dashv\rangle tt \wedge [a]\nu Y.[a]ff \wedge [\dashv]Y}(\text{Un})}{\dfrac{S_1 \vdash [\vdash a]U_0 \wedge \langle\dashv\rangle tt}{\dfrac{S_1 \vdash [\vdash a]U_0}{S_2 \vdash U_0}([\vdash])}\quad\dfrac{S_1 \vdash \langle\dashv\rangle tt}{S_3 \vdash tt}(\langle\dashv\rangle)}(\wedge)\qquad\dfrac{\dfrac{S_1 \vdash [a]\nu Y.[a]ff \wedge [\dashv]Y}{S_4 \vdash \nu Y.[a]ff \wedge [\dashv]Y}([\dashv])}{\dfrac{\dfrac{\dfrac{S_4 \vdash U_1}{S_5 \vdash U_1}(\text{Thin})}{S_5 \vdash [a]ff \wedge [\dashv]U_1}(\text{Un})}{\dfrac{S_5 \vdash [a]ff}{S_6 \vdash ff}([\vdash])\quad\dfrac{S_5 \vdash [\dashv]U_1}{S_7 \vdash U_1}([\dashv])}(\wedge)}(\text{Def})}$$

Theorem 7.5 Exactly Once($out_{k\%c}(k)$)

Proof The generic tableau generates S_i sets which we must instantiate. We decide to instantiate S_1 as Before(k), following our recipe based on Inv for defining the set of states preceding an action. The other state sets, except for S_5 which we will discuss, can be instantiated immediately. The rules for a correct tableau generate the following proof obligations

Closure of S_1 under $out_{k\%c}(k)$ actions. Once again Before(k) serves as our S_1 and we have already shown it closed under these actions.

Wellfoundedness $S_1 \sqsupset S_2$ from the measure m for Inv, of which Before(k) is a subset.

Sub tableau $S_1 \vdash \langle\dashv\rangle tt$ by examination of Before(k), whenever $k < r$.

Never for the sub-tableau at $S_1 \vdash [a]\nu Y.[a]\mathrm{ff} \wedge [\dashv]Y, (a = \mathrm{out}_{k\%c}(k))$. We define a set which represents all states where the action has happened

$$\mathrm{After}(k) \vdash [\![\mathrm{out}_{k\%c}(k)]\!]\,\mathrm{ff}$$

S_4 is composed only of $\xrightarrow{a}$-successors of $s \in S_1$, so $S_4 \subseteq \mathrm{After}(k)$). We set $S_5 \triangleq \mathrm{After}(k)$ and we then need only show

1 $\mathrm{After}(k) \vdash [\![\mathrm{out}_{k\%c}(k)]\!]\,\mathrm{ff}$
by examination

2 for all $s' : \exists s \in \mathrm{After}(k), a \in \mathrm{Act} \,.\, s \xrightarrow{a} s'$ then $s' \in \mathrm{After}(k)$
by examination

7.5 Annotating Streams

The design of the streaming subsystem model turned out to provide a particularly clean view of how streaming could be implemented, and resulted in the original streaming implementation being replaced by one which explicitly used the model's process and communication structure: the detailed analysis of the model meant that a system implementing it inspired more confidence than an ad-hoc implementation. This was a significant endorsement of the benefits of the modelling and proving components of the framework.

Unfortunately this success posed difficulties for the experiments in annotation which I hoped to carry out with streaming. Because the system *implemented* the model, it was highly unlikely that there would be any inconsistencies between model and system, making it impossible to positively demonstrate that auditioning this system would find inconsistencies. Indeed, the limited number of traces collected all turned out to be correct (via inspection) with respect to the model. The question still remains whether auditioning would discover inconsistencies when they exist.

But I was at least able to demonstrate that a system could be annotated and then run with annotation in place. In fact, it is common for any large piece of distributed software to have a *test harness* built for it. This is a collection of support software components designed to mimic the environment in which the system operates, to allow the central system to be easily tested in flexible ways. For example if a failure is reported in a released version of the software a programmer typically devises a series of tests to isolate the fault and runs them in a test harness rather than building a physical test system. I took advantage of the new filesystem's test harness, replaced a few components with ones designed to gather auditioning output and was almost immediately able to drive an annotated version of the streaming system with auditioning traces recorded. So the process of annotating a system in preparation for auditioning was successfully demonstrated.

7.6 An Alternative Formalism

The other area within the filesystem on which I worked was the distributed file-locking protocol. This work was less detailed than that for tape streaming, but I was able to model and verify fundamental properties of this system. And because the work was in a different formalism it provides support for the assertion that the methodology can be applied to many formalisms. The motivation for using TLA rather than CCS was that TLA was already being used by another engineer in the development group, who could assess the specification.

7.6.1 The System

The filesystem uses a distributed lock manager to mediate access to files on a server computer. A client acts on behalf of an entity which may be another process, or indirectly a user. The client must hold a write lock on a file before submitting the read and write requests that constitute an update. The aim of the locking mechanism is to ensure that views of the file at all clients are in some sense consistent. I wrote a model of the system in TLA [Lam91], formalised consistency and proved it.

As with streaming, the core of the case study is in the development of the model. TLA views a state of the system or the collection of values of variables. An atomic transition under a particular name relates the values of variables at the initiation of the transition to those at its conclusion. For this example we take TLA's view of transitions and states, but we use μM as the logic, rather than TLA's own. So for example a TLA system:

Some Variables

```
VARIABLES
  flag : Bool
  count : Nat
  thing : { "a", "b", "c" }
```

has states in Bool × Nat × {"a","b","c"}. TLA calls transitions actions, and they are defined by the logical connection of before (unprimed) variables, v, and after (primed) variables, v'. TLA presents conjunctions/ disjunctions as lists bulleted by connectives, so all elements of a conjunction, including the first, are prefixed by ∧. For example

Some Actions

```
  Step(n) ≜
    ∧ flag = true ∧ count = n
    ∧ flag′ = true ∧ count = n+1
    ∧ UNCHANGED thing
```

defines the transitions

$$(\text{true}, 0, \text{"a"}) \xrightarrow{\text{step}(0)} (\text{true}, 1, \text{"a"})$$
$$(\text{true}, 0, \text{"b"}) \xrightarrow{\text{step}(0)} (\text{true}, 1, \text{"b"})$$
$$(\text{true}, 0, \text{"c"}) \xrightarrow{\text{step}(0)} (\text{true}, 1, \text{"c"})$$
$$(\text{true}, 1, \text{"a"}) \xrightarrow{\text{step}(1)} (\text{true}, 2, \text{"a"})$$
$$\ldots$$
$$(\text{true}, 2, \text{"a"}) \xrightarrow{\text{step}(2)} (\text{true}, 3, \text{"a"})$$
$$\ldots$$
$$(\text{true}, n, \text{"a"}) \xrightarrow{\text{step}(0)} (\text{true}, n, \text{"a"})$$
$$\ldots$$

and whenever an action does not constrain a variable, it can take any value. The action specification

Lots of Actions

$$\text{ACTION}$$
$$\text{Up(n)} \triangleq \text{count} = \text{n} \land \text{count}' = \text{n+1}$$

specifies the transitions, amongst others

$$(\text{false}, n, \text{"b"}) \xrightarrow{\text{up}(n)} (\text{true}, n+1, \text{"a"})$$
$$(\text{true}, n, \text{"b"}) \xrightarrow{\text{up}(n)} (\text{false}, n+1, \text{"c"})$$
$$\ldots$$

7.6.2 Distributed File-Locking

The model of the so-called *epoch* system abstracts further from the implementation than the model of tape-streaming. It can be considered as a particular view of a large system rather than an independent small system. The very development of the model was made descending the abstraction hierarchy, and we discuss several models which can be placed against the implementation to test for our intensional implements relation. We also show that we can compare different *models* using this relationship; this yields a third application for auditioning as a refinement tester for a semi-formal implementation process.

We model only one file, and the value of that is a single number. The model is based on serialising requests according to a logical file clock. Every node maintains the age of its most recent write-lock request, which is updated from the clock on a request for a new write lock. The most abstract version of the epoch model is the *failure-free* model. In this, we have a cluster of nodes in which none of nodes ever crashes.

Cluster

```
CONSTANTS
  Nodes ∈ SUBSET Nodenames
  Writes ≜ [node : Nodes, val : Nat, age : Nat]

VARIABLES
  it ∈ [val : Nat, age : Nat]      model file stored at the server as number and generation
  writelocks ∈ SUBSET Nodes        the nodes holding a write lock
  readlocks ∈ SUBSET Nodes         the nodes holding a read lock
  reads ∈ [Nodes → Nat]            n,if there are n reads queued from that node
  writes ∈ [Writes → Nat]          n,if there are n writes from the node,
                                       of a particular value,made at a particular age
  ages ∈ [Nodes → Nat]             count time by write locks
  clock ∈ Nat                      nowness

PREDICATES
  Init ≜ no locks,no requests
    ∧ reads = [n ∈ Nodes ↦ 0] ∧ writes = [n ∈ Writes ↦ 0]
    ∧ readlocks = {} ∧ writelocks = {}
    ∧ clock ≥ it.age
```

The actions of the failure-free model are

Actions

```
ACTIONS
  WReq(n,v) ≜ queue a write on the server for later execution
    ∧ n ∈ writelocks
    ∧ writes' =
      entry indexed by (n,ages[n],v) is incremented
      all other entries in writes are unchanged
      [writes EXCEPT ![node↦n,age↦ages[n],val↦v] = @+1]
    ∧ UNCHANGED reads,readlocks,writelocks,ages,clock,it

  RReq(n) ≜ queue a read on the server for later return
    ∧ n ∈ readlocks ∨ n ∈ writelocks
    ∧ reads' = [reads EXCEPT ![node] = @+1]
    ∧ UNCHANGED writes,readlocks,writelocks,ages,clock,it

  WCom(n,v,a) ≜ commit a write previously queued
    ∧ writes[node↦n,age↦a,val↦v] > 0
    ∧ writes' = [writes EXCEPT ![node↦n,age↦a,val↦v] = @-1]
    ∧ it' = [val↦v,age↦a]
    ∧ UNCHANGED reads,readlocks,writelocks,ages,clock

  RCom(n,v,a) ≜ commit a read - generate value and age for return
    ∧ reads[n] > 0 ∧ reads' = [reads EXCEPT ![n] = @-1]
    ∧ v = it.val ∧ a = it.age
    ∧ UNCHANGED writes, readlocks,writelocks,ages,clock,it

  WLock(n) ≜ lock prior to submitting reads and writes to modify
```

```
  ∧ writelocks = {} ∧ writelocks' = {n}
  ∧ readlocks = {}
  ∧ clock' > clock ∧ ages' = [ages EXCEPT ![n] = clock]
  ∧ UNCHANGED readlocks,reads,writes,it

RLock(n) ≜ lock for reading only
  ∧ writelocks = {} ∧ readlocks' = readlocks ∪ {n}
  ∧ UNCHANGED writelocks,reads,writes,ages,clock,it

WRel(n) ≜ release a write lock
  ∧ n ∈ writelocks ∧ reads[n] = 0
  ∧ ∀ w ∈ writes : w.node = n ⇒ writes[w] = 0
  ∧ writelocks' = writelocks - {n}
  ∧ UNCHANGED readlocks,reads,writes,it,ages,clock

RRel(n) ≜ release a read lock
  ∧ n ∈ readlocks ∧ reads[n] = 0
  ∧ readlocks' = readlocks - {n}
  ∧ UNCHANGED writelocks,reads,writes,it,ages,clock
```

WReq, RReq Requests for operations should be submitted only while holding appropriate locks. The protocol is non-blocking, the requests will be queued on the server and an answer returned later.

WCom,RCom Committal of an operation is the point at which the file values are read and modified, and hence also is the serialisation point.

WLock,WRel,RLock,RRel It is for the client to obey the locking protocol, and only submit requests under the appropriate locks.

The file clock lets us express the necessary ordering properties of the distributed filesystem. As shorthand we put $\Box$ (always)

$$\Box\phi \triangleq \nu X.\phi \wedge \boxdot X$$

The properties we are interested in are

1. A write is never committed if it is older than a previous write

$$Init \vdash \Box([\![WCom(n_1, v_1, t_1)]\!]\Box[\![WCom(n_2, v_2, t_2)]\!]\,ff\,) \text{ for } t_2 < t_1$$

2. While a read lock is held by any node, all reads yield the same value. An immediate corollary is that while a particular node holds a read lock, all the reads it issues will yield the same value; we express this by saying that after a request, before the release, it's not possible to read two different

values.

$$\Box[\mathrm{RLock}(n)]\nu X.[\vdash\mathrm{RRel}(n)]X \wedge$$
$$[\mathrm{RReq}(n)]\nu Y.[\vdash\mathrm{RRel}(n)]Y \wedge$$
$$[\mathrm{RCom}(n,\nu_1)]\nu Z.[\vdash\mathrm{RRel}(n)]Z \wedge [\mathrm{RCom}(n,\nu_2)]\,\mathrm{ff}$$
$$(\text{ for } \nu_1 \neq \nu_2)$$

These properties are rather obvious for the failure-free model. But they become non-obvious or false in less abstract models, and with a verity extracted from the proofs we can hope to test the properties in these more complex models. To prove (1) we must instantiate the tableau

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{S_0 \vdash \nu X.[\vdash]X \wedge [\mathrm{WCom}(n_1,\nu_1,t_1)]\nu Y.[\vdash]Y \wedge [\mathrm{WCom}(n_2,\nu_2,t_2)]\,\mathrm{ff}}{S_0 \vdash U_0}(\mathrm{Def})}{S_1 \vdash U_0}(\mathrm{Thin})}{S_1 \vdash [\vdash]U_0 \wedge [\mathrm{WCom}(n_1,\nu_1,t_1)]\nu Y.[\vdash]Y \wedge [\mathrm{WCom}(n_2,\nu_2,t_2)]\,\mathrm{ff}}(\mathrm{Un})}{\dfrac{S_1 \vdash [\vdash]U_0}{S_2 \vdash U_0}([\vdash]) \quad \dfrac{S_1 \vdash [\mathrm{WCom}(n_1,\nu_1,t_1)]\nu Y.[\vdash]Y \wedge [\mathrm{WCom}(n_2,\nu_2,t_2)]\,\mathrm{ff}}{\dfrac{\dfrac{\dfrac{S_3 \vdash \nu Y.[\vdash]Y \wedge [\mathrm{WCom}(n_2,\nu_2,t_2)]\,\mathrm{ff}}{S_3 \vdash U_1}(\mathrm{Def})}{S_4 \vdash U_1}(\mathrm{Thin})}{S_4 \vdash [\vdash]U_1 \wedge [\mathrm{WCom}(n_2,\nu_2,t_2)]\,\mathrm{ff}}(\mathrm{Un})}{\dfrac{S_4 \vdash [\vdash]U_1}{S_5 \vdash U_1}([\vdash]) \quad \dfrac{S_4 \vdash [\mathrm{WCom}(n_2,\nu_2,t_2)]\,\mathrm{ff}}{S_6 \vdash \mathrm{ff}}([\vdash])}(\wedge)}(\wedge)$$

We work in the same way as with the streaming example, looking for a stateset to form an invariant for the top ν loop.

Invariant

```
Inv(k) ≜
  ∧ writelocks = {} ∨ ∃ n ∈ Nodes : writelocks = {n}
  ∧ readlocks = {} ∨ writelocks = {}
  ∧ ∀ w ∈ Writes : writes[w] > 0 ⇒
    ∧ w.node ∈ writelocks
    ∧ clock ≥ w.age ≥ it.age
    ∧ w.age = ages[w.node]
  ∧ ∀ n ∈ Nodes : reads[n] > 0 ⇒ n ∈ readlocks
  ∧ ∀ n ∈ writelocks : clock ≥ ages[n] ≥ it.age
  ∧ clock ≥ it.age ≥ k
```

Theorem 7.6 (Write Ordering)

$$\mathrm{Init} \vdash \Box([\mathrm{WCom}(n_1,\nu_1,t_1)]\Box[\mathrm{WCom}(n_2,\nu_2,t_2)]\,\mathrm{ff}) \text{ for } t_2 < t_1$$

Proof Instantiating $S_1 \triangleq \mathrm{Inv}(0)$ and $S_4 \triangleq \mathrm{Inv}(t_1)$, the tableau determines the

proof obligations...

$\left[S_0 \triangleq \text{Init} \subseteq S_1\right]$ (S_1 our first invariant set)

containment is trivial

$[S_2 \subseteq S_1]$ ($\xrightarrow{-}$-derivatives of S_1)

by case analysis of the actions in Inv(k)

[**RReq**(n)] for the n read, precondition gives $n \in$ readlocks

[**WReq**(n, v, t)] for writes[node $\mapsto n$, val $\mapsto v$, age $\mapsto t$] > 0,

as a result of the request, pre-condition demands

$n \in$ writelocks $\wedge\ t =$ ages[w.node]

[**RCom**(n, v)] trivial

[**WCom**(n, v, t)] it.age $\leftarrow$ ages[w.node] $= w$.age so it.age increases.

But as all writes were of the same age, they're still $\geq$

(actually just $=$) to it.age

[**RLock**(n)] trivial

[**WLock**(n)] clock$' \geq$ it.age and clock$' \geq$ ages[n]

[**RRel**(n)] trivial

[**WRel**(n)] trivial

$[S_3 \subseteq S_4]$ $\overset{\text{WCom}(n_1,v_1,t_1)}{\longrightarrow}$-derivatives of S_1 are contained

in the nominated invariant set

$\overset{\text{WCom}(n_1,v_1,t_1)}{\longrightarrow}$ – derivs of Inv(0) $\subseteq$ Inv(t_1). Any w has age $= t_1$

so it$'$.age $= t_1$,and the rest is trivial

$[S_5 \subseteq S_4]$ ($\xrightarrow{-}$-derivatives of S_4)

by closure under $\xrightarrow{-}$ for any Inv(k)

$[S_6 = \{\}]$ ($\overset{\text{WCom}(n_2,v_2,t_2)}{\longrightarrow}$-derivatives of S_4)

any possible w has age $\geq t_1$ □

Theorem 7.7 (Read Consistency)

$$\Box[\text{RLock}(n)]\nu X.[\text{-RRel}(n)]X \wedge$$
$$[\text{RReq}(n)]\nu Y.[\text{-RRel}(n)]Y \wedge$$
$$[\text{RCom}(n, v_1)]\nu Z.[\text{-RRel}(n)]Z \wedge [\text{RCom}(n, v_2)]\,\text{ff}$$
$$(\text{ for } v_1 \neq v_2)$$

Proof uses the tableau (with W standing for the □ fixpoint):

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{S_0 \vdash \nu W.\phi}{S_0 \vdash U_0}\,(\text{Def})}{S_1 \vdash U_0}\,(\text{Thin})}{S_1 \vdash \phi \wedge \psi}\,(\text{Un})}{\dfrac{S_1 \vdash \boxdot\phi}{S_2 \vdash U_0}\,(\boxdot)} \qquad \dfrac{S_1 \vdash [\![\mathrm{RLock}(n)]\!]\phi}{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{S_3 \vdash \nu X.\phi}{S_3 \vdash U_1}\,(\text{Def})}{S_4 \vdash U_1}\,(\text{Thin})}{S_4 \vdash \boxdot\mathrm{RRel}(n)]\!]\phi}\,(\text{Un})}{S_5 \vdash \phi \wedge \psi}\,(\boxdot)}{S_5 \vdash U_1}\,(\wedge) \qquad \dfrac{S_5 \vdash [\![\mathrm{RReq}(n)]\!]\phi}{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{S_6 \vdash \nu Y.\phi}{S_6 \vdash U_2}\,(\text{Def})}{S_7 \vdash U_2}\,(\text{Thin})}{S_7 \vdash \boxdot\mathrm{RRel}(n)]\!]\phi}\,(\text{Un})}{S_8 \vdash \phi \wedge \psi}\,(\boxdot)}{S_8 \vdash U_2}\,(\wedge) \qquad \dfrac{S_8 \vdash [\![\mathrm{RCom}(n, v_1)]\!]\phi}{\dfrac{\dfrac{\dfrac{\dfrac{S_9 \vdash \nu Z.\phi}{S_9 \vdash U_3}\,(\text{Def})}{S_{10} \vdash U_3}\,(\text{Thin})}{S_{10} \vdash \phi \wedge \psi}\,(\text{Un})}{\dfrac{S_{10} \vdash \boxdot\mathrm{RRel}(n)]\!]\phi}{S_{11} \vdash U_3}\,(\boxdot) \qquad \dfrac{S_{10} \vdash [\![\mathrm{RCom}(n, v_2)]\!]\phi}{S_{12} \vdash \mathrm{ff}}\,(\boxdot)}\,(\wedge)}\,(\boxdot)}\,(\wedge)$$

The instantiation is simpler than the size might suggest; the initial set is given, and the invariant we gave earlier, which is just an expression of any state where the locking rules hold, serves here (if it didn't, our locking algorithm would be broken). The fixpoint sets to solve S_4 and S_7 narrow down the invariant set to those where n holds a read lock, which makes sense, and the final fixpoint is further narrowed to the states where the value held is that returned by the first read. Formally, the solution sets are

$$\begin{aligned}
S_0 &\triangleq \mathrm{Init} \\
S_1 &\triangleq \mathrm{Inv}(0) \\
S_4 &\triangleq S_7 \triangleq \mathrm{Inv}(0) \wedge n \in \mathrm{readlocks} \\
S_{10} &\triangleq \mathrm{Inv}(0) \wedge n \in \mathrm{readlocks} \wedge \mathrm{it.val} = v_1
\end{aligned}$$

It is routine if tedious to check this, but the observation to make is that the state sets defined as solution sets are *obvious*, in retrospect. They just encode the characteristics of all system states which we think should satisfy the property, and then we check them. In this sense TLA is a very intuitive system to work with.

7.6.3 Models Tolerating Failure

Distributed file locking is intended to tolerate the failure of nodes. But if we add the appropriate action simulating failure to the present model:

Crashing Nodes

```
ACTIONS
  Crash(n) ≜ and reboot instantly
    ∧ readlocks' = readlocks - {n} ∧ writelocks' = readlocks - {n}
    ∧ ∀ n2 ∈ Nodes : n ≠ n2 ⇒ UNCHANGED ages[n2]
    ∧ UNCHANGED reads,writes,it,clock
```

then Theorem 7.6 no longer holds. Because the requests submitted by a client are preserved while its locks are lost, other clients can acquire locks to sneak requests in before the orphaned requests; the upshot is that write commits are no longer time ordered.

The aim of refinement testing using auditioning should be to detect failures resulting from such *faulty* refinement steps as admitting crashes without the appropriate supporting corrections. Once the mistake is identified the technical solution is to extend the implementation with an *epoch* counter. The idea is that whenever a reconfiguration occurs (a node crashing is a reconfiguration) the system generates a new, higher value for the epoch. The server rejects requests which have an out of date view of the epoch. This is enough to reinstate Theorem 7.6, and it can be proven using the same tableau and with an slightly modified invariant. The complete epoch model and its invariant are given in Appendix A.

Theorem 7.8 (Write Ordering)

$$\mathrm{Init}_{\mathrm{failure-tolerant}} \vdash \Box([\mathrm{WCom}(n_1, v_1, t_1]\!]\Box[\mathrm{WCom}(n_2, v_2, t_2]\!]\,\mathrm{ff}) \text{ for } t_2 < t_1$$

Proof $S_1 \triangleq \mathrm{FTInv}(0), S_4 \triangleq \mathrm{FTInv}(t_i)$, and by checking of containments and calculation of successors □

A crash proof version of Theorem 7.7 can be approached by using FTInv in the same way as the original uses Inv.

The failure-tolerant specification is significantly bigger than the failure-free one. This leads us to an interesting test for auditioning. If we audition an implementation known to fail (by crashing) against the failure-free model, we should expect an oracle to observe failures when the implementation is made to crash. In particular, we would expect to see write-ordering failures.

7.7 Summary

In this chapter I have described two practical examples of the application of auditioning. In both the systems which I specified there was a significant advance

in the understanding of the systems, to the extent in one case that the system was re-implemented based on the structure defined in the model which I wrote.

I have shown that although writing large proofs about non-trivial systems is not itself easy, it can be done when the developer understands the system being studied and applies that knowledge to the construction of invariants for proofs. It seems natural that this knowledge should be central to the checking process, as our intensional mechanisms make it.

Although I have concentrated on the first (specification and proof) phase of auditioning, I have made some initial and positive experiments in auditing systems. In order to make it easier to proceed to this phase, and also to simplify proof development, I next consider what mechanical help would be useful for an auditioning system.

Chapter 8

Mechanisation

8.1 Summary

Chapter 7 suggests that the hardest part of developing a convincing proof can be the exhaustive analysis of a complicated term for a set of states. The invention of the term is unavoidably the work of the designer, and because she knows the system she can bring domain knowledge to bear. Decideability results showing that automatic derivation of terms is impossible support this approach.

By contrast, checking involved terms is best left to an automatic system. Where I defined a large invariant set and verified various containments under actions, the work would have been much easier with mechanical support.

In this chapter I look at where mechanical support for the auditioning process would be helpful, and at how it can be provided.

- I use canonical tableaux with uninstantiated state sets as a structure in which to look for proofs.
- I place labelled transition systems at the centre of a system for interactive development and checking of proofs.
- I examine the relationship between the solution sets of states at tableau nodes, and develop a notation for identifying the state sets which the user must instantiate, and those which an automatic system can infer.
- I describe a language for denoting families of CCS terms and look at how the automatic generalizing of these terms gives some help to users in identifying state sets.
- I describe an implementation of the family language.
- I conclude with some general remarks summarising what I have discovered about mechanisation.

8.2 What to Mechanise

As I have suggested, some parts of the auditioning methodology are more amenable to mechanisation than others. We need to consider whether mechanical support is formally tractable, and also whether it is pragmatically useful. The areas to consider in auditioning are

- Specification/model writing
- Proof development
- Oracle generation
- Implementation annotation

First of all, generating a checker should be a mechanical process. Given a proof in tableau form, the transformation to an oracle (Chapter 6) and from an oracle to a program is a large but routine processing task, loosely analogous to transforming assembly language into machine code. Further, because ultimately we need to provide input to an oracle generator in some formal language, we may as well think about whether we should have machine support to develop the input in the first place; that will save us some typing.

The first step towards a more explicit, mechanically influenced methodology is to analyse the approach we have taken in applying tableau systems in case studies. We want to develop a partly mechanised methodology for proof development by taking on the tasks which were most onerous in the manual developments such as those illustrated in Chapter 7.

In Chapter 7 we have made much use of canonical tableau where the state sets have not been instantiated. We will make this formal. Because of the tree structure, when we generate indices for state sets they must be multi-dimensional. Each state set variable is termed S_i, where $i = n.m.p\ldots$. We define a little bit of shorthand for deriving unique indices from parent state set indices.

Definition 8.1 (State set indexes)

$$n^+ \triangleq n + 1$$
$$n.m^+ \triangleq n.m + 1$$
$$n.m.p^+ \triangleq n.m.p + 1\ldots$$

We translate from a μM formula to an outline canonical tableau. It is an outline because it has a number of uninstantiated variables which stand for sets of states.

Definition 8.2 (Outline Canonical Tableaux)

$$T(\phi) \triangleq T(S_0 \vdash_\Delta \phi)$$

$$T(S_i \vdash_\Delta tt) \triangleq \quad S_i \vdash_\Delta tt$$

$$T(S_i \vdash_\Delta ff) \triangleq \quad S_i \vdash_\Delta ff$$

$$T(S_i \vdash_\Delta Z) \triangleq \quad S_i \vdash_\Delta Z$$

$$T(S_i \vdash_\Delta \phi_1 \wedge \phi_2) \triangleq \quad \frac{S_i \vdash_\Delta \phi_1 \wedge \phi_2}{T(S_i \vdash_\Delta \phi_1)T(S_i \vdash_\Delta \phi_2)}$$

$$T(S_i \vdash_\Delta \phi_1 \vee \phi_2) \triangleq \quad \frac{S_i \vdash_\Delta \phi_1 \vee \phi_2}{T(S_{i.0} \vdash_\Delta \phi_1)T(S_{i.1} \vdash_\Delta \phi_2)}\ S_i = S_{i.0} \cup S_{i.1}$$

$$T(S_i \vdash_\Delta [K]\phi) \triangleq \quad \frac{S_i \vdash_\Delta [K]\phi}{T(S_j \vdash_\Delta \phi)}\ j = i^+, S_j = \{s' : \exists s \in S_i, a \in K\ .\ s \xrightarrow{a} s'\}$$

$$T(S_i \vdash_\Delta \langle K\rangle\phi) \triangleq \quad \frac{S_i \vdash_\Delta \langle K\rangle\phi}{T(S_j \vdash_\Delta \phi)}\ j = i^+, \forall s \in S_i\ .\ \exists a \in K\ ,\ s' \in S_j\ ,\ s \xrightarrow{a} s'$$

$$T(S_i \vdash_\Delta \sigma X.\phi) \triangleq \quad \frac{S_i \vdash_\Delta \sigma X.\phi}{S_j \vdash_{\Delta[(U=\sigma X.\phi)]} \phi[U/X]}\ j = i^+, S_i \subseteq S_j$$

$$T(S_i \vdash_\Delta U) \triangleq \quad S_i \vdash_\Delta U$$

The σ rule combines the *Thin* and *Un* rules of tableaux, which always occur together in canonical tableax.

Theorem 8.3 The outline canonical tableau is instantiable for ϕ (there are values for each S_i which make the tableau true) just in the case $S \models \phi$

Proof By Bradfield's definition of canonical tableaux [Bra91]. The only difference is that we have restricted thinning to the unfolding points, but monotonicity ensures this is sufficient □

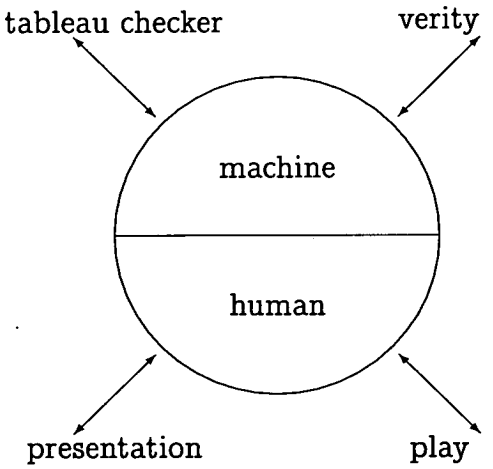
So the simple expression of *the* outline canonical tableau in effect does nothing except to present the formula differently. In practice, however, its expression serves to identify the crucial points of real-world proofs, which amounts (our case studies of Chapter 7 provide copious evidence) to the definition of a fixpoint set. In a sense we go further along the road from computer proof than Bradfield, and reject not only undecidable or high complexity tableau generation by machine, but any choice at all as to the shape of the tableau. We view the tableau as a way of defining the structure and requirements for proving. By using the tableau system just to define relationships between state sets we can work in a more modular way. Once we have the canonical structure we can concentrate on the transition system. The separation is aesthetically pleasing because it increases modularity and helpful when we come to deal with oracles, where we have already taken the view that they are separate; the input to the oracle generation process must provide a transition system along with a tableau proof which uses that transition system.

Consequently, our approach to mechanisation will be to place LTSs at the core, and branch out by considering LTS interactions with proofs at one end, and LTS implementations (particularly CCS) at the other.

8.3 Labelled Transition Systems

Contrast the human and mechanical implementation and analysis of an LTS. The former requires a presentational bias to the representation, and the latter demands an internally efficient and manipulable bias. A user's job is to *play* with an LTS in order to instantiate a proof, while the machine's job is to *calculate* functions of statesets to verify correctness or translate to a verity. This suggests the view of an LTS of Figure 8.1

Figure 8.1 Interfaces to an LTS Implementation



What must be the contents of each of these interfaces ? Oracle generation is the easiest to define, because the translation from proof is fixed and automatic. Of course the *representational* question is still open, in that for any large or infinite LTS a large or infinite oracle will be required. But this can be achieved by building on the basis of a successful solution to the representation problem for LTSs, and proofs, and we attack these next.

8.3.1 Checker

A tableau is formed of a series of rules:

$$\frac{S_0 \vdash C(\phi_1, \phi_2, \ldots)}{S_1 \vdash \phi_1 \quad S_2 \vdash \phi_2 \quad \ldots}$$

The states S_0 and S_1 mostly bear simple relationships to each other. For instance, in the step

$$\frac{S_0 \vdash \phi_1 \wedge \phi_2}{S_1 \vdash \phi_1 \quad S_2 \vdash \phi_2}$$

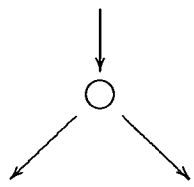
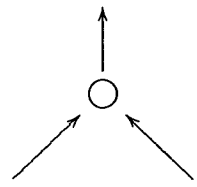
if the goal is $S_0 \subseteq [\![\phi_1 \wedge \phi_2]\!]$ then the minimum tableau-valid goal stateset for S_1 is S_0, and so is the minimum for S_2. Clearly the minimum goal calculation functions

$$f : S_0 \to S_1 \text{ and } f : S_0 \to S_2$$

are monotonic. Conversely, if we have instantiated S_1 and S_2 as conclusions (working bottom up) then there is a maximum value with which we can instantiate S_0, to wit $S_1 \cup S_2$. This maximum conclusion function is also monotonic. For the connective $\vee$ the goal sets are interdependent, and a smallest cannot be calculated

$$\frac{S_0 \vdash \phi_1 \vee \phi_2}{S_1 \vdash \phi_1 \quad S_2 \vdash \phi_2} \text{ then } S_0 \subseteq S_1 \cup S_2$$

In contrast, where S_1 and S_2 are known, we can calculate the maximum S_0 which may be inferred. We can create a taxonomy of connective based on the ability to infer *best* (smallest necessary goal or largest possible conclusion) statesets. We represent the calculability by way of *nodes with arrows*, where the presence of an incoming edge requires a stateset from the appropriate subtree, and the presence of an outgoing edge represents the fact that a best stateset can be calculated, given the required best inputs. Thus $\wedge$ gets the graph

(1)  (2)  from $\dfrac{S_0 \vdash \phi_1 \wedge \phi_2}{S_1 \vdash \phi_1 S_2 \vdash \phi_2}\,(\wedge)$

(1) because given S_1 and S_2 conclusions, then the best S_0 conclusion can be calculated, and (2) because given an S_0 goal, the best S_1 and S_2 goals can be calculated (identity!).

To present the taxonomy precisely we need to know what functions we can calculate. Because μM is decidable for a finite-state LTS, all best statesets are calculable in that context. But if we allow an inifinite-state LTS, and define only a small number of operations, we can separate tractable and intractable rules. It's enough to take as our basic operations

$\vec{K}: S \to S'$	$S' \triangleq \{s' : \exists s \in S, a \in K \,.\, s \xrightarrow{a} s'\}$
$\cup : (S_1, S_2) \to S$	$S = S_1 \cup S_2$
$\cap : (S_1, S_2) \to S$	$S = S_1 \cap S_2$
$\mathbf{id} : S \to S$	identity function

and optionally to add the backward transition rule

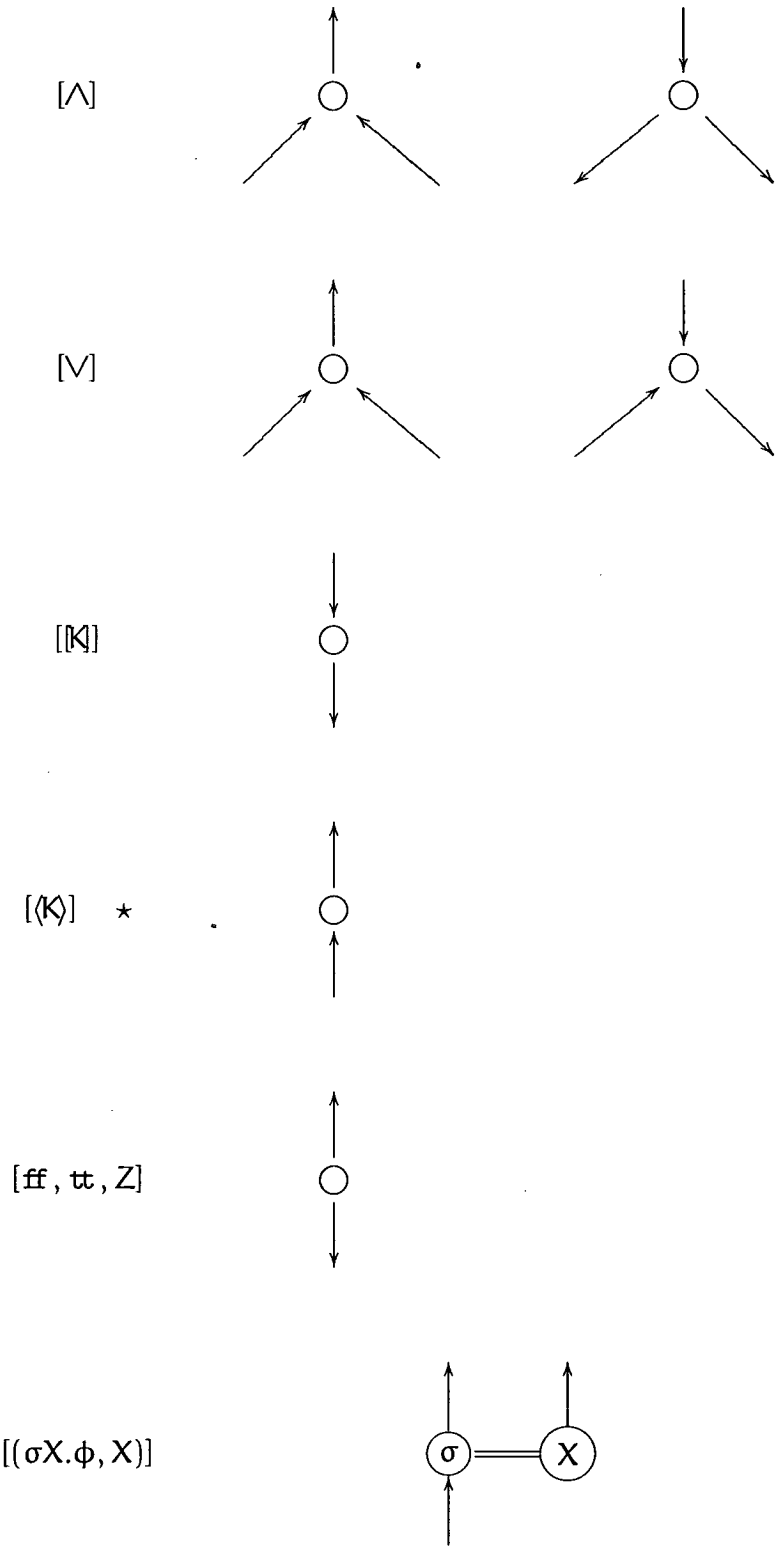
$$\overleftarrow{K}\colon S \rightharpoonup S' \quad S' \triangleq \{s' : \exists s \in S, a \in K \,.\, s' \xrightarrow{a} s\}$$

We mark with $*$ the graphs which depend on the $\overleftarrow{K}$ rule.

The complete set of taxonomic graphs is in Figure 8.2. Each diagram describes a condition by which some known state sets (represented by incoming arrows) allow the calculation of other state sets (represented by outging arrows). Knowledge flows up the way and goals flow down. From the figure we see that for $\wedge$, there are two cases:

- Knowledge of $S_1 \vdash \psi_1$ and $S_2 \vdash \psi_2$ allows us to calculate a maximum $S(= S_1 \cap S_2) \vdash \psi_1 \wedge \psi_2$.
- A goal of $S \vdash \psi_1 \wedge \psi_2$ gives minimal subgoals of $S \vdash \psi_1$ and $S \vdash \psi_2$.

Figure 8.2 Paths by which best stateset can be inferred



The notation can be used for an algorithm to identify *unsolved* nodes. By marking solved nodes (goal or conclusion stateset defined) and propagating marking according to taxonomic graphs (mark a node if all inputs are marked, a bit like Petri nets) an interaction between checker and user can be set up. The user is offered unsolved nodes to solve, and with newly solved nodes the checker can mark extra nodes until all are marked. Sometimes the checker will need to perform stateset comparisons to ensure that a solution is valid. A node can receive statesets both as goals and as conclusions, and clearly the conclusion must be as big as the goal. This points to the final function needed from the LTS interface:

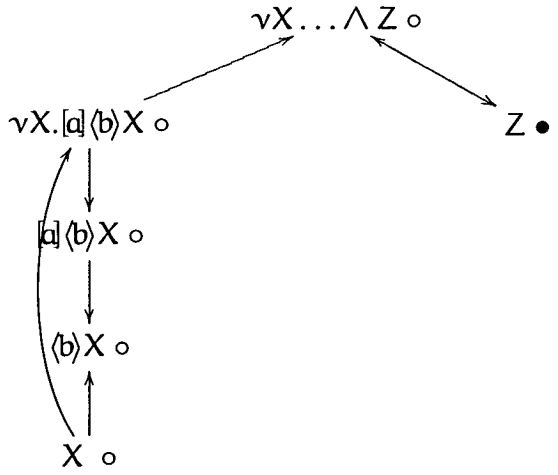
$$\subseteq: (S_1, S_2) \rightarrow \textbf{bool} \quad \text{set containment}$$

A process of dialogue between user and checker proceeds through prompting for statesets for unsolved nodes, and reporting of validity or not of the result.

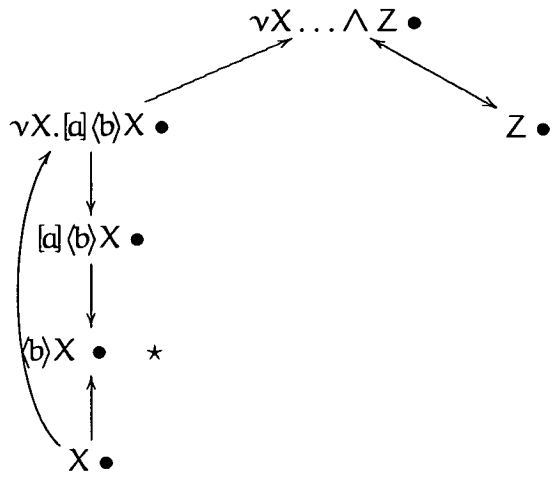
The fixpoint rule requires some comment. The node $\sigma X.\phi$ and all nodes X are identified. They share a single solution stateset which plays 3 roles

- a conclusion for all X nodes
- a conclusion in the tree above $\sigma X.\phi$
- a goal below $\sigma X.\phi$

The tableau/graph for $\nu X.[a]\langle b\rangle X \wedge Z$ illustrates the mechanisms. The marking is • for solved nodes and ○ for unsolved ones, and edges are defined by exhaustively applying the rules of Figure 8.2. Whenever any one has its incoming edges defined, we can add the outgoing edges to the graph. We therefore have a graph which shows how far goals and instantiations can be percolated. The initial graph for $\nu X.[a]\langle b\rangle X \wedge Z$ is:



and after a solution for X is defined:



The node ($\star$) has more than 1 incoming arrow, so it receives a goal (from $[a]\langle b\rangle X$) and a conclusion (from X). These must be checked for $\subseteq$.

We can denote a sufficient set of stateset definitions more succinctly by defining formulae decorated by statesets

$$\phi \triangleq Z|tt|ff|\phi_1 \wedge \phi_2|\phi_1 \vee \phi_2|[K]\phi|\langle K\rangle\phi|\nu X.\phi|\mu X.\phi|S : \phi$$

for example

$$S_1 : \mu X.S_2 : (S_3 : [a]X \vee S_4 : \langle b\rangle P)$$

8.4 User Interaction

We turn our attention to the human interface to the LTS. Our focus is to propose a system which allows the user to iteratively define and explore a state space. It is important to support infinite-state systems, so we describe a method of identifying families of states, and representing them as a single entity. The system is particular to CCS, and we have made a prototype implementation which demonstrated that the concept is plausible. The representation can easily support all the verification functions required by the verification interface to LTS, except for $\overleftarrow{K}$, which is problematic for CCS.

8.4.1 Details

We call our system a CCS Explorer. The key is to identify whether particular generalisations of process expressions are sufficient to capture an infinite set of processes. For example

$$P = a.(P \mid P) \text{ behaves as } P \xrightarrow{a} (P \mid P) \xrightarrow{a} (P \mid P \mid P)$$

and can be captured as an infinite family using the product, giving

$$P = \Pi P_1^1, P \mid P = \Pi P_2^2 \ldots$$

We reconstruct the CCS language to support the definition of families. Instead of

$$\text{Expr} = \quad \text{nil} \mid a.P \mid P\backslash L \mid P[f] \mid P + Q \mid P \mid Q \mid X$$

and

$$\text{Proc} = \textbf{rec}\ \tilde{X} = \tilde{E}$$

we write expressions as

Definition 8.4 (Explorer Expressions)

$$\text{Expr}' = \quad \text{nil} \mid a.P \mid P\backslash L \mid P[f] \mid \Sigma\langle\tilde{P}\rangle \mid \Pi\langle\tilde{P}^n\rangle \mid X$$

This identifies some processes syntactically which under CCS are syntactically not equal (but bisimilar). The syntactic identification makes syntax-based searches more powerful, and syntactic search is our main tool in the explorer. Of course no syntactic identification can be complete for CCS, but we consider that redefinition is of practical value in that it captures common cases. Figure 8.3 shows the CCS rules for the more general terms.

Figure 8.3 Modified Explorer SOS-rules

$\Sigma_{i\in I}\, E_i$ (generalised)

$$\frac{E_j \xrightarrow{\alpha} E'_j}{\Sigma_{i\in I}\, E_i \xrightarrow{\alpha} E'_j}$$

$\Pi_{i\in I}\, E_i^{n_i}$ (generalised)

$$\frac{E_j \xrightarrow{\alpha} E'_j}{\Pi_{i\in I} E_i^{n_i} \xrightarrow{\alpha} \Pi_{i\in I} E_i^{n_i} (i \neq j), E_j^{n_j - 1}, E'_j}\ (j \in I, n_j > 0)$$

8.4.2 Implementation

I prototyped an explorer system as a table (a 2-3 tree) of expressions. The prototype was efficient at matching expressions due to the use of a normalised representation allowing the ordered storage of previously encountered expressions (the candidates for matching).

Normalisation

- Product (Π) processes are dealt with specially. Sets of processes can be represented by a single term.

  $$\Pi\langle P_i^a, Q_j^b\rangle \text{ represents } \{\Pi\langle P^m, Q^n\rangle : a \leq m \leq i \text{ and } b \leq n \leq j\}$$

- redundant ΠP^1 are replaced by P.
- $\Pi P^n, Q^m, P^r$ become $\Pi P^{n+r}, Q^m$
- Σ terms are simplified similarly

Ordering

1. Leading constructor
2. Recursively on sub-expressions
3. Fixed elements of expression

    - prefix name
    - renaming name
    - definition name
    - process-counts for Π

so that all Πs are gathered together, and within that processes with the same structures of subformulae are closest.

In use, a system is explored by defining a set of named processes, and nominating an initial process expression. Processes can be added to the table, and processes can be traced by a syntactic analysis of their expressions to yield successors. Multiple tables can be used, with each representing a stateset of the LTS denoted by the CCS specification. User control of process expansion conforms to the principle of locality which allows local model-checking; that is, the system can be analysed on an as-needs basis, with no requirement to expand process definitions where the processes are not relevant to properties under consideration.

8.4.3 Families, Matching and Generalisation

In basic CCS, infinite state processes are constructed by the regeneration of processes alongside others by a combination of | and recursion. A counter is a standard example:

$$C = \mathrm{up}.(C|C) + \mathrm{down}.\mathbf{nil}$$

The translation to the Π-formalism gives an expression more obviously part of a pattern

$$\Pi C^1 \xrightarrow{\mathrm{up}} \Pi C^2 \text{ is an example of } \Pi C^i \xrightarrow{\mathrm{up}} \Pi C^{i+1}$$

The idea for Π-formulae is that many infinite regenerations follow this pattern, and that we can therefore unite them in a finite representation. Note again that normalisation gives us a version of the expression which is much easier to work with:

$\Pi\langle P^1, \Pi\langle Q^2, R^1\rangle^1\rangle$ is normalised to $\Pi\langle P^1, Q^2, R^1\rangle$

P, Q and R are always in order under the normalised representation

Definition 8.5 (Matching) $P \simeq Q$ iff

$$P = Q \qquad \text{or}$$
$$P = \Pi\langle R_i^a, S_j^b \ldots\rangle, Q = \Pi\langle R_k^c, S_l^d \ldots\rangle$$

Matching is designed to identify candidates for generalisation. Where processes match they can sometimes be replaced by a single process with a large range of indices.

Definition 8.6 (Generalisation) We have already seen, discussing normalisation, the form

$$\Pi\langle P_i^a, Q_j^b\rangle = \{\Pi\langle P^m, Q^n\rangle : a \leq m \leq i \text{ and } b \leq n \leq j\}$$

to say that all members of the shorthand's set are members of the table's LTS stateset. The successor function can easily be calculated across such a family, taking into account that it yields families as successors.

Generalisation may be used in developing a set of states by expansion. As new processes are evolved they can be matched in the table of existing processes. The ordering scheme allows us to search precisely, because all $\Pi\langle P_i^a, Q_j^b\rangle$, whatever the values of a, i, b and j, are stored together and can be found with a range search on $\Pi\langle P, Q\rangle$. If such a match is achieved when a stateset is being expanded, a generalisation of new and old processes can be suggested (by the system to the user) and verified (by the system if the user suggests it).

Using the prototype families implementation I have been able to show that normalisations and generalisations are sufficient to allow the automatic identification of families in some small examples, and thus to provide candidate invariant sets for solving fixpoints. The next stage would be to apply such a system to larger examples, and refine it with the feedback from these. There may be other structures of terms which better reflect invariant sets, or it may turn out that the present versions of families work well; they have practical support in that they came out of the sort of terms which I used to define solution sets in chapter 7.

8.5 Summary

In this chapter I have examined the auditioning methodology and identified some areas in which mechanical support would be useful. I have described the mechanisms which I have constructed for mechanical support in these areas.

First of all, I have been careful not to disturb the pattern of work that anyone using the methodology by hand would adopt. The major target of the work has been the development of proofs of properties, where I have respected the separation of tableaux and LTSs by devising a system for using tableaux to make abstract inferences about state sets independent of language and by

devising a system for exploring state sets within the particular CCS language through the families mechanism.

The families mechanism and the explorer software which I developed for it are therefore of general use, independent of tableaux systems, and positive results with initial examples suggest it deserves some further work to test more practical examples of how good it is at generating solution sets for invariants.

The system for defining goal and result percolation over tableaux is complementary to the explorer. An implementation could be used with an explorer for any language to define constraints on a set of statesets being sought/explored.

Of more theoretical or architectural interest is the identification of the LTS (or in general, the model) as the interface between tableaux (or in general, the proof system) and CCS (or in general, the language). Isolating components to mechanise has necessitated my identifying a technique for interfacing these components; each makes some kind of assertions about the model, and it may be fruitful to develop a workbench of tools where the model is the backbone; only the model explorer needs to work *in* the model rather than to make assertions *of* it.

Chapter 9

Conclusions and Further Work

9.1 Summary

In this chapter I conclude the thesis by assessing the work in relation to what I set out to do, and by examining what further work is suggested by it.

My aim in the thesis was to present a development methodology which is intermediate between informal testing and pure formal refinement. Auditioning as I have set it out appears to fit this niche, and I deal with some details of how successful it is in Section 9.2.

I use Section 9.3 to visit a selection of interesting topics which have arisen from the thesis, and to consider what further work could be done with them.

9.2 Conclusions of the Thesis

The major concern of the thesis has been the design and analysis of a framework for auditioning. In relation to the elements required for auditioning outlined in Chapter 1, I have shown how to

Specify a system using CCS, or any other language which generates an LTS.

Prove properties of the system using the modal-mu calculus.

I have presented a method for taking a proof (expressed as a verity) and translating it into an oracle for language acceptance. This makes the final link in the chain between specification and implementation. More deeply, I have derived some positive results about the power of oracles which suggests that they are good for the job; oracles with greater information are stronger (Theorem 6.35) and oracles can check for erroneous non-termination (Theorem 6.45). The latter result shows my oracles to be strictly more powerful than those generated from intensional mechanisms; defining acceptable behaviours based on proofs of properties can practically be stronger than defining behaviours based on properties alone, as I argued at the outset that it could be.

Finally I have shown through case studies that the development of specifications and proofs for practical systems is viable, though some further work is necessary to have a system which is likely to be taken up industrially.

9.3 Further Work

9.3.1 Practical Application of Auditioning

The obvious next step for auditioning would be to develop an automatic oracle generator. Chapter 6 represents a very precise recipe for taking a verity (straightforward to construct from a μM-proof) and generating an oracle program/process which acts as a trace checker.

A stage beyond this would be to make the oracle generator part of an *integrated development environment*, feeding the output of a proof-tableau editor into the oracle generation program.

An integrated environment in which transition systems and proofs can be iteratively developed using an LTS-Explorer (Chapter 8), attached to an oracle generator and the oracle analogue of a debugger would constitute an attractive system for industrial use. This package of work is really a software engineering task rather than a computer science research task, although from a research point of view it would clarify whether things like the size of OTSs will really be a problem. Of course it is typical of research work to describe a theoretical framework and consider the job done. Developing the theory is the legitimate task of research work, but writing off the rest as trivial is a mistake which can generally be relied upon to permanently frustrate the adoption of the ideas.

Again, practice can be of benefit to theory if the use of a more sophisticated system allows larger case studies to be carried out.

9.3.2 OTS Size Problems

The example proofs I have presented are simple, and the OTSs derived from them are therefore small. But as auditioning is applied to larger examples the size of the OTS will become more troublesome. I can suggest that practitioners are as spare as possible about the properties they really want to check, and the models they need to check them on, but this will not do in general, and there will doubtless be much optimisation of oracle implementations to be done.

One practical observation is that a single oracle implementing the conjunction of interesting properties will probably be more efficient than several separate oracles. An environment should automatically guide practitioners to conjoin proofs.

9.3.3 Oracle Simplification

A more theoretical attack on OTS size problems is to ask whether there are languages which are weaker than languages of oracles but still stronger than intensional languages and still practically useful. It may be that any extensional system, however weak, makes all the difference that is needed.

One might start thus; first of all, let a simplification of an oracle be a function generating a second oracle. A simplification is safe if it does not decrease the language. A simplification must be safe in order not to falsely indicate failure.

Definition 9.1 (Safety) Set $O_2 = f(O_1)$ where f is the simplification. f is safe for O_1 iff

$$L(O_1) \subseteq L(O_2)$$

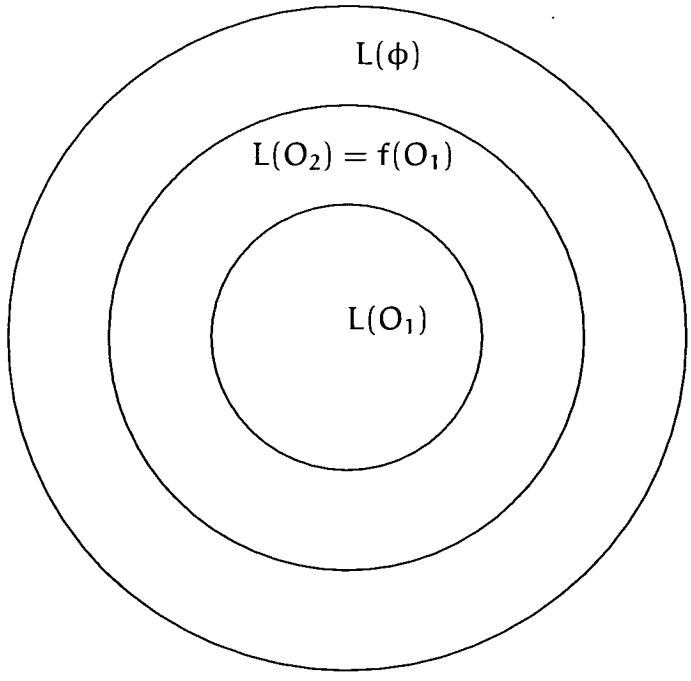
We can set a very lax bound at the other end of the language spectrum by defining a maximal language; it is just the set of all traces accepted in any oracle for the formula in question

Definition 9.2 (Language of Formula)

$$L(\phi_0) \triangleq \bigcup\{L(O_i) : \phi(O_i) = \phi_0\}$$

Extensionality tells us that any property-only oracle must accept any trace in $L(\phi_0)$. More pertinently, it points out the range in which useful refinements exist (Figure 9.1).

Figure 9.1 Containment of Oracle Languages



9.3.3.1 Languages of Formulae

The language of a model allows us to explain the weakness of using oracles based only on formulae. If a trace is in the language of a model which satisfies a formula then it must be in a notional language of the formula. And any trace in that language must be accepted by a safe oracle which only deals with formulae. Thus

$$\{t : \exists M, s \models_\rho \psi \,.\, t \in L(M, s)\} \subseteq L(\phi)$$

and the example of Figure 6.4 gives us

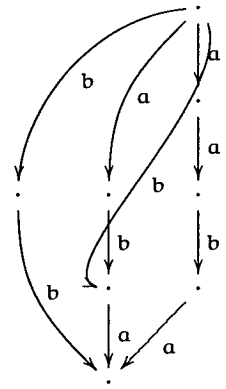
$$a\{\}a\{\}a\{\neg P\} \in L(s_1, \mu X.[a]X \vee P)$$

9.3.3.2 Families in Simplification

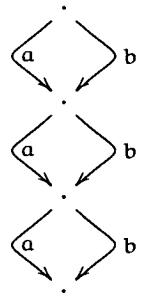
We should distinguish, when we consider the size of transition systems (in our case OTSs), between the number of states and transitions, and the size of the representation. If we start to use more complex representation techniques, such as ones based on families (i.e. see Section 8.4.1) or BDDs [Bry92], some large or infinite systems yield very compact representations. In the following case a uniform descending chain is a good simplification for a complicated collection of states satisfying a termination property. The property

$$\mu X.[a, b]\mathrm{ff} \vee [a, b]X$$

states that chains of (a-s or b-s) eventually expire. The LTS



can be simplified to



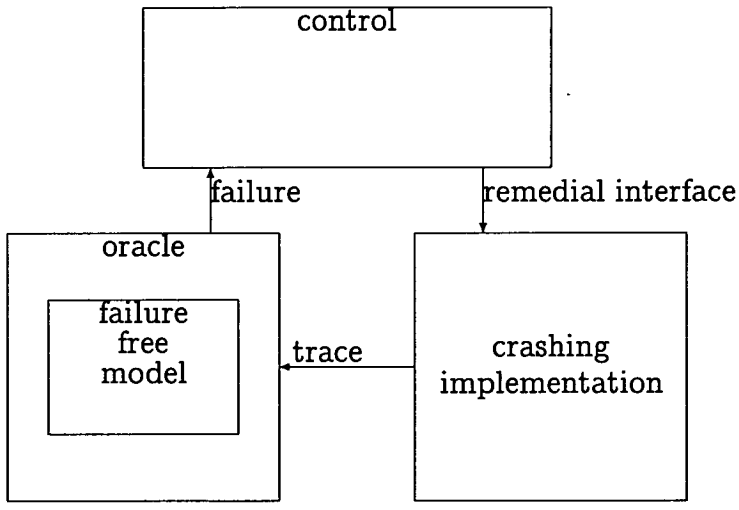
and the simplification can be represented as the family $\{s_{i+1} \xrightarrow{a,b} s_i, i \geq 0\}$ Many other simplifications yielding large savings in representation size may exist.

9.3.4 Auditioning Based Architectures

The combination of monitors based on oracles, and the development of systems through a series of more complex instantiations, suggests an alternative approach to dealing with large systems. The genesis of a complex system can usually be traced to a simple system into which has been added the logic necessary to handle a multitude of exceptional conditions. Eventually, a large majority of the lines of code in a large piece of software can be there solely to cope with something which happens very rarely, or is a logical possibility which because of the particular environment in which the system runs, will never in fact happen.

The auditioning based approach to the problem is to implement a *correct* system as a simple core system without exception handling, plus an external monitor to handle exception processing. The core system needs only a small amount of extra processing to expose a *revert to safe state* interface to a monitor; now exception processing can be handled by a proof that the exceptional condition never arises, together with a monitor process to run when the monitor detects failure of the proof. The monitor interprets failure diagnosis by the oracle as the cue to instigate remedial operations on the running system. Figure 9.2 illustrates this.

Figure 9.2 Correcting an implementation with auditioning



9.3.5 Further Theoretical Questions

The verity concept which we use to represent proofs (Definition 5.16) may turn out to be of more general theoretical applicability. Verities are closely related to proofs and to game strategies, and in their favour they are structurally simpler than the former and more concrete than the latter. It ought to be a reasonably small task to more precisely relate verities to the other two. Are they equivalence classes of proofs ? Do they represent answers to exactly the sets of decisions a player may be asked to make when playing a game according to a strategy ? Are there succinct representations for verities, perhaps in the spirit of BDDs, which help with oracle simplification problems ?

9.3.5.1 Complexity of model-checking

Verities may give an alternative account of why model-checking is in NP $\cap$ co-NP. A standard account of NP problems is in terms of polynomially-verifiable witnesses [Pap94] for solutions to problem instances. Since a verity is just a subgraph of a game-graph it can easily stand as a witness, and the problem becomes merely one of verity-checking; the question then is whether there is a natural algorithm directly in graph and verity terms.

9.3.5.2 Model and Implementation Languages

It would be interesting to relate our language equivalence to the CCS strong bisimulation relation. Is bisimilarity of systems strong enough to require equality of languages, even given the addition of declarations ? One might proceed via Stirling's modal characterisation theorem [Sti97].

9.4 Conclusion

I have presented the concept of auditioning, which attempts to bridge the gap between proof and testing of systems by testing against the more detailed artefacts of *proof of property* rather than just *property*. Auditioning has much practical potential in serious distributed systems development, and its development and analysis gives some insights into the structure of software systems as refinement hierarchies of models at various levels of abstraction.

Appendix A

Fault-Tolerant Epoch Model

Epochal Cluster

```
CONSTANTS
  Nodes ∈ SUBSET Nodenames
  Writes ≜ [node : Nodes, val : Nat, age : Nat, epoch : Nat]
  Reads ≜ [node : Nodes, epoch : Nat]

VARIABLES
  it ∈ [val : Nat, age : Nat]      canonical file stored at the server
  writelocks ∈ SUBSET Nodes        currently granted
  readlocks ∈ SUBSET Nodes         currently granted
  reads ∈ [Reads → Nat]            count of requests waiting at the server
  writes ∈ [Writes → Nat]          count of requests waiting at the server
  ages ∈ [Nodes → Nat]             of its latest write lock
  clock ∈ Nat                      nowness
  epoch ∈ Nat                      perfect (lock-manager) value
  myepoch ∈ [Nodes → Nat]          each node's view
  srvepoch ∈ Nat                   view at the server

PREDICATES
  Init ≜ no locks,no requests
    ∧ reads = [n ∈ Nodes ↦ 0] ∧ writes = [n ∈ Writes ↦ 0]
    ∧ readlocks = {} ∧ writelocks = {}
    ∧ ∀ n ∈ Nodes : myepoch[n] ≤ epoch      last each heard
    ∧ clock ≥ it.age   value is not back from the future
```

Figure A.1: State to Model the Service

Actions

```
ACTIONS
  RReq(n) ≜ submit to server for later doing
    ∧ n ∈ readlocks ∨ n ∈ writelocks
    ∧ reads' = [reads EXCEPT ![node↦n,epoch↦myepoch[n]] = @+1]
    ∧ srvepoch' = MAX(srvepoch,myepoch[n])
    ∧ UNCHANGED writes,readlocks,writelocks,ages,clock,it,
                epoch,myepoch

  WReq(n,v) ≜
    ∧ n ∈ writelocks
    ∧ writes' = [writes EXCEPT
      ![node↦n,age↦ages[n],val↦v,epoch↦myepoch[n]] = @+1]
    ∧ srvepoch' = MAX(srvepoch,myepoch[n])
    ∧ UNCHANGED reads,readlocks,writelocks,ages,clock,it,
                epoch,myepoch

  RCom(n,v,a,e) ≜ commit a read - generate value and age
    ∧ reads[node↦n,epoch↦e] > 0
    ∧ reads' = [reads EXCEPT ![node↦n,epoch↦e] = @-1]
    ∧ e = srvepoch
    ∧ v = it.val ∧ a = it.age
    ∧ UNCHANGED writes,readlocks,writelocks,ages,clock,it,
                epoch,myepoch,srvepoch

  WCom(n,v,a,e) ≜ commit a write
    ∧ writes[node↦n,age↦a,val↦v,epoch↦e] > 0
    ∧ writes' = [writes EXCEPT
      ![node↦n,age↦a,val↦v,epoch↦e] = @-1]
    ∧ e = srvepoch
    ∧ it' = [val↦v,age↦a]
    ∧ UNCHANGED reads,readlocks,writelocks,ages,clock,
                epoch,myepoch,srvepoch

  Is it bad if things can just magically see this ? It shouldn't be,it's just a complication. Should
  also do it for the server,though it rather more clearly has no effect there.

  Notice ≜ see that the epoch has changed
    ∧ ∃ n ∈ Nodes : myepoch' = [myepoch EXCEPT ![n] = epoch]
    ∧ UNCHANGED reads,readlocks,writes,writelocks,ages,clock,
                epoch,srvepoch
```

Figure A.2: Failure-tolerant request/response actions

```
Locking and Unlocking

RLock(n) ≜ lock for reading
  ∧ writelocks = {} ∧ readlocks' = readlocks ∪ {n}
  ∧ myepoch' = [myepoch EXCEPT ![n]↦epoch]
  ∧ UNCHANGED writelocks,reads,writes,ages,clock,it,
              epoch,srvepoch

WLock(n) ≜ write lock
  ∧ writelocks = {} ∧ writelocks' = {n}
  ∧ readlocks = {}
  ∧ clock' > clock ∧ ages' = [ages EXCEPT ![n] = clock]
  ∧ myepoch' = [myepoch EXCEPT ![n]↦epoch]
  ∧ UNCHANGED readlocks,reads,writes,it,epoch,srvepoch

RRel(n) ≜ release is possible whenever not reading
  ∧ n ∈ readlocks
  ∧ ∀ r ∈ Reads : r.node = n ⇒ reads[r] = 0
  ∧ readlocks' = readlocks - {n}
  ∧ UNCHANGED writelocks,reads,writes,it,ages,clock

WRel(n) ≜
  ∧ n ∈ writelocks
  ∧ ∀ r ∈ Reads : r.node = n ⇒ reads[r] = 0
  ∧ ∀ w ∈ Writes : w.node = n ⇒ writes[w] = 0
  ∧ writelocks' = writelocks - {n}
  ∧ UNCHANGED readlocks,reads,writes,it,ages,clock
```

Figure A.3: Lock-related actions of failure-tolerating system

```
Failure-related Actions
ACTIONS
  RFail(n,v,a,e) ≜ fail to read
    ∧ reads[node↦n,epoch↦e] > 0
    ∧ reads' = [reads EXCEPT ![node↦n,epoch↦e] = @-1]
    ∧ e < srvepoch
    ∧ UNCHANGED writes,readlocks,writelocks,ages,clock,it,
                epoch,myepoch,srvepoch
  WFail(n,v,a,e) ≜ fail to write
    ∧ writes[node↦n,age↦a,val↦v,epoch↦e] > 0
    ∧ writes' = [writes EXCEPT
      ![node↦n,age↦a,val↦v,epoch↦e] = @-1]
    ∧ e < srvepoch
    ∧ UNCHANGED reads,readlocks,writelocks,ages,clock,it,
                epoch,myepoch,srvepoch

  Crash(n) ≜ and reboot instantly,but who care about that ?
    ∧ readlocks' = readlocks - {n} ∧ writelocks' = writelocks - {n}
    ∧ epoch' > epoch
    ∧ ∀ n2 ∈ Nodes : n ≠ n2 ⇒
      UNCHANGED ages[n2],myepoch[n2]
    ∧ UNCHANGED reads,writes,it,clock,srvepoch
```

Figure A.4: Unusual case actions of failure-tolerating system

FTInv

```
PREDICATES
  OkLocks ≜
    ∧ writelocks = {} ∨ ∃ n ∈ Nodes : writelocks = {n}
    ∧ readlocks = {} ∨ writelocks = {}

  OkWrites ≜ ∀ w ∈ Writes : writes[w] > 0 ⇒
    ∧ 1 current locker has epoch to himself
       ,and those from previous epochs are older
      ∀ n ∈ writelocks :
        ∨ w.epoch = myepoch[n] ∧ w.age = ages[n] ∧ w.node = n
        ∨ w.epoch < myepoch[n] ∧ w.age ≤ ages[n]
    ∧ 2 no lock means nothing at this epoch
      writelocks = {} ⇒ w.epoch < epoch
    ∧ 3 only one [node,epoch,age] set per epoch
      ∀ w' ∈ Writes : writes[w'] > 0 ∧ w'.epoch = w.epoch ⇒
        w'.node = w.node ∧ w'.age = w.age
    ∧ 4 others ordered by epoch
      ∀ w' ∈ Writes : writes[w'] > 0 ∧ w'.epoch < w.epoch ⇒
        w'.age ≤ w.age
    ∧ 5 w.age < clock
    ∧ 6 w.epoch ≤ srvepoch ≤ epoch
    ∧ 7 w.age ≥ it.age ∨ w.epoch < srvepoch can't do a bad write

  NodeView ≜ ∀ n ∈ Nodes : myepoch[n] ≤ epoch

  Clocks(k) ≜
    ∧ clock ≥ it.age ≥ k
    ∧ ∀ n ∈ writelocks : it.age ≤ ages[n] < clock

  FTInv(k) ≜ Clocks(k) ∧ NodeView ∧ OkWrites ∧ OkLocks
```

Figure A.5: Failure-tolerant Invariant

Appendix B

Glossary of Terms and Symbols

Verity, $(V = \langle V_V, E_V \rangle)$

OTS, (O)

Tableau

$a, \bar{a}, |, +, \setminus, \tau$,**(4.2)**

CCS algebraic connectives

$\xrightarrow{a}$, (4.2)

Successively, the CCS (and general) transition system notation for state transition (4.2) and the OTS transition relationship (6.25).

$\stackrel{a}{\Longrightarrow}$, (4.3

Weak bisimulation "transition"

$\vee, \wedge, \mathtt{tt}, \mathtt{ff}, [K], \langle K \rangle, \mu X.\psi, \nu X.\psi$,**(4.4.1)**

Logical forms of the modal-mu calculus

$[\![\phi]\!]$,**(4.4.2)**

Model-based interpretation of modal-mu calculus formulae

$\vdash, \vdash_\Delta$,**(4.5.1)**

Provability relationship of statsets and formulae via tableaux. Δ is a set of definitions which form context for provability.

$\models$,**(4.4.2)**

Truth relationship for tableaux.

$\prec$,**(5.1)**

Subformula relationship, of modal-mu calculus formulae

$\longrightarrow$

Successively the game transition relationship (5.2) and the verity transition relationship (5.16)

ψ

A single open modal-mu calculus formula

Ψ

A set of open modal-mu formulae

ϕ

A single closed modal-mu calculus formula

Φ

A set of closed modal-mu formulae

$\llcorner - \lrcorner$,**(6.16)**

A representative set of prop-atomic formulae for a general modal-mu formula.

$\ulcorner \_ \urcorner$

A formula, its representative set and all intermediates

Δ,**(6.18)**

The derivative set of a set of formulae.

Δ_V

The derivative of a verity.

Δ_O

The derivative of an OTS.

$\downarrow$,(6.19)

The restriction of a set of verity/OTS configurations to those consistent with a set of formula declarations.

Bibliography

[ABG96] Patrizia Asirelli, Antonia Bertolino, and Stefania Gnesi. Automated testing of safety requirements with the support of a deductive database. In Sandro Bologna and Giacomo Bucci, editors, *Proceedings 3rd. Int. Conf. on Achieving Quality in Software*, pages 145–157. Chapman and Hall, 1996.

[Bat95] P. C. Bates. Debugging heterogeneous distributed systems using event-based models of behaviour. *ACM TOCS*, 13(1):1–32, 1995.

[BDZ89] G. V. Bochmann, R. Dsoulli, and J. R. Zhao. Trace analysis for conformance and arbitration testing. *IEEE Transactions on Software Engineering*, 15(11):1347–1355, November 1989.

[Bei90] B. Beizer. *Software Testing Techniques.* Van Nostrand Reinhold, 2 edition, 1990. New York.

[BH95] Jonathan Bowen and Mike Hinchey. Ten commandments of formal methods. *IEEE Computer*, 28(4):56–63, April 1995.

[BIM95] Bard Bloom, Sorin Istrail, and Albert R. Meyer. Bisimulation can't be traced. *Journal of the ACM*, 42(1):232–268, January 1995.

[Bow95] Jonathan P. Bowen. *Towards Verified Systems.* Elsevier Real-Time Safety-Critical Systems series, 1995.

[Bra91] Julian Charles Bradfield. *Verifying Temporal Properties of Systems with Applications to Petri Nets.* PhD thesis, University of Edinburgh, Department of Computer Science, 1991.

[Bra96] Julian C. Bradfield. On the expressivity of the modal mu-calculus. In *13th Annual Symposium on Theoretical Aspects of Computer Science*, volume 1046 of *Lecture Notes in Computer Science*, pages 479–490, Grenoble, France, 22–24 February 1996. Springer.

[Bru94] Glenn Bruns. Applying process refinement to a safety-relevant system. Technical Report ECS-LFCS-94-287, Laboratory for Foundations of Computer Science, University of Edinburgh, March 1994.

[Bry92] Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[BS92] Julian Bradfield and Colin Stirling. Local model checking for infinite state spaces. *Theoretical Computer Science*, 96(1):157–174, 6 April 1992.

[CD95] Richard H. Carver and Ronnie Durham. Integrating formal methods and testing for concurrent programs. In *(Proceedings)*, pages 25–33, NIST, Gaithersburg, Maryland, 1995. 10th Annual Conference on Computer Assurance.

[CG95] Marsha Chechik and John Gannon. Automatic analysis of consistency between implementations and requirements: A case study. In *(Proceedings)*, pages 123–131, NIST, Gaithersburg, Maryland, 1995. 10th Annual Conference on Computer Assurance.

[CIW97] Daniele Compare, Paola Inverardi, and Alexander L. Wolf. Uncovering architectural mismatch in dynamic behaviour. Technical Report CU-CS-828-97, Dept. of Computer Science, University of Colorado, Boulder, 1997.

[Dam94] Mads Dam. CTL* and ECTL* as fragments of the modal μ-calculus. *Theoretical Computer Science*, 126(1):77–96, 11 April 1994.

[Dij76] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[DKMMS94] L. K. Dillon, G. Kutty, L. E. Moser, and P. M. Melliar-Smith. A graphical interval logic for specifying concurrent systems. *ACM Transactions on Software Engineering and Methodology*, 3(2):131–165, April 1994.

[DR96] L. K. Dillon and Y. S. Ramakrishna. Generating oracles from your favourite temporal logic specifications. In *Symposium on Foundations of Software Engineering*, volume 4, pages 106–117. ACM SIGSOFT, October 1996.

[DV90] R. DeNicola and F.W. Vaandrager. Action versus state based logics for transition systems. In *Proceedings Ecole de Printemps on Semantics of Concurrency*, number 469 in LNCS, pages 407–419, 1990.

[GBD96] Russell J. Green, Alasdair C. Baird, and J. Christopher Davies. Designing a fast on-line backup system for a log-structured file system. *Digital Technical Journal*, 8(3):33–46, October 1996.

[HM85] Matthew Hennessy and Robin Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32(1):137–161, January 1985.

[JLSU87] J. Joyce, G. Lomow, K. Slind, and B. Unger. Monitoring distributed systems. *tocs*, 5(2):121–150, May 1987.

[Koz83] Dexter Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27(3):333–354, December 1983.

[KST94] Stefan Kahrs, Donald Sannella, and Andrzej Tarlecki. The Definition of Extended ML. Technical Report ECS-LFCS-94-300, Laboratory for Foundations of Computer Science, University of Edinburgh, August 1994.

[Lam91] Leslie Lamport. The temporal logic of actions (src report 79). Technical report, DEC Systems Research Centre, about 1991.

[Lar90] K. Larsen. Proof systems for satisfiability in hennessy-milner logic with recursion. *Theoretical Computer Science*, 72:265:288, 1990.

[MC84] L. Morris and C.B.Jones. An early program proof by alan turing (checking a large routine, paper for the edsac inaugural conference, 24 june 1949). *Ann. Hist. Computing*, 6(2):129–143, 1984.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[OAR92] T. O. O'Malley, S. L. Aha, and D. J. Richardson. Specification-based test oracles for reactive systems. In *International Conference on Software Engineering*, volume 14, pages 105–118, Melbourne, Aus, May 1992.

[ORD96] T. O. O'Malley, D. J. Richardson, and L. K. Dillon. Efficient specification-based oracles for critical systems. In Richard Taylor Walter Scacchi, editor, *California Systems Symposium*, pages 50–59, Los Angeles, USA, April 1996.

[Pap94] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[Pet62] C. A. Petri. Fundamentals of a theory of asynchronous information flow. *Proc. IFIP Congress, N-H*, 1962.

[Phi90] M. Phillips. Cics/esa 3.1 experiences. In J.E. Nicholls, editor, *Z User Workshop, Oxford, 1989*, Workshops in Computing, pages 179–185. Springer-Verlag, 1990.

[Plo81] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report 19, Aarhus University, 1981.

[Pot90] Michael D. Potter. *Sets: An Introduction.* Oxford University Press, 1990.

[PW92] D.E. Perry and A.L. Wolf. Foundations for the study of software architecture. *SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992.

[RO92] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[SG96] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, 1996.

[Sti87] Colin Stirling. Modal logics for communicating systems. *Theoretical Computer Science*, 49(2–3):311–347, July 1987.

[Sti92] Stirling. Modal and temporal logics. In *Handbook of Logic in Computer Science, Volumes 1 (Background: Mathematical Structures) and 2 (Background: Computational Structures), Abramsky & Gabbay & Maibaum (Eds.)*, volume 2. Clarendon, 1992.

[Sti95] C. Stirling. Local model checking games. In Insup Lee and Scott A. Smolka, editors, *Proceedings of the 6th International Conference on Concurrency Theory (CONCUR'95)*, volume 962 of *LNCS*, pages 1–11, Berlin, GER, 1995. Springer.

[Sti97] Colin Stirling. Bisimulation, model checking and other games. Notes for Mathfit Workshop Meeting on Games and Computation, June 1997. University of Edinburgh.

[SW91] Colin Stirling and David Walker. Local model checking in the modal mu-calculus. *Theoretical Computer Science*, 89(1):161–177, 21 October 1991.

[Tar55] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[WBW96] Christopher Whitaker, J. Stuart Bayley, and Rod D. W. Widdowson. Design of the server for the spiralog filesystem. *Digital Technical Journal*, 8(3):15–31, October 1996.

[Wey82] E.J. Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):460–465, 1982.

[Win] Geoff Winn. A filesystem specification. DEC Internal Document.