Structural usability techniques for dependable HCI.

Gimblett, Andy

Structural Usability Techniques for Dependable HCI

Andy Gimblett

Submitted to the University of Wales in fulfillment

of the requirements for the degree of

Doctor of Philosophy

May 2014

Swansea University
Prifysgol Abertawe

Department of Computer Science

Swansea University

Summary

Since their invention in the middle of the twentieth century, interactive computerised systems have become more and more common to the point of ubiquity. While formal techniques have developed as tools for understanding and proving things about the behaviour of computerised systems, those that involve interaction with human users present some particular challenges which are less well addressed by traditional formal methods. There is an under-explored space where interaction and the high assurances provided by formal approaches meet.

This thesis presents two techniques which fit into this space, and which can be used to automatically build and analyse formal models of the interaction behaviour of existing systems.

Model discovery is a technique for building a state space-based formal model of the interaction behaviour of a running system. The approach systematically and exhaustively simulates the actions of a user of the system; this is a *dynamic analysis* technique which requires tight integration with the running system and (in practice) its codebase but which, when set up, can proceed entirely automatically.

Theorem discovery is a technique for analysing a state space-based formal model of the interaction behaviour of a system, looking for strings of user actions that have equivalent effects across all states of the system. The approach systematically computes and compares the effects of ever-longer strings of actions, though insights can also arise from strings that are *almost* equivalent, and also from considering the meaning of *sets* of such equivalences.

The thesis introduces and exemplifies each technique, considers how they may be used together, and demonstrates their utility and novelty, with case studies.

Declaration

This work has not been previously accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

Signed	[illegible]	(candidate)
Date	16 May 2014	

Statement 1

This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by footnotes giving explicit references. A bibliography is appended.

Signed	[illegible]	(candidate)
Date	16 May 2014	

Statement 2

I hereby give my consent for my thesis, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed	[illegible]	(candidate)
Date	16 May 2014	

To my parents

Acknowledgements

First and foremost, I owe a huge debt of gratitude to my supervisor, Harold Thimbleby, for his guidance, insight, engagement, unfailing encouragement, apparently inexhaustible patience, and deep compassion and kindness, without which this thesis would surely not have been realised.

I am deeply indebted to Paul Curzon and José Creissac Campos for acting as examiners of this work, and for the great feedback they gave me during and after my *viva*; and also to Michael Harrison and Anke Dittmar for their encouragement, interest, and kindess throughout this process, and for reading and very helpfully commenting on drafts of large parts of this work as it took shape.

I have been very lucky to work with many excellent colleagues in my time at Swansea, and I would particularly like to thank Abigail Cauchi, Patrick Oladimeji and Paolo Masci for all the fun times we've had together while working on this stuff, and Karen Li, Dan Craggs, Ben Spencer, Parisa Eslambolchilar, Matt Gwynne, Philip James, Tom Owen, Jen Pearson, Simon Robinson, Matt Edmunds and, I am sure, many others I have forgotten to mention, for their friendship and encouragement over the last few years.

This work has been funded by EPSRC grants EP/F020031/1 ("Formally Based Tools for user interface analysis and design") and EP/G003971/1 ("Healthy interactive systems : Resilient, Usable and Appropriate Systems in Healthcare"), for which I am deeply grateful.

Above all, I am thankful for the love and support of my family and other loved ones: again, there are too many to mention, but I must particularly thank my parents; my brothers and their families; Basheera, Carys, and Markéta. Your constant encouragement and belief in me have carried me this far.

Table of Contents

Chapter 1

Introduction

Since their invention in the middle of the twentieth century, interactive[1] computerised systems have become more and more common to the point of ubiquity. Desktop calculators, automated teller machines, video games, personal computers, aircraft control systems, nuclear power station control systems, medicine infusion pumps, … Each of these systems has a computerised information-processing part, and an interactive part through which humans control the system and receive feedback as to its status and behaviour; in the case of ATMs, aircraft, nuclear power and medical systems, there is also a physical counterpart under control — the *plant* — and the systems are *critical*: their correct behaviour is of vital importance.

Traditional formal approaches for producing assurance that systems behave as intended are expensive, both in terms of computation and the human expertise required to implement them. Techniques such as formal specification, theorem proving and model checking are very powerful and have proved transformative where used, but due to their expense they remain in niches combining high criticality with deep pockets — such as defence, aerospace, and microprocessor design. Extending their use into the mainstream of system development remains an area of research. [CW96; Sch96; Rus07; Woo+09; Par10]

In the mainstream there are of course other tools and techniques for increasing our confidence in system

[1]Throughout this thesis, by *interactive system* we mean one that has some component with which human users interact — as opposed to the more general class of *reactive* system, i.e. those that consist of interacting subsystems (e.g. a handshaking protocol, any system specified using the process algebra CSP, etc.).

correctness — but they tend to be ones that are easily automated and subsumed into existing toolchains and languages. To pick three examples:

- The use of *type systems* and *strong typing* (e.g. as described in [Mil78]) in programming languages, which enable the compiler to automatically rule out the existence of whole classes of bug; a form of *static analysis*, this requires careful design up-front of the language and its toolset.
- *Model based* approaches, where formal models of some aspect of a system are transformed automatically to code. Key examples here include the UML family of languages [RJB99] and associated tools such as Rational Rose[2]; and frameworks such as Ruby on Rails [BK07], which can be used to rapidly produce interactive web-based systems based on a simple data-model description language.
- *Test-driven development* [Bec03], where automated (but hand-written) tests of software behaviour (at multiple levels of granularity) are placed at the heart of the software engineering process. This has become particularly widespread in industry within the last decade, in tandem with the growing popularity of *dynamic* programming languages (and, arguably, as a response to the inherent limitations of static analysis in such languages).

Interaction poses many challenges of its own. The ergonomics, human factors, usability and user experience (UX) movements have all emerged as responses to these challenges, placing humans firmly at the centre of the design process in order to produce interactive systems that work *for* people rather than against them — as computerised systems so frequently seem to do. These approaches also require practicioners with a high degree of skill and are difficult (if not inherently impossible) to automate. What's more, they tend to be oriented towards 'soft' questions of ease of use, ease of learning, pleasure, and so on — with 'harder' issues such as safety and dependability unattended to. This is partially because the techniques advocated in these movements are in general *non-exhaustive*: recording a sample user's actions and analysing their mistakes and hesitations can help identify issues and areas for potential improvement but (unlike type checking in programming languages, say) can not provide certainty that certain problems are completely absent throughout the system. Exhaustivity is simply impractical when humans are involved [Dwy+04]. Yet, serious flaws may exist in the unexplored parts of the system, only

[2]`http://www.ibm.com/developerworks/rational/products/rose/`

to manifest when it is put into use — potentially with life-threatening or mission-critical consequences [Thi07b].

There is, then, an under-explored space where these concerns meet: a space where the systems are interactive, and we wish to learn or prove things about the systems' interaction behaviour with certainty and exhaustivity. We might, for example, specify a system's interaction behaviour using a modal logic of some sort, and then prove properties of that specification using a model checker (see section 2.3 for some examples of this kind of work). Again, though, such techniques require great expertise on the part of the investigator — in particular facility with the formal languages used (very different from programming languages: indeed, at an entirely different conceptual level) and their associated tools. A further problem is that specification up-front sits uneasily with the highly flexible, iterative nature of modern software development, where requirements emerge and change as the system takes shape over time [SB82]: formal methods techniques typically require the investment of much effort before implementation, and as such tend either to discourage iterative development, or to be bypassed where it takes place, which can lead to the situation where a model is formalised and verified prior to implementation, but the implementation itself may have unknown bugs independent of the model.

In practice, user interfaces are rarely formalised, and are generally implemented in an imperative style, based on side-effects, with complex timing/queueing issues, and low level error management — all of which are difficult to formalise. Furthermore, they are implemented in terms of APIs and frameworks that, even for routine tasks such as number entry, are unformalised and usually harbour bugs of their own [TC10]. Many UIs also have a significant hardware component, particularly in safety critical settings, and are subject to unspecified hardware-oriented interactions (e.g. key bounce and display size limitations). Hence even if traditional formal methods *are* used for interactive systems, 'full stack' assurance of the final system's quality is rarely achieved.

An alternative approach is to use processes that can reliably and automatically analyse actual system implementations, even as developed using mainstream methods. Such processes aim to provide some of the benefits of using formally-based tools without the high cost and inflexibility of conventional formal methods — and to integrate well with existing tools and techniques. These efforts should be seen in the context of Reason's "swiss cheese model" [Rea00]: adding new layers of defence against error to those already routinely used by developers.

The core of this thesis presents two techniques which, we argue, fit this bill, and which can be used to automatically build and analyse models of the interaction behaviour of existing system implementations. They are:

1. **Model discovery** — a technique for building a state space-based formal model of the interaction behaviour of a running system. The approach systematically and exhaustively simulates the actions of a user of the system; this is a *dynamic analysis* technique which requires tight integration with the running system and (in practice) its codebase but which, when set up, can proceed entirely automatically.

2. **Theorem discovery** — a technique for analysing a state space-based formal model of the interaction behaviour of a system, looking for strings of user actions that have equivalent effects across all states of the system. The approach systematically computes and compares the effects of ever-longer strings of actions for equivalence, though insights can also arise from strings that are *almost* equivalent, and also from considering the meaning of *sets* of such equivalences.

While they could be used separately, clearly these two techniques are synergistic, with model discovery providing input for theorem discovery, and theorem discovery raising properties and questions which can lead to further model discovery tasks. Each of the techniques is conceptually quite simple, and within the capabilities of any competent programmer, but nonetheless powerful, automated, embeddable in existing toolchains, and open to numerous extensions and specialisations — some of which we propose and discuss.

There do exist approaches (e.g. Petri nets, CSP, statecharts, etc. — see chapter 2) that can formalise UI features, and that are able (to varying degrees) to address usability and safety issues, but usually as part of an approach in the style criticised above; the techniques described in this thesis are distinctive in two key ways. First, although we can check user interfaces implement required properties, a distinctive benefit is that we can *discover* unforeseen emergement properties of the system that may be salient from a usability or dependability point of view, but which had not been previously considered. Second, the techniques are simple and based in practical programming rather than in advanced mathematics; as such, they are eminently accessible to any competent programmer.

We envisage their integration into the software development toolchain as a mechanism for discovering, specifying, and performing *regression tests* on a system's interaction behaviour — where a property is identified as being desirable (or not) and the techniques provide a means for automatically checking for that property's presence (or absence) — indeed, in the case of theorem discovery, unthought-of properties may be learnt of. Automated testing of interaction behaviour is still less widely supported and performed than unit testing of a system's internal behaviour, though functional testing tools such as Selenium [HK06; Sir10] are helping to close this gap. It is our hope that the techniques described in this thesis can also make a contribution in this area.

Thesis overview

The rest of this thesis is structured as follows. Chapter 2 presents a survey of relevant background material in order to contextualise our contribution; this includes formal methods and their application to HCI, with a particular focus on model checking and on state space-based techniques; material related to reverse engineering; HCI as applied in the medical context; and lightweight formal methods in the tradition from which our techniques have emerged. Chapters 3 to 6 form the core of the thesis, and our original contribution. Chapter 3 describes model discovery in detail, including a formalisation of the necessary API in order to implement it, the basic algorithm, and some extensions and variations. Chapter 4 presents four case studies of model discovery in action. Then, chapter 5 describes theorem discovery in detail, including a mathematical formalisation, the algorithm, a performance analysis, and discussion of extensions to the basic setting. Chapter 6 describes our implementation of theorem discovery in a Haskell library and command-line tool, and presents three case studies of the use of that software, analysing models produced by model discovery as described in chapter 4. Finally, in chapter 7 we conclude, reflect, and propose future work.

The case studies in chapters 4 and 6 are based on simulations of actual devices. This simulation-based approach has both limitations and advantages. It would, obviously, be ideal in practice (though not necessarily for a research programme) to apply our techniques directly onto actual devices or at least their underlying software, but as this was not feasible, as the software was not available for the devices we were interested in, we took the simulation approach. The systems in question are small enough that we can be confident that our simulations are accurate to a point where they potentially provide useful insights into actual device behaviour, and indeed this has been the case: for example, for the HS-8V calculator we found some surprising results that are valid for the actual device. More usefully, we have shown our techniques scale to a real level of complexity, and therefore are likely to have wider applications in user interface software development.

Much of the work in chapter 3 and section 4.2 was presented in [GT10], co-authored with Harold Thimbleby; similarly, parts of chapter 5, and sections 4.5 and 6.5 were presented in [GT13], also co-authored with Harold Thimbleby. In both cases, I have presented only those aspects of the work that are directly attributable to me. The models discussed in case-study sections 4.4 and 4.5/6.5 were produced by Patrick Oladimeji and Harold Thimbleby, respectively, but the commentary and analysis given here is my own work — and I thank them for enabling this by sharing those models with me.

Chapter 2

Background and Related Work

Contents

In this chapter we present some background and related work in order to set the scene for the techniques which form the body of this thesis. We provide a (largely historically-oriented/chronological) overview of a number of relevant themes. The central theme is the application of formal and semi-formal approaches to questions of human-computer interaction and the design of interactive systems. As such, we begin with a discussion of formal methods in general, and then consider some major (and relevant) aspects of their application to HCI, leading to a discussion of the family of semi-formal techniques targetting *dependable HCI* of which our techniques are a member. In the final section we consider some examples of the application of human factors/HCI to the medical domain, as this is a key motivating domain for dependable HCI, and relevant to some of our case studies.

2.1 Formal methods

Formal methods is the field of computer science in which mathematical models are used — at various stages of the development life-cycle — in order to develop systems, typically with the intention of providing high levels of certainty as to their reliability, safety, efficiency, etc. In [CW96] and [Woo+09], Clarke and Wing, and Woodcock et al., respectively survey the state of the art as of 1996 and 2009, and outline future directions envisaged at those times.

Clarke and Wing note that at that time techniques were beginning to scale, and tools to mature, to the point that real world examples could be dealt with more commonly, though there were still difficulties to be overcome. They note that the process of formal specification is not only useful as an input to a later verification stage, but also in its own right, as it frequently reveals design flaws and ambiguities, and produces an artifact which can not only be analysed and transformed but also used as a communication tool between stakeholders. Thus, the process of formalisation is useful not only in and of itself, but also because it forces practitioners to be more rigorous than they might otherwise be, and to consider details which might otherwise go overlooked — and tool support is particularly useful in this regard. This is a recurrent theme, echoed for example by Ambramson and Pike [AP11] and others. That paper focuses on the requirements engineering stage of development, but the point is applicable outside that context, and later in this thesis we make a similar argument, in the implementation stage, regarding our use of the programming language Haskell.

Clarke and Wing describe two major approaches, still pre-eminent today, to the task of *verifying* system specifications (i.e. checking that they meet certain properties, which can include being a faithful refinement of some other specification, for example): *model checking* (considered in more detail in section 2.3) and *theorem proving*, where systems and properties are encoded in formal logics, and deductive techniques are applied (typically semi-automatically) in order to prove or falsify properties of the systems. Looking ahead, they emphasise the importance of integration of formal methods techniques and tools into traditional software development workflow, and that it is critical that they be easy to learn.

Wookcock et al. note that by 2009 the use of formal methods had become more common, to the point that they are able to describe several major industrial examples. They note that formal methods are proving successful but still lack widespread adoption except for the niche of critical systems in certain domains. Their expectation is that diffusion into the mainstream will continue to take place gradually, with formal methods not explicitly adopted as a distinct technology, in general. They also consider 'lightweight' and 'semi-' formal methods, the increased application of formal methods in a focused manner to particular system aspects, and the importance of toolchain integration and automation.

Rushby [Rus07] echoes a number of those points, commenting on some particular recent developments of note which have been instrumental in enabling greater uptake of formal methods in the mainsteam of software engineering. In particular, he notes: that tools are finally delivering practical benefits for modest investment (both in terms of time and required expertise); that practitioners are realising that outcomes other than full formal correctness proofs of entire systems can still be valuable (e.g. static checks, test case generation); and that model-based development (see discussion of [UIM92] in section 2.2, for example) is bringing formally analysable artifacts into the development lifecycle early and throughout, thus encouraging the use of automated (i.e. formal) processing techniques on those models. The first two points are particularly germane to this thesis: we describe techniques which are implementable by any competent programmer, and envisage those techniques as forming just one part — but a useful part — of the toolkit employed in order to develop dependable interactive systems.

An interesting insight into the historical perception of formal methods is provided by Hall's "Seven Myths of Formal Methods" [Hal90], and Bowen and Hinchey's "Seven More Myths..." [BH95]. The seven myths, seen as unhelpfully holding back the field, are: perfection is possible; program verification is the only thing you can do; they're only useful for safety critical systems; they require highly trained mathematicians; they increase the cost of development; they're unacceptable to users; they're not used on real, large-scale software. The 'seven more' myths, similarly, are: they delay the development process; they lack tools; they replace traditional engineering design methods; they only apply to software; they are unnecessary; they are not supported; they are always used by formal methods people. While some of these 'myths' probably remain more true today than one might hope, they act as an important point of focus and reflection for the field.

Finally we note that it has long been recognised that it is impossible to complete specification before implementation, but rather that implementation inevitably feeds back *into* specification. Swartout and

Balzer [SB82] argue that this occurs for two reasons: first, because of physical limitations in implementations (e.g. choosing to use an array instead of a linked list means adding a bound to the specification); and second because of imperfect foresight (in the natural manner). This emphasises the need for iterative development, even (indeed, especially) when applying formal methods, and thus flexible (and, as much as possible, automated) tools to support that process. We argue that the techniques described in this thesis fit that bill.

2.2 Formal methods in HCI

Attempts to apply formal notations and reasoning methods to interactive systems have a long history. For example, in 1969, Parnas [Par69] considered problems related to designing the user interface of a program in a top-down manner, at a time when 'user interface' really meant 'command language'; that paper highlights the previously unrecognised importance of interaction as a concern in designing such languages, and proposes a form of state transition diagram as a mechanism for reasoning about this problem. Using this approach, Parnas uncovers several design problems in a real-world example. Jacob [Jac83] also uses transition diagrams to represent interaction behaviour in the specification of a command language, by augmenting transitions in the diagrams with responses to be performed by the system when the transition takes place — somewhat analogously to the treatment of non-terminals in [Shn82]. In that paper, Shneiderman proposes an extension to BNF called *multi-party grammars* "to describe the actions of several parties, people or machines, using the same notation", where non-terminal symbols in the grammar are associated with actions of either the user or the computer. He then looks at their application to particular problems in the specification of human-computer interactions, including considering how to deal with I/O using modalities other than just hard copy, including and in particular a character-oriented screen divided into windows. Similarly, Wasserman [Was85] presents a formal executable description of user interaction with a (text-based, command-langauge) system, based on augmented state transition diagrams (and a textual representation of same); he discusses perceived shortcomings of pure state transition diagrams for this purpose (diagrammatic complexity; inability to specify output; difficulties specifying choice in logical flow; lack of encoding timing), and describes the use of automated tools to support this (e.g. an interpreter for prototyping).

Those four papers are notable in that they apply and extend 'classic computer science' formalisms (i.e. BNF and transition systems) into the interaction domain, an approach which remains important today and which this thesis essentially represents another example of. However, significant work has also been done in inventing entirely new formalisms specifically targetting aspects of user interaction — and much work has been done in the space between these two extremes.

A key example of a formalism invented specifically for reasoning about interactive systems is Dix's PIE model (and its extensions such as red-PIE) [Dix91] — these are extremely abstract models of interactive systems, indeed it is hard to imagine a model which could be more abstract and still worth thinking about. The basic PIE model relates a set of possible programs P to a set of possible effects E (which can be anything, e.g. an image on a screen, a print-out, or the movement of a robotic arm) via an interpretation function $I : P \rightarrow E$. This simple model provides an abstract framework for reasoning about concerns such as observability of effects, and reachability of program states. For example, the *strong reachability property*, that "you can get from anywhere in the system to anywhere else" may be written succinctly as:

$$\forall\, p, q \in P \bullet \exists\, r \in P \bullet I(pr) = I(q)$$

The PIE model can be extended to the red-PIE model, adding spaces for display D and result R, both of which E map on to, recognising that what is seen by the user on a display and the final result (e.g. a print-out) may differ, and specifically allowing reasoning about the behaviour of WYSIWYG ("What You See Is What You Get") systems. PIEs are very high-level 'black box' models of systems, in complete contrast to the very concrete state space models which are the subject of the techniques described in this thesis, but they are significant as such, and historically, and are also relevant as the context of Dix's formal reasoning concerning the behaviour of *undo*, which we utilise in section 5.5.4.

There are a number of significant threads in the field of formal methods in HCI after this point, some of which we expand on in their own sections below. A full survey is beyond the scope of this chapter, but here we pick out some interesting examples.

In [HVV91] de Haan et al. survey techniques for modelling users' required knowledge and expected performance, in order to evaluate and predict a system's usability. Examples of such techniques include ETAG (Extended Task Action Grammar) [Tau90] and GOMS (Goals, Operators, Methods and Selection

Rules) [CNM83]. They note that such modelling techniques do not aim to replace empirical user testing, but can allow certain questions to be answered earlier in the design process and with less effort and cost than testing. We make a similar argument regarding the techniques in this thesis: they augment rather than replace user tests, and can indeed raise questions which provide a focus for such testing.

The 1992 UIMS tool developers workshop [UIM92] introduced the Arch model and Slinky metamodel, frameworks for designing interactive systems based on modelling the nature of the data passing between the UI and non-UI parts of such systems, based on five layers: interaction toolkit components, presentation components, dialogue components, domain adaptor components, and domain-specific components. This way of thinking about interactive systems has proved very influential subsequently, particularly with regards to *model-based development*, which aims to use formal models throughout the development life-cycle, replacing the informal models historically used for, e.g., requirements analysis. For example, in [Lim+05] Limbourg et al. describe *UsiXML*, a UIDL (User Interface Description Language) allowing UIs to be specified and developed at multiple related levels of abstraction, supporting automated model-to-model transformation (via standard graph transformation techniques) at each stage in order to provide a full model-based development 'stack', with four levels of abstraction: tasks and concepts; abstract UI; concrete UI; final UI, with different kinds of entities in the models at each layer.

In [Bum+95], Bumbulis et al. use formal methods to prototype a system and reason about it using theorem proving (in higher-order logic, HOL [GM93], and PVS [ORS92]); however they note that while they are able to prove standard properties of safety and liveness, "formalizing exactly what constitutes a good interface is an open problem" at that time.

In [HC96], Hussey and Carrington use Object-Z (an object-oriented extension to the Z specification language) to compare the MVC [Ree79] and PAC [Cou87b] architectures; and in [Doh98], Doherty introduces a specification approach for interactive systems based on Object-Z.

In [DH97], Dearden and Harrison argue for the development of a generic model of a class of interactive systems at an intermediate level of abstraction, in order to obtain "wider reusability than detailed specifications of a single system, but greater expressiveness and support for software development than fully general abstract models" — looking to hit a spot between the extreme abstraction of models such as PIEs (which "lack expressiveness and operationality") , and the specifications of individual systems otherwise popular in the literature at that time (which "offer limited re-usability").

In [BF99], Bowman and Faconti apply formal methods to questions of cognitive modelling, by specifying a particular 'information processing'-based model of human cognition (*Interacting Cognitive Subsystems*) formally in LOTOS [ISO89] (a process algebra with a rich data language) and then verifying certain properties against it, by hand and with the aid of a simulation tool.

In [CB02], Curzon and Blandford describe a generic formal model of principles of cognition expressed in higher order logic, specify cognitively plausible behaviour, and semi-formally derive design rules from that model that, if followed, prevent certain kinds of erroneous actions. Building on this, in [CRB07], Curzon et al. describe the use of HOL in a generic approach to formally model both devices and cognitively plausible human behaviour, producing specific user models which are then verified for certain properties such as absence of post-completion errors.

In [Bla+08], Blandford et al. compare eight usability evaluation methods of various kinds, ranging in formality from highly formal approaches such as state-transition networks (STN) and Z, to very informal methods such as cognitive walkthrough and heuristic evaluation; their aim is to catalogue the different kinds of insights the various methods can provide. They note, for example, that "Z and STN, although not designed to identify usability problems, were reasonably effective at supporting the identification of system-related problems such as the lack of an 'undo' facility, redundant operators, and long action sequences."

In [DHF08], Dittmar et al. propose the use of *higher order processes* for formally modelling interactive systems in a process-algebraic style, emphasise the recursive nature of interactive systems, and introduce a tool (HOPS) for prototyping systems in such a way. This is followed up in [DF09] which draws out how to do task-based design in this framework, by sketching and refining ConcurTaskTree-like (CTT, [Pat99]) models in HOPS. In [CP09], Combéfis and Pecheur analyse mode confusion via a bisimulation analysis between a given system model and a CTT model of the user's mental model of the system.

In [BB10], Bolton and Bass describe the use of SAL[1] to model a patient-controlled analgesia pump, and particular scaling problems encountered therein. Their response is to use slicing and data abstraction in order to make the models tractable (e.g. only allowing the millilitres unit, excluding milligrams and

[1] `http://sal.csl.sri.com/`

micrograms). They produce independent models of the human mission, human task behavior, human-device interface, device automation, and operational environment.

In [Mas+11; Mas+13], Masci et al. look at formalising the notion of predictability of a user interface using higher order logic; they specify two real number entry interfaces from infusion pumps (including the BBraun example from sections 4.3 and 6.4), and attempt to verify the predictability property in each with SAL. With the BBraun they find the same issues we do, namely that the memory facility makes its behaviour unpredictable (at least in the formalisation of predictability used), though their technique does not reveal anything about inconsistent behaviour around *minimum* values.

2.3 Model checking in HCI

As noted above, there are two key strands in classical formal methods, namely model checking and theorem proving. In the previous section we met some examples of the latter technique, but in this section we concentrate in particular on model checking, which shares a number of features with the techniques described in this thesis: state space-orientation, full automation, and a strong concern with sequences of actions — indeed we note that theorem discovery can be seen as somewhat dual to model checking, in that the latter involves proving the truth or falsehood of given statements within some model, whereas the former involves producing statements that are true within some model.

Model checking [CES86; CGP99] is a technique for automatically verifying properties of finite-state systems. In brief: given a system and some desired behavioural property of that system, the property is verified by exhaustively enumerating and exploring all reachable states of the system, looking for states in which the property does not hold; if no such states are found, the property is verified; otherwise, the property is falsified.

Model checking is probably *the* key example of a formal method which has successfully moved from academic research into industrial use [Rus07]. The following attributes are particularly of note and are widely seen as critical for its success:

1. It is *exhaustive* — model checking verifies properties over entire systems; this is a great advantage over techniques such as simulation and testing, particularly in settings where

unnoticed errors are costly and humans are unable, realistically, to consider all possibile errors.

2. It is highly *automated* — once the system and properties have been specified, the model checking algorithm proceeds fully automatically. (Having said that, *applying* the algorithm is nonetheless really an iterative technique — writing and debugging specifications, determining the right abstractions, etc. is a process akin to programming. Theorem proving has historically tended to be seen as a more involved process, but in context, and particularly with the advent of powerful automated proof tactics (e.g. 'grind'), the distinction is not strong.)

3. It provides *counterexamples* — when a property is falsified, model checking finishes with a counterexample, i.e. the state at which the property is found to be false, and the path (history of actions) that led to that state; by examining the state and path, the investigator can attempt to discern *why* the property is false, and thus effectively 'debug' the system or — it often turns out — the specification. This is a great advantage over theorem-proving techniques, which are often far more inscrutable when properties are found to be false.

Two key issues with model checking are that it suffers from 'state space explosion' (ameliorated somewhat by the use of *symbolic* model checking, rather than explicitly representing all states), and that it is (usually) restricted to *finite*-state systems.

Model checking operates on two entities — the system and the specification — both of which must be represented in some suitable formalism. Typically the system is represented using a modal logic (e.g. MAL, Modal Action Logic) or as a state transition system, and the specification of properties is usually written in a temporal logic (e.g. CTL [CES86], LTL [Pnu77]) — though there are variations on these themes, e.g. the process algebra CSP [Ros98].

Having introduced model checking in general, we now consider some particular examples of its application in our domain.

In [JH92], Johnson and Harrison use temporal logic to express dynamic requirements within a formal model of interaction. They introduce a tool, *Prelog*, to support specification and prototyping of interactive control systems, and focus on questions of decomposition and device abstraction.

In [DCH97], Dwyer et al. describe model checking specifications of GUI implementations, using CTL, SMV [McM92], and *abstract transition systems*, which are abstractions over (complete) concrete transition systems; the main focus of the paper is on identifying exactly which abstractions are suitable for working with GUIs.

In [LC99], Luttgen and Carreño describe the use of model checking (using and comparing Murϕ [Dil96], SMV and Spin [Hol91]) in order to analyse mode confusion in an avionics setting.

In [PS01], Paternò and Santoro describe a prototype environment that integrates a tool for task modelling with a tool for model checking, in order to support modelling and analysing multi-user interactive applications in a safety-critical setting. Here task models are represented as ConcurTaskTrees and compiled to LOTOS, and the approach is exemplified with an air traffic control case study.

In [Rus02], Rushby describes the use of the Murϕ tool to encode a system specification and a corresponding user mental model, and to check for inconsistency between them leading to automation surprises (in particular mode confusion). The case study is a real observed problem in a flight simulation involving an autopilot, though the models are quite simple (with less than 15 states each). The same situation had been analysed using a different (manual) technique by other authors, but Rushby's analysis found further problems with the fix suggested there (and is automated). He argues for the importance and utility of abstraction, and suggests that refinement-based analysis (as used by CSP) may be necessary in order for the technique to scale up.

In [KSH08], Kamel et al. describe a technique for model checking certain classes of usability properties of multimodal user interfaces (MUIs). The properties, collectively called CARE, concern the complementarity, assignation, redundancy and equivalence of the various modalities available for a given interface. An earlier paper (in French) describes how to encode MUIs and their CARE properties as transition systems and CTL formulae respectively, ready for model checking using (in their case) SMV. The later paper builds on that by showing how to decompose such systems and properties according to subsets of the full set of modalities on offer, in order to make the model checking task more tractable. A small example of a mobile phone's menu structure (with two modalities — push button and voice) is given.

In [Bas+11] Bass et al. model and look for mode confusion in hybrid systems (i.e. ones with continuous dynamical components) using infinite bounded model checking (SMT [Bar+09]), using the SAL and

Yices[2] tools, with an Airbus A320 Speed Protection System example.

In [CH97], Campos and Harrison review the state of formal verification of interactive systems at that time and propose an agenda for further work, based on York Interactors [DH93] — objects capable of making their state perceivable to users, with specifications built compositionally. They differentiate explicitly between program verification and specification verification, and aim to address problems of reachability and reliability. Building on this, in [CH98] they argue for verification as an aid to decision making, and not just a tool for exhaustively proving properties, and introduce a tool for compiling interactors written in MAL to SMV. Going on, in [CH01], they describe the use of interactor specifications to analyse mode consequences (and look for possible mode confusions) early in the design process, using that same tool, now called `i2smv`.

In related work, in [Loe03; LH04; LH06] Loer and Harrison introduce the IFADIS (Integrated Framework for the Analysis of Dependable Interactive Systems) framework and associated tool, also compiling to SMV, but from Statemate statecharts [Har87; HN96] rather than MAL interactors. The tool includes a logic property editor, emitting LTL or CTL properties, and produces visualisations of outputs (such as model-checking traces) aimed at the designers/engineers, e.g. UML sequence diagrams, and animations.

In [CH08; CH09], Campos and Harrison introduce IVY, the successor to the `i2smv` tool, describing in detail its purpose, components, and usage, and the results of a formative user evaluation experiment, complementing [TG08] (see section 2.6) by showing how some of the analyses performed there may be encoded in the model-checking setting of IVY. IVY features *property patterns*, as introduced by Dwyer et al. in [DAC98], to aid the authoring of CTL properties, incorporating patterns from that work as well as introducing new ones.

In [Har+08], Harrison et al. consider how to extend this work into the area of user experience requirements, and the ambient/mobile context, and in [CH11], Campos and Harrison apply the IVY approach to the analysis of medical devices with a focus on comparing different devices for particular usability properties as an aid to making procurement decisions. The case study is a model of an Alaris GP infusion pump, based on its manual and on the simulation produced by Oladimeji and discussed briefly in section 4.4.

[2]`http://yices.csl.sri.com/`

2.4 State based approaches

In this section we consider, as a special case, some particularly state-based approaches, i.e. formalisms based on the classic notions of state transition diagrams/finite state machines and variants/extensions thereof, including some with extensive structuring mechanisms. Of course, many of the other works discussed in this chapter utilise state-based notations, and in particular the formalisms related to model checking conceptually unfold to state spaces, but here we focus on cases where they are the primary artifact. We do so particularly because our basic formalism is of this kind: flat graph models whose nodes represent system states and whose edges represent user or system actions.

In [AWM95], Abowd et al. consider the representation of dialogue models, which capture constraints concerning what actions are available to users at particular times in an interactive system, as finite state machines, and demonstrate the use of a model checker (SMV) to verify properties such as deadlock freedom and state inevitability.

In [DH02; HD02; DH03] Degani and Heymann describe an approach for verification of human-automation interfaces which is also outlined and exemplified in chapter 16 of [Deg04]. The approach uses state machine models, though the authors argue that the techniques are generic and could be applied to other models such as Petri-nets, temporal logics, etc. In particular, models of the task requirements, the system, and the user's mental model of it, are compared in order to verify that their combination is adequate. The verification approach is described using two examples (one abstract, one from an automatic flight control system) in [DH02]; then in [HD02] they repeat this description with a different example (a semi-automatic transmission control system for a car), and build on it with a process for producing user models that, by construction, pass the given verification checks.

In [RW06], Roscoe and Wu present a framework for the verification of statecharts based on a translation from Statemate semantics to CSP, and for subsequent analysis using the FDR tool [Sca98], with a burglar alarm example.

In [LH01], Loer and Harrison consider the problem of combining formal analysis and 'discount usability inspection', in particular the question of how to formally support heuristic evaluation [Nie93] and how to analyse for specific usability properties via exhaustive analysis, using Statecharts and the SMV model checker. The combination of formal methods with the very 'soft' and informal approach of heuristic analysis is novel and, we argue, an interesting and largely unexplored space; in section 6.4 we show how theorem discovery can automatically provide insight into particular cases where heuristics are violated.

Somewhat relatedly, Bowen and Reeves [BR10] investigate the use of formal models of user interfaces as the basis for designing software evaluation studies. Their approach was to manually reverse engineer some software into UI models in the μ-charts language [Ree05] (though they would normally advocate producing such models via user-centred design), and to write a system specification in Z, and then to automatically transform the models into tests describing the conditions necessary to satisfy the behaviour given in the Z specification.

Models based on Petri-nets [Pet62], extended to deal with the particular requirements of modelling interactive systems, have been extensively explored by a team centred at Toulouse; as these models have an executable semantics, they can be used for a wide range of activities in the development life cycle, including specification, prototyping, and even final systems.

[Nav+09] is the key paper describing this formalism, which is called ICOs (*Interactive Communicating Objects*). The formalism consists of a user interface description language, a notation and a formal description techique, based on Petri-nets extended with features capturing object-structuring aspects [Bas+99] and interaction. "A cooperative object (CO) states how the object reacts to external stimuli (method calls or event notifications) according to its inner state." ICOs extend COs with a presentation part, and functions ('activation' and 'rendering') linking the two. The formalism has a well-defined semantics and extensive tool support; furthermore, the presentation part is abstracted and can, if desired, be connected to a dedicated UI description technique such as UsiXML [Lim+05]. The *PetShop* CASE tool provides modelling, prototyping and testing capabilities (as ICOs are fully executable), and ICOs may also be validated and analysed using other proof tools. There are several case studies in the paper from avionics and aerospace.

See also [Nav+01] for a description of structuring mechanisms used by the formalism, and [BNP03] which focuses on the PetShop tool, and notes that highly interactive (or 'post-WIMP') UIs are appearing in safety critical systems and that formal methods should thus be used to enhance their safety. More recent examples of work in this area include, for example, [Cho+11] in which Choitat et al. introduce mechanisms for fault tolerance and self-checking components in ICOs, and [Pal+11], in which Palanque et al. describe usability evaluation in the ICO context.

2.5 Reverse engineering of interactive systems

Reverse engineering refers to a broad collection of techniques and tools related to the process of analysing an existing subject system in order to create representations of the system at a higher level of abstraction, in order to support activities such as maintenance, testing, quality assurance, reuse and integration. It is generally seen as a two-step process: *information extraction* and *abstraction*, with two broad objectives: *redocumentation* (producing artifacts at the same level of abstraction as the system, e.g. pretty-printing source code) and *design recovery* (producing more abstract representations). The field is a major area of software engineering, with many successes over the past several decades. [CD07]

In this section, we review some key examples of the application of reverse engineering techniques to interactive systems, where the abstract representations produced will thus typically include aspects relating to user interaction, presentation, etc. Model discovery, one of the two techniques described in this thesis, is in fact a reverse engineering technique whereby a running system is *dynamically analysed* in order to produce a model of its interaction behaviour — in the terminology just introduced, it is an information extraction technique for design recovery. As such we consider some related examples of such an approach. We also examine some examples of *static analysis*, where rather than running the system in order to analyse its behaviour, its source code (possibly accompanied with other artifacts) is examined in order to produce some more abstract model.

Dynamic analysis techniques

In [ABL02], Ammons et al. describe a "specification miner" which turns a set of traces of program API calls into a probabilistic regular grammar or state machine modelling the observed system. The specifications thus produced cover temporal and data-dependence properties, and can be used to find 'rare bugs' using a tool which examines all program paths such as a model checker or a program analysis tool such as xgcc[3]. While GUIs are not considered specifically, the probabilistic aspect is interesting: in this thesis we suggest conditional exploration as a strategy for dealing with cases where complete exploration of a state space is not possible — but an approach modelling expected or observed user behaviour probabilistically might be fruitful. The approach described in this paper may also be seen as somewhat dual to theorem discovery as described in this thesis: they take program traces and (probabilistically) infer models, whereas theorem discovery takes a model and analyses the effects of action sequences, i.e. traces.

In [CLM03], Chan et al. describe a dynamic analysis technique for event-driven interactive systems, based on identifying distinct execution bursts corresponding to actions performed by the user, in which they then describe those actions visually using a relevant fragment of the application's display at the time the action is performed; the aim is design recovery, though the focus of the paper is on the information extraction stage. They present a case study on the LyX[4] WYSIWYG editor for LaTeX, under the X Window System, and show how (thousands of) actions are collected automatically in the background as a user interacts with the system.

In [MBN03], Memon et al. describe *GUI Ripping*, a technique which dynamically constructs a model of a running GUI (particularly hierarchical GUIs of desktop programs) in order to aid test case creation. A GUI's state is modelled as sets of widgets, properties and values (a structured specialisation of the very generic notion of state employed by model discovery — see section 3.2.1); *event flow graphs* model event paths (but with events on nodes, not edges as in the state spaces produced by model discovery and analysed by theorem discovery). The motivation is model-based GUI testing, and there is no consideration of the underlying/inner state of the system and its relation with the GUI. A depth-first search algorithm is used, with Windows/C++ and Swing implementations.

In [MBD08], Mesbah et al. describe a tool which crawls rich AJAX web applications; the end product is a state space similar to that produced by model discovery, and the algorithm in fact can be seen as a special case of model discovery, with domain-specific aspects (such as handling of the browser's "back" button) tightly integrated.

In [PFM08; Pai+05], Paiva et al. describe the automatic generation, by dynamic analysis, of a "skeleton"

[3]http://www.cs.stevens.edu/~wbackes/xGCC/
[4]http://www.lyx.org

formal model of a GUI, which is then completed manually to produce a test oracle. The formalism used for the models is a rich pre/post specification language, *Spec#*. The GUI is explored via the operating system's window manager, so the anaysis tool and system being analysed need not be written in the same language or run together — as the motivation is reverse engineering existing systems whose source code may not be available and thus impossible to integrate with (see section 3.2 for further discussion of this issue).

In [Giv+13] Givens et al. combine computer vision techniques (to identify key graphical elements in a GUI, and implemented in a tool called *Sikuli script*) with grammatical inference (on the output resulting from a simulated user's input, implemented via an SMT solver) in order to build a finite state machine model of a system's behaviour. The focus of the paper is on producing the model, but they also discuss visualisation and analysis for potential mode confusion patterns. They provide two case studies: a scientific calculator running under Microsoft Windows, and a simulation of a commercial insulin infusion pump.

Static analysis techniques

In [SCS06] and [SCS07], Silva et al. describe static analysis of Java/Swing code based on the use of *program slicing* [Tip95] to isolate the user interface parts of a program from its functional core via AST (abstract syntax tree) traversal, in order to produce abstract user interface models. The approach, intended as an adjunct to the IVY tool described in section 2.3, targets single-window hierarchical WIMP user interfaces, and in particular cannot handle UIs with synchronisation, timing or continuous aspects. The AST traversal looks for GUI-related code such as window and widget definitions, and in particular for four key abstractions related to the user: user input, user selection, user action and output to user. From this, several kinds of model are produced: MAL interactors; event-flow graphs describing the construction of the GUI (in terms of widget parent/child relationships, for example); and finite state machine models describing the dynamic behaviour of the GUI, in terms of the state chaging effect of user actions on the system.

In [SSC09], they build on this work, describing the ongoing development of the framework — now called GUISurfer — and its application to software testing. Here the framework is able to handle multi-window systems, and has been extended/genericised to also deal with wxHaskell[5] GUIs (see also [CH07], which applies program slicing to wxHaskell). With regard to testing, the models produced by the framework are used to produce and run large numbers of automatically generated test case instances (i.e. strings of user input) using the *QuickCheck* testing tool [CH00], and this feature is demonstrated with a test case checking that only certain windows can be opened. In [SCS10], [Sil10] and [Sil+10] they continue the description of the development of the framework, including the application of graph-based analytical techniques such as shortest distance, pagerank, etc. as in [TG08] (see section 2.6).

In [Sta07a] and [Sta07b], Staiger describes a similar approach to static analysis of GUIs, with a focus on generic detection of GUI elements and their relationships, whose output is a directed graph of windows, where edges represent actions in the source node window that cause the destination node window to be created or shown. The context is C/C++ GUIs implemented in the GTK and Qt frameworks, which raises some particular challenges such as pointer analysis; the work is an extension to a tool suite called *Bauhaus*, which already contains a number of features useful in solving this problem.

In [LW08], Li and Wohlstadter consider GUI modelling combining both static and dynamic analysis, using the aspect-oriented Java programming extension *AspectJ* and thus targetting Java interfaces only. Their focus is on clean software engineering rather than HCI/usability concerns, however, and the models extracted focus on static structure rather than system behaviour.

2.6 Lightweight formal methods for dependable HCI

The two techniques presented in this thesis are examples of 'lightweight' formal methods targetting HCI and usability concerns, i.e. they are formally-based approaches to interaction programming, with a lower barrier to entry than traditional formal methods such as model checking and theorem proving, and they can conceivably (and indeed by design) be implemented and used by any competent programmer without any specialist training. In this section we consider some earlier examples of such approaches to HCI and in particular to making interactive systems more *dependable*.

In [Thi97], Harold Thimbleby points out various problems with popular desktop calculators and recom-

[5]`http://www.haskell.org/haskellwiki/WxHaskell`

mends a declarative approach to the operation of a calculator; in [Thi00] he continues this argument that calculators are needlessly bad, and identifies particular problems of inadequate documentation, bad implementation, feature interaction, and feature incoherence; then, in [Thi04b] Will Thimbleby describes a novel 'interactive whiteboard' style calculator with a highly responsive design and declarative operation, and gesture recognition. In sections 4.5 and 6.5 we apply our techniques to a typical desktop calculator, and describe a previously unnoticed problem there.

In [Mar+02], Marsden et al. advocate the application of what might be termed 'basic Computer Science' principles to the design and analysis of interfaces, an approach exemplified in two proposed redesigns of a mobile phone's menu system, using a balanced binary tree and a hash table respectively. They argue that such an approach leads to more consistent behaviour, greater efficiency, and an overall improved experience for the user.

In [Thi07b], Thimbleby argues that while traditional methods of HCI and user-centred design are essential, they are insufficient to meet the needs of safety critical interactive system development, in particular for medical devices; the paper outlines a method based on simulation (with the example of a Fluke 114 multi-meter) using the Mathematica[6] language/tool, and the computation of the simulation's state space in an early example of model discovery. The approach includes visualisation and formal evaluation of the design, including an analysis which can be seen as a prototype form of theorem discovery. The paper advocates interactive exploration of this kind as part of the process of device development.

In [TG08], Thimbleby and Gow show how to apply graph theoretical methods to the analysis of models of interactive devices (the case study is a syringe pump, the Graseby 9500), giving a number of quantitative and qualitative usability measures. The approach is motivated by the desire to have "a very clear notion of what the device model is", as opposed to an abstract one or an incomplete one, as many formal approaches to HCI provide. The analysis includes standard graph-theoretical properties such as reachability, diameter and radius, and completeness, as well as showing how to relate usability-specific notions such as undo cost and observability to such models.

In [Thi09], Thimbleby advocates the use of formal approaches to UI design in safety critical interactive systems, and in particular introduces some early examples of the use of the technique of model discovery

[6]`http://www.wolfram.com/mathematica/`

which is a major topic of this thesis — including the Casio HS-8V calculator considered in more detail in section 4.5 and a prototype of a new design for a drug dosage calculator — and describes the use of (custom) model checking in such cases. [Thi10] is similar, with more of an emphasis on the identification of *latent* error conditions. In [TO09], Thimbleby and Oladimeji consider the application of Social Network Analsysis (SNA) techniques to these kinds of models. SNA is a group of graph-theoretical analytical techniques, such as the *Sabadussi*, *Jordan* and *betweenness* measures of state centrality. While all three of these papers discuss models that have been produced using early *ad hoc* implementations of model discovery, they do not explain the technique. In [GT10], Gimblett and Thimbleby describe model discovery in detail (and formally) for the first time; that paper forms the basis of chapter 3 and section 4.2 of this thesis.

In [TC10], Thimbleby and Cairns consider a particular class of error, namely number entry errors, and argue that while typing numbers is such a mundane task that it seems not to merit 'a second glance', this is in fact a large unsolved problem area, affecting a range of types of system including safety critical systems and medical systems in particular, and that some 'simple' solutions could have a large and beneficial impact. The paper includes a quantitative (Monte Carlo) analysis on the impact of a design change on 'out by 10' errors in an example system, and presents a number of problematic real world examples.

In [OTC11] and [Ola12], Oladimeji et al. continue this work on number entry; they introduce a taxonomy of number entry systems commonly found on interactive systems and in particular on medical devices such as infusion pumps, and describe some user experiments comparing such systems in terms of users' abilities to recognise and recover from error when it occurs, leading to concrete recommendations for manufacturers in terms of interface style (from a safety point of view).

Similarly, in [Thi+12], Thimbleby et al. look closely at one particular style of number entry system, referred to as the '5-key' style, and describe automatic experiments that explore the space of possible behaviours of such systems, again leading to concrete recommendations as to their design for error recoverability. This class of systems is the subject of one of the case studies of model discovery and theorem discovery in this thesis (sections 4.3 and 6.4).

One of the recurrent themes of the four papers just described is that ignoring errors (i.e. not forcing users to note and respond to them) is a problematic design choice. In [Thi12], Thimbleby argues this point in particular and in detail: the essential argument is that user errors are impossible to eliminate completely [Rea00] and as such it is critical to respond to them properly, by noticing when they happen and allowing (indeed supporting) recovery from them.

2.7 Human factors and HCI in medical systems

We conclude our survey of background and related work by considering some applications of HCI principles and human factors engineering to a particular kind of safety-critical system: medical devices. One of the case studies in this thesis concerns number entry on medical devices, and another concerns desktop calculators which, given their widespread use in hospitals for drug dosage calculations, may also be seen as safety-critical medical devices. The idea that HCI and human factors have an important contribution to make to the design of medical devices is gaining acceptance, but uptake remains slow. Here we consider some examples of interesting work in this area.

In [Cha+92], Charante et al. provide an early example of work noting 'classic' HCI flaws such as lack of feedback in a device used in heart surgery, with clear evidence that the flaws do "increase the potential for erroneous actions", with a clear contribution to potential for 'actual incidents'.

In [Lin+98; LVD01] Lin et al. look at the application of human factors research to the design of a medical device for patient-controlled analgesia. They evaluate the user interface of an existing pump, using cognitive task analysis and field observations, and redesign the device interface (in a computer prototype) for greater efficiency, better feedback, better recovery from error, etc.; a small experiment validated their improved design.

In [Dea+02], Dean et al. describe a single study in the UK focussing on prescribing errors. To pick out some key figures: 36,200 medication orders over 4 weeks were examined; a prescribing error was identified in 1.5% of cases, potentially serious in 0.4%; 54% of errors were associated with choice of dose. The reasons behind the errors are not unpacked; it would be interesting to know how much of a contribution number entry errors made.

In [Vic+03], Vicente et al. describe a single case where a patient died from an overdose of morphine, apparently as a result of the use of a higher concentration 'cassette' than prescribed, without a corresponding modification of how the PCA (patient-controlled analgesia) device (an Abbott Lifecare 4100) was programmed; the case is related to others, and the paper makes a number of recommendations including greater use of human factors. Interestingly, it is an example where a number being entered too *low* (the drug concentration) resulted in death — because the actual concentration used was higher. The authors note that "all reported user error deaths with this device were explicitly attributed to programming of drug concentration." The paper also makes an interesting point about data logging: the device logs 200 'events', but the device was kept in use after the death and so the data was lost.

In [KNP03], Kaye et al. describe a tool & structured repository for dealing with and cataloguing medical device use problems, based on a model of medical device usability properties called UPCARE (not to be confused with the CARE properties of multimodal user interfaces used by [KSH08], section 2.3). The domains of the model, arising from a programme of interviewing medical professionals (mainly nurses) about device issues, are: unmet user needs; perception; cognition; actions; results (allowing adverse events to be recorded via the model); and evaluation (allowing for various evaluation strategies when applying the model). Each area is further divided into components highlighting particular commonly occurring issues. The model is posited as an aid to communication among professionals analysing and comparing devices, with the intention of creating a shared repository of device analyses for that purpose.

In [Zha+03], Zhang et al. apply the heuristic evaluation technique [Nie93] to medical devices, with the aims of discovering usability problems that could lead to medical errors, and supporting purchasers in the comparison of patient safety features across different devices, and supporting manufacturers to improve patient safety features in their design process. They use 14 heuristics, taken from [Nie93] and [SP04], and also add a 'severity rating' dimension to their analysis. Four analysts applied the technique to two similar devices (infusion pumps), and found 192 hueristic violations for the first pump, and 121 for the second; as the first pump also had more violations of high severity, the authors conclude that overall it has worse usability.

In [RJJ09], Ray et al. discuss two 'complementary' techniques for developing dependable medical systems: model based development and static analysis. The introduction gives a good motivation for formal methods over traditional software development techniques in this context. Model-based development is described from the points of view of model-checking and IBV (instrumentation-based verification). Static analysis is proposed for checking architectural properties and detecting low-level runtime errors such as buffer overruns and memory leaks. The paper then discusses how regulatory frameworks (such as those for which the FDA, which employs two of the authors, is responsible) might incorporate awareness of these methods.

That paper cites the related 4-page note [JJ07], in which Jetley and Jones present an overview of a hazard analysis of a generic infusion pump (GIP), and discuss its application to one particular model of infusion pump's GUI code (the pump is unnamed, but the code is 20,000 lines of C++), testing assertions with the *Verisoft* model checking and state space exploration tool produced by Bell Labs.

In [ATO10], Acharya et al. make the case that HCI is under-applied to healthcare, and look at a computerised hospital bed as a case study, modelling its interaction behaviour as a state transition diagram.

In [WCB13], Wiseman et al. describe and discuss the results of analysing 58 log files from 32 infusion pumps, in order to determine the digit distributions in numbers entered for variables such as VTBI (Volume To Be Infused), rate, etc. The main points are that the distribution is roughly according to Benford's Law [Ben38] but 0, 5, and 9 are all more common than predicted by that law. The explanations are that 0 is used for large round numbers, 5 is used for "in between" values (this follows known bias in how doctors record blood pressure readings), and 9 is used in surgery where an indeterminate amount of infusion is required and the device is constantly monitored, so the maximal value (typically all 9s) is chosen. There is also some discussion of number length distribution, implications for design, etc., and a study of three common number entry interfaces (including the 5-key style examined in sections 4.3 and 6.4) in the light of that information, in particular noting that designs are not optimised for the kinds of numbers most frequently entered.

Chapter 3

Model Discovery

Contents

3.1 Introduction

In this chapter we present the first of the two techniques at the core of this thesis: **Model Discovery**. Using this technique, a model of a user interface can be *automatically* discovered by a tool simulating the actions of a user. Here we provide a detailed, formal and generic description of the technique, which should be sufficient to allow any competent programmer to implement model discovery for themselves, provided their UI framework meets the requirements we set out. In particular:

- We informally describe the requirements that must be met by the implementation context in order to allow model discovery to even be possible (section 3.2).
- We formalise those requirements in an API for UI model discovery; the implementation of this API is a prerequisite for performing model discovery in a given setting (section 3.3).

- We describe a generic model discovery algorithm in terms of that API (section 3.4).
- We consider variants and extensions to the API, with which the algorithm can perform a variety of sophisticated UI exploration tasks (section 3.4.3).
- In chapter 4, we consider four examples of model discovery in use: two detailed examples by the author, and two examples from other sources, where we briefly draw out some interesting aspects of the application of the model discovery technique.

The description given here is language-neutral, and suitable for retrospective integration with existing UI applications and development tools; as such, we hope and believe that this will facilitate the application of model discovery in diverse settings. In particular, integration into IDEs and development workflows seems a promising area, and one where the requirements of the API (discussed below) are easily met; reverse engineering existing systems could also be valuable, but harder to achieve in general because of the deep level of integration required.

In model discovery we systematically explore the state space of an interactive system by simulating the actions of a user, using standard search techniques [Win92] augmented with domain-specific aspects such as discovery/actuation of UI widgets. Model discovery produces a finite directed graph whose nodes represent (sets of) states of the target system, and whose edges represent user actions or events that change the system state. A path through the graph is thus a path through possible system states, following a sequence of possible user (and possibly internal) actions.

While such flat models can become very large, experience has shown that by choosing the right level of abstraction, tractable models yielding useful results are obtainable; Jackson's *small model hypothesis* [Jac06; ADK03] suggests that bugs can be found in small models, and if a large model is required, a user would plausibly be unable to understand the operation of that system in any case. Various analyses of such a model are possible, using standard graph-theoretical techniques or model checking tools [Thi07a; Thi09; TO09], not to mention the other major topic of this thesis, *Theorem Discovery* (chapter 5).

Model discovery complements purely analytical techniques such as abstract interpretation [CC77]; in the presence of running code it is much simpler to apply, and does not rely on detailed knowledge of the code's running environment. For example, consider an interface running in a web browser: its event-handling/buffering behaviour, and their interaction with the code in question, will tend to be non-trivial to interpret abstractly; by using model discovery we can automatically produce a model that really represents some aspect of the behaviour of the actual running system.

3.1.1 Scope

Our current scope is reactive devices with discrete interfaces and finite state spaces, subject to a number of assumptions. In particular:

1. That the system responds (almost) immediately to user actions. This is primarily a matter of practicality rather than absolute necessity: if the system responds slowly or needs time to reach a steady state, model discovery might still be possible, but would require the introduction of forced delays to allow the system to settle, which would slow the process down considerably. Typically, a large number of states and actions need to be explored, so such delays would tend to be tedious in practice.

2. That the system's responses to user actions are manifested in the system's state (possibly, but not necessarily only, including in the state of its UI). What exactly we mean by *the system's state*, and how it is to be dealt with meaningfully by the model discovery process, is considered in section 3.2.1.

3. That any silent or external (non-user) actions that modify the state may be modelled by (perhaps explicitly) adding them to the set of explored actions — see section 3.2.2. For example, in the model of the Alaris GP infusion pump discussed in section 4.4, *tick* actions were used in certain modes to represent the passage of time.

There are clearly many non-trivial and interesting devices that satisfy these assumptions, including and in particular those considered in chapter 4.

We take the view that such an interactive system consists of an interface with which the user interacts, and (usually) an underlying system implementing domain-specific logic. This separation of concerns may be designed into the architecture (e.g., using the MVC [Ree79] or PAC [Cou87a; Cou87b] design patterns), or it may be less well-defined. We call the state of the interactive part the *GUI state* — in principle visible to the user — and we call the state of the underlying system the *inner state*. Model discovery may involve probing and even modifying (while backtracking) both kinds of state; doing so for a system whose state is a complex object graph is conceptually no different from doing so for a system whose state is a simple bundle of values (such as the example in the next section): it might be a bit more complicated, but it's essentially the same task, provided the requirements set out in section 3.2 are met.

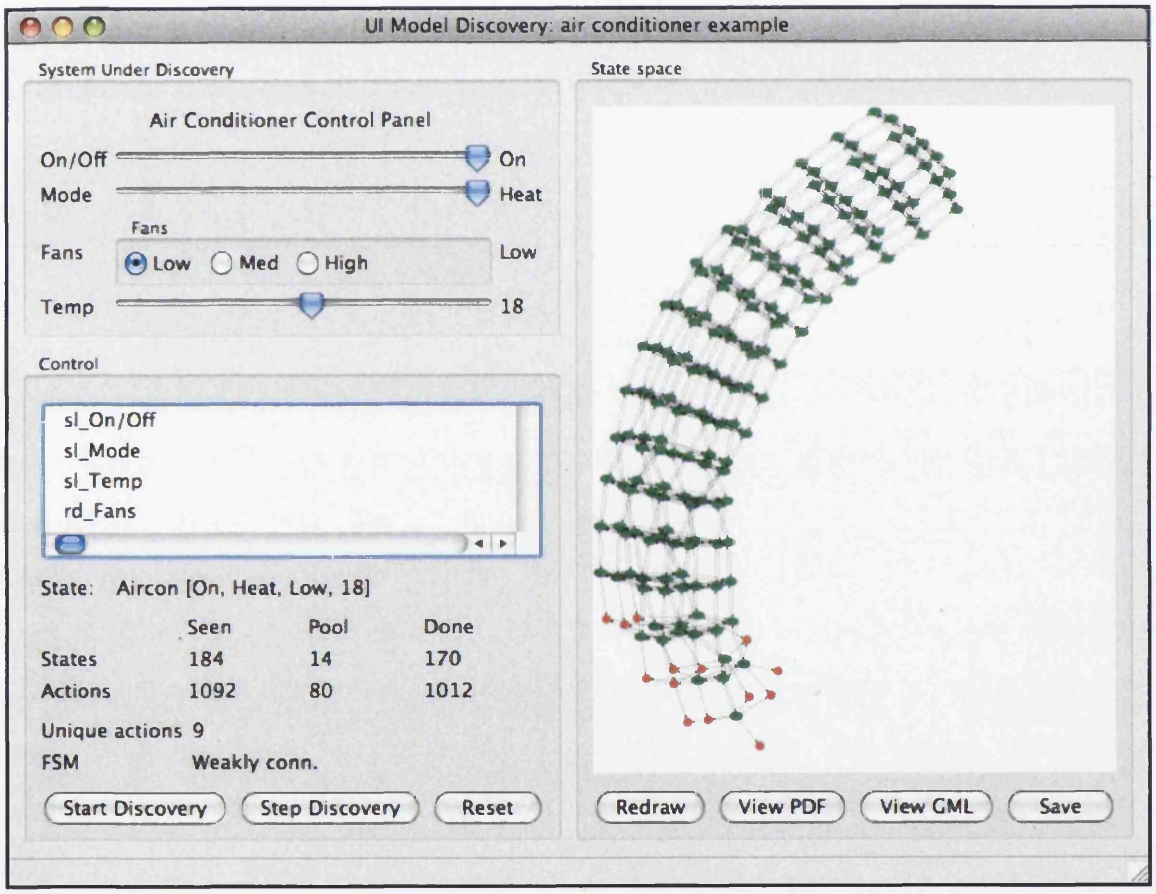
3.1.2 Illustrative Example

In order to further motivate and concretise our discussion, let us briefly consider an example of model discovery in action; we return to this example in more detail in section 4.2. The example involves model discovery on the control panel for an air conditioning unit — see the screenshot in figure 3.1. Here we see a number of features typical of our discovery approach.

The interface of the system being discovered, or *System Under Discovery* (SUD), can be seen in the top-left portion of the window. The rest of the window contains controls and feedback for the model discovery process itself (below the SUD section) and a graphical preview of the discovered model (on the right). The SUD was originally developed as a standalone application, the body of which we have embedded directly into the model discovery tool for the sake of simplicity and ease of exposition. While convenient, in general such a direct embedding is not necessary (indeed it may not even be practical, e.g. if the SUD has more than window); whether the SUD is embedded or 'free floating' is unimportant, provided the model discovery code can in some sense access the SUD's code and state: see section 3.2 for further discussion of this point.

The SUD is a simulation of the control panel of an air conditioning control unit, with the following controls:

- on/off;
- heat/cool;
- fan speed (low, medium, high);
- target temperature (5–30°C, i.e. 26 settings).



Fans: [✓ Low] Med High — Low

Temp: 18

State: Aircon [On, Heat, Low, 18]

	Seen	Pool	Done
States	184	14	170
Actions	1092	80	1012

Unique actions 9

FSM Weakly conn.

Figure 3.1: UI model discovery for a simple air conditioning control unit

The *Control* section is used to control and monitor the model discovery process. First, the SUD's controls are listed: three sliders `sl_On/Off`, `sl_Mode`, `sl_Temp`, and one radio control `rd_Fans`. These controls and their associated actions (which are not displayed in this example) have been discovered automatically by the model discovery tool — and may in general vary between states of the SUD. In this case, the set of available controls never changes, but the available actions for each control *do* change: e.g. if a slider is in its minimum position, the 'slider down' action is unavailable. We consider the task of computing available controls and actions in section 3.2.2.

The *State* part shows a term representation of the SUD's state: a 4-tuple projection of the current values of the four controls. In the case of this SUD, the state we are interested in, and which we project, is exactly the GUI's visible state — though this need not always be the case: we describe this important aspect of model discovery in section 3.2.1.

Beneath the *State* part are a number of counters indicating the progress so far of the discovery process, and a "health check" of the discovered model. The basic task of model discovery is to identify states and the possible user actions performable in those states, and then execute each of those actions in each of those states, and see where that takes you. In this screenshot, model discovery has found 184 states so far, 14 of which still have unexplored actions (and so remain in the *pool* of unexplored state/action pairs), and 170 of which have been fully explored; similarly, model discovery has found 1092 actions (properly, 1092 state/action pairs), of which 1012 have been explored and 80 remain in the pool, waiting to be explored. Thus we may infer that the graphical view of the model on the right of the screenshot contains 170 green (i.e. fully explored) states, and 14 red ones, and 1012 edges. (Given the possible values of each control in the SUD, and the fact that they are entirely orthogonal, the total state space to be discovered consists of $2 \times 2 \times 3 \times 26 = 312$ states.)

To be completely clear: each node in that graph, i.e. each of those 180 states which have so far been discovered, represents a unique state of the SUD, i.e. a unique combination of the values of the four SUD controls listed above; furthermore, each edge represents a user action that changes the state. For example, consider figure 3.2. Here we see a much 'younger' model of the aircon control panel. The starting point for this exploration was the state represented by the green node:

$$(Off, Heat, High, 5)$$

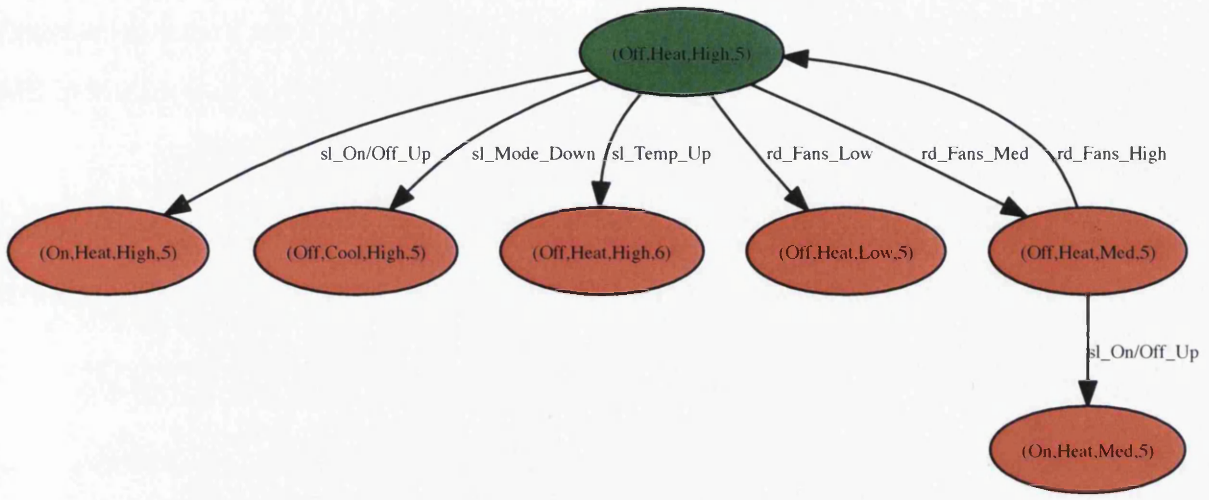


Figure 3.2: A model of the air conditioning control panel, very early in the discovery process: only the first state has been fully explored.

and we have five edges leading from that node, representing the five actions possible in the SUD when it is in that state. In particular, we have an an edge labelled `sl_On/Off_Up` (representing moving the on/off slider up), leading to the state:

$$(On, Heat, High, 5)$$

In this example, all 5 actions possible in that initial state have been explored (which is why it has been coloured green), leading to 5 new states, one of which, (*Off*, *Heat*, *Med*, *5*), is in mid-exploration, with two of its five actions explored (one of which happens to lead back to the initial state).

A graph of this nature is the basic product of model discovery — the model it produces.

Returning to the screenshot in figure 3.1, we also have a counter of actions found so far, of which there are nine in this example: ‘up’ and ‘down’ for each of the three sliders, plus three possible radio button states; and a health check, telling us that the graph is currently weakly connected — upon completion of this particular discovery task, the graph will be strongly connected (though that need not be the case in general).

Finally, there are buttons to start discovery (which becomes “Pause Discovery” if it is running), to execute just a single cycle of the discovery algorithm, and to reset to an initial state.

The *State space* panel to the right previews the model as discovered so far, as rendered using the

GraphViz tool [Ell+02]. Below this are buttons to redraw the preview (if the window is resized), to view the state space in full using an external application (rendering to PDF via GraphViz, or to GML, another graph language), or to save it in various formats suitable for further processing.

We return to this example in section 4.2, where we describe its implementation in more depth, and show more screenshots of the discovery process in action; in section 6.3 we apply theorem discovery to the model thus produced.

3.2 Requirements for User Interface Model Discovery

The UI model discovery algorithm simulates the actions of a user systematically performing all possible actions in all possible states. The API is the formal interface between the UI application and the discovery tool — a bridge that must be implemented before discovery can take place on a given platform. The API provides four key capabilities:

- the ability to compute, store and compare SUD states;
- the ability to identify, for a given SUD state, the actions that may be performed in that state;
- the ability to perform such actions;
- the ability to restore SUD state, i.e. to *backtrack*, so that all of a state's actions can be explored.

Before considering these requirements in more detail, a few high level remarks are in order about the question of *process separation* as regards model discovery. In general, we would expect the SUD to be capable of running as a standalone system — and indeed, in each of our case studies, that is the case. The model discovery API, however, requires a certain degree of automated access to the SUD — including, in general, to aspects of its internal state. There are two ways this can be achieved.

The simplest and most powerful approach is to embed the SUD within the model discovery tool in some way. This doesn't necessary imply the GUI-level embedding seen in figure 3.1: just that the SUD and the model discovery tool are running in the same process (or collection of processes, or sandbox. . .) — for example they have been compiled and linked together into a unified executable. The key is that, from the model discovery code's point of view, the requisite parts of the SUD are in scope at compile-time (if that exists) and available for access at run-time.

This is exactly the approach taken in all four of our case studies; for example:

1. The air conditioner control panel (section 4.2) is a compiled Haskell executable; here the SUD *can* be compiled as a standalone application, but for model discovery it has been embedded within the model discovery tool, and they are compiled/linked together.

2. The '5-key' number entry system (section 4.3) is an HTML/JavaScript client-side web application; again, the SUD can be run on its own, but for model discovery it has been embedded within the greater context of the model discovery tool, and interpreted within the process space of a single page in a web browser.

Sometimes, however, such a close relationship might not be possible — for example, consider a reverse engineering task where we wish to perform model discovery on an existing tool for which the source code is unavailable. In such a case, it might still be possible to perform model discovery, but there will need to be some way of remotely (i.e. from another process, at least — and conceivably from another computer) interacting with and probing the SUD from the model discovery tool. For example, both [MBN03] and [PFM08] describe approaches where the GUI structure of a running Windows application is dynamically learnt at runtime from a separate process, via the intermediation of the Windows API.

Even if such an approach is feasible for the task of identifying and interacting with the SUD's GUI components, there will still in general be limits to what state can be projected — and in practice, probably only those parts explicitly exposed via the GUI will be accessible in such cases. For this reason, the more tightly-coupled approach described above is recommended where possible; in our experience, the SUD usually needs little or no modification in order to expose whatever aspect of its state is of interest. We envisage model discovery primarily as a component in the software development process, where its integration into IDEs (for example) would tend to make these issues if not trivial, then certainly tractable.

3.2.1 Identifying and modelling SUD state

A node in the graph of the discovered model represents some projection of the state of the SUD, and by definition, each node represents a *different* value for this projected state. This notion of projected state is deliberately left abstract and generic in order to support a wide range of possible implementations: it is the task of the interaction programmer modelling the SUD to choose and implement an appropriate projection. However, this task is critically important (e.g., see [Dix91] for an in-depth examination of questions surrounding such projections) and so we consider some key factors here. The basic points are:

1. The SUD has state, which conceptually consists of a collection of key/value pairs (i.e. names of state items, and their corresponding values), though in practice such state may be highly structured, e.g. a collection of objects.
2. The model discovery algorithm repeatedly projects a 'snapshot' of the current SUD state, tracking how it changes in response to the UI actions performed.
3. These projections/snapshots need not (and in general *will not*) be complete copies of the SUD's entire state, but will usually consist only of particular interesting or necessary aspects of the state.
4. The model discovery algorithm uses the projected state in order to *backtrack*, so that the effect, at a given state, of more than one action may be explored. As such, the projected state must be chosen carefully, and must include enough state to allow backtracking to be *sound* (see section 3.2.3).

To expand on the third point, there are two reasons why the projection need not be complete. First and foremost, the purpose of model discovery is to generate a model of the *interaction behaviour* of the system, i.e. in order to understand something about how it behaves in reaction to user input. As such it is necessary to operate at an appropriate level of abstraction. For example, consider once again the air conditioning control system discussed previously. An actual instance of such a system would probably track the actual temperature in the room (or maybe across several rooms), and perhaps also humidity, and the time, say; to project all of these aspects during model discovery would be to attempt to create a model of all possible states of the system, not only in terms of its responses to user actions, but also its responses to changes in the environment. This removes the focus of the model from the system's interaction behaviour, and also makes the model considerably (perhaps infinitely!) larger.

Second, even while focused on the interaction behaviour of the system, there can be good reasons to focus on particular aspects of its state, at the expense of others. For example, in section 4.5 we consider model discovery applied to a desktop calculator, but focused on one particular aspect of interest, namely number entry; as such, in that example we ignore (i.e. do not project) those aspects of its state related to memory or arithmetic operations. In general, tracking 'uninteresting' state will introduce many 'garbage' states to the model, and should be avoided if possible — but note that that may *not* be possible, at least at the discovery stage: see section 3.4.2. Even ephemeral parts of the state structure (i.e. ones which only arise/have content at certain times) may be projected: two states are clearly non-equivalent if they contain differently-named elements — though care might be required for backtracking, to ensure that an ephemeral piece of state has a home to be projected back onto. (Again, the techniques described in section 3.4.2 can help here.)

We noted earlier that the system's state may be divided into *GUI state* (in principle visible to the user) and *inner state* (everything else). In practice, this distinction is not of great importance, for several reasons. First, there will tend to be overlap, e.g. if the system has been developed following an MVC design, its model (inner state) will be reflected in its view (GUI state) at least to some extent. Furthermore, it is sometimes simply necessary to project some aspects of inner state in order to get the desired model (see section 4.3.3.4, for example). Thus, while a valid ontological distinction, in practice for model discovery, we are potentially interested in both kinds of state, and so distinguishing strictly between them is unnecessary. As noted above, there may be cases where model discovery can *only* probe GUI state, but we consider these to be a special case.

Another important question is: which values are to be projected for each projected variable? Projection of all possible values may not be tenable; e.g. if we project a numerical variable directly, the state space can rapidly explode. It may be sufficient in such cases to use a small subset of possible values (as in section 4.3), or a set of named equivalence classes (e.g. the classic "zero, one or many"). Conversely, too much restriction here can result in an insufficiently detailed model — though in our experience a surprising amount can be learnt from very restricted models [Thi09].

These two questions — what variables to project, and what values to project of those variables — interact to define an equivalence relation on SUD states. Again, it is a question of choosing an appropriate abstraction for the modelling/analysis task at hand, and we suggest that this is one of the key activities of the investigating programmer when performing model discovery.

A final question regarding state projection is: how is the state to be represented? Like an object's state in a traditional OO language, a state can be viewed as a mapping from names to values of various types; because we need to compare states for equality, we must be able to compare their values. As a general approach, we have found data types with trivial semantics similar to that of JSON [Cro06] (i.e. trees of key/value pairs with basic data types such as numbers, bools, strings, lists) to be adequate. As noted above, richer models might be used, but our experience so far has been that the state structuring and abstraction mechanisms which aid implementation of UIs are not relevant or required in the setting of discovery.

3.2.2 Identifying and performing actions

Edges in the model represent discrete user actions that, assuming an event-driven GUI framework is used, generally correspond to events in that framework, such as button clicks, slider moves, text entry, menu selections, etc. Clearly we require a way to identify the actions that may be performed in a given state; the most general solution is to automatically inspect the SUD's GUI and discover the possible actions automatically — with integration implications as above. How this is implemented will depend on the implementation platform and the capabilities it offers.

Restricted cases, such as learning the actions only once at the start of discovery, or hand-coding them into the discovery implementation if automatic learning is not possible, are then special cases of this approach — see section 3.4.3.4.

Having discovered possible actions, it is necessary to perform them and see where they lead; thus, we require an automatic way to trigger discovered actions which, again, in detail will vary between implementation platforms. Fortunately, every framework for which we have implemented discovery has supported this quite directly; in HTML/JavaScript, for example, a button click may be enacted simply by calling the button object's `click()` method. Obviously, to do so generally requires maintaining a reference to the widget being interacted with.

There may be more than one way to handle a given widget's actions. For example, for a slider, the actions might be to move the slider up or down incrementally, or they might be to set the slider to particular values; which is used will depend on the capabilities offered by the GUI framework, and by the modelling task. Some kinds of widget, such as text entry, cannot be automatically fully explored, and will require some sort of scripting (see section 3.4.3.7).

After the action has been triggered, an SUD state is projected and inspected. If it is unchanged, the action had no effect — it is a self-loop. (Self-loops need not be explicitly represented, though they may be semantically important during subsequent analysis, particularly in cases involving nondeterminism; see [Thi09] and section 6.3.2.) Otherwise, we have discovered a new edge in the state space, and possibly a new state. If we *have* discovered a new state, we must discover its possible actions as described above, and explore them later.

3.2.3 Backtracking

To explore all actions performable in a given SUD state, it is in general necessary to be able to backtrack to that state — otherwise only one action per state may be explored. Conceivable approaches include:

1. Reset to an arbitrary saved state.
2. Reset to an initial state, then follow a saved path to the desired state.
3. Undoable actions (if they exist).
4. Using only actual user actions to restore an arbitrary state.

In our experiments so far we have exclusively used strategy 1, and the algorithm (and our discussion) assumes this to be the case.

After backtracking, the restored SUD state must be equivalent to the one seen earlier. This does not imply that the actual underlying state must be identical — only that the projected state is *and* that the effects of possible actions are identical; if they're not, then our model will be *unsound* — discovering it again in a different order could lead to a different model — see section 3.4.2.

3.3 Model Discovery API

We now describe, using Haskell data types and type signatures, the API elements that must be implemented in order to use the model discovery algorithm on a given UI development platform; familiarity with Haskell is not necessary to understand this section: we shall explain the few required concepts.

3.3.1 Use of Haskell

We specify our API using Haskell [Pey03; Hud+07] for the following reasons: it is high-level, rich but compact; it is independent of a particular SUD or GUI framework; it is easily translated to other formalisms and programming languages; finally, our specification is derived from actual running/working UI model discovery code. We would like to stress however that our goal here is a general description of model discovery, not just a particular implementation; we believe the formal description here enables its implementation in any adequate language, and in particular we have done so in several other settings including Java, JavaScript and ActionScript [Thi09; TO09].

Haskell is a pure functional language with a number of features, including the strongest and most thoroughgoing type system of any reasonably mainstream programming language. Collections of Haskell type signatures are comparable in appearance and expressiveness to signatures as found in the univeral algebra/algebraic specification tradition [BM04]. Haskell is both *strongly* and *statically* typed: every value in a Haskell program has a particular type, and that type is fixed at compile-time. Furthermore, functions in Haskell are *first class* (they may be passed to and from functions), and subject to the same type system as atomic values. Type names start with an upper-case letter, as in *Int* or *Set x*. Here *Set* is polymorphic: it is a *parametric type*, representing a set of something; it may be instantiated by providing a concrete parameter type, as in *Set Int* (a set of integers). Function types are written using arrows, where multiple arguments are written using multiple arrows: for example, $Int \rightarrow Int \rightarrow Int$ is the type of a function (indeed of all functions) taking two *Int*s and returning one of them. Names of values and functions start with a lower-case letter, and are assigned types using "::", as in $double :: Int \rightarrow Int$, which declares a function from *Int* to *Int*.

3.3.2 Discovery and Manipulation of GUI Controls

We start with data types representing GUI controls (widgets), their values, and the actions we can perform on them. The details will vary from framework to framework, so we leave these types loosely specified here, simply noting that they must exist and be adequate for their intended purpose.

$$\begin{aligned}&\textbf{data}\ \mathit{GuiControl} = \ldots\\ &\textbf{data}\ \mathit{GuiValue} \quad = \ldots\\ &\textbf{data}\ \mathit{GuiAction} \ \ = \ldots\end{aligned}$$

Note that in our conception, a *GuiAction* encapsulates not just the action to perform (say, "move slider up") but also a reference to the control (a *GuiControl* value) on which the action is performed. This is not shown above, but is implicit in some of the type signatures given below, i.e. those containing *GuiAction*: where this is seen, it should be remembered that there is also an implicit associated *GuiControl* available to that function.

Now we consider some functions on these data types.

$$\mathit{setControlValue} :: \mathit{GuiControl} \to \mathit{GuiValue} \to \mathit{IO}\ ()$$

Given a reference to some GUI control, and a value for that control, *setControlValue* sets that controller to that value (it is called when resetting/backtracking to some state). Note the presence of *IO* here (and in the next two type signatures). This indicates that the function may have side-effects — in this case interacting with the SUD's GUI, by modifying a control. If *IO* is absent from a function's type signature, we know that that function is *referentially transparent*: it can perform no side-effects — see, e.g., *addToPool*, below, which does not modify the pool, but rather returns a *new* pool. (Of course, a Java *addToPool* implementation would modify a pool rather than constructing a new one: the point is that the type signatures given here distinguish clearly between functions that interact with the SUD, and those that do not.)

$$\mathit{getGuiActions} :: \mathit{Parent}\ w \Rightarrow w \to \mathit{IO}\ [\mathit{GuiAction}]$$

Given some container widget *w*, *getGuiActions* discovers a GUI's controls, and the currently available actions on those controls, returning a list of *GuiAction*s (see section 3.2.2). Here *w* is another parametric type, and *Parent* $w \Rightarrow$ is a *context* requiring that whatever w is, it is an *instance* of the *typeclass Parent*. Haskell typeclasses are analogous to Java interfaces: here, *Parent* $w \Rightarrow$ indicates that whatever w is (panel, window, set of windows) it must have children — and in particular, a *children* function for listing them; then, *getGuiActions* uses that function to recursively inspect a widget and its children for manipulable controls and their actions.

$$doGuiAction :: GuiAction \rightarrow IO\ ()$$

Given some *GuiAction*, the function *doGuiAction* performs that action (pressing a button, moving a slider, etc.)

3.3.3 The Pool: Discovered But Unexplored States

Some mechanism is required to track those parts of the state space that have been discovered but not yet fully explored: we call this the "pool" of states and actions yet to explore. Note that there are many possible ways to explore the state space, with depth-first and breadth-first as extreme choices. We aim to be as generic as possible in our description; in particular, we wish to accommodate strategies — such as depth-first — that do not fully explore a given state's actions before moving on to another state, as well as those — like breadth-first — that do.

A pool or priority queue of (state, action) pairs is sufficient; it contains one element for every unexplored action from every discovered state. Discovery implementations may then pick pairs from that pool using whatever strategy they see fit. There are many ways to implement such a pool (see section 4.2.2.4 for examples), so again we leave it specified loosely here — though we note that it depends on the type of the SUD state, which we write as *st*.

$$\textbf{data}\ Pool\ st = \ldots$$

So a value of type *Pool st* is a collection of $(st, GuiAction)$ pairs. However it is implemented, we need to add/remove items:

$$addToPool :: Pool\ st \rightarrow st \rightarrow [GuiAction] \rightarrow Pool\ st$$
$$pickFromPool :: Pool\ st \rightarrow (Maybe\ (st, GuiAction), Pool\ st)$$

addToPool takes a pool, an SUD state, and a list of actions (performable in that state) and returns a new pool with the state's actions added — one pair per action. *pickFromPool* chooses one element from the pool, returning that element and a new pool, with that element removed. *Maybe* is another parametric type, used for computations that may in some sense "fail" — in this case, "failure" occurs if the pool is empty: then, instead of a $(st, GuiAction)$ pair, it returns the value *Nothing*. See section 3.4.3 for more discussion of exploration strategies and the critical role of the pool.

3.3.4 Model Discovery

The model discovery algorithm repeatedly picks (state, action) pairs from a pool and explores them, building a state space on the fly (and sometimes extending the pool). It stops when the pool is empty. Further, it needs to be able to query and reset the SUD, and to compare SUD states. All this can be packaged in a data type, a value of which encapsulates current discovery status, and some related functions:

```
data Discovery st = Discovery {
  statespace    :: Gr st GuiAction,
  pool          :: Pool st,
  project       :: IO st,
  reset         :: st → IO (),
  eq            :: st → st → Bool,
  getGuiActions :: Parent w ⇒ w → IO [GuiAction]
}
```

This type is polymorphic over the SUD state type *st*. Here, *statespace* is a graph (*Gr*) whose nodes are labelled with SUD states (*st*) and whose edges are labelled with GUI actions (*GuiAction*) ; *pool* is as described above. *statespace* and *pool* are initialised empty and are modified as discovery progresses.

The remaining four members of *Discovery* are callback functions, specified as parameters when *Discovery* is initialised (see below). *project* computes the SUD's current SUD state, as discussed in section 3.2.1. Dually, *reset* takes an SUD state and imposes it onto the GUI using *setControlValue* for backtracking — see section 3.2.3. *eq* compares two SUD states for equivalence, and should create equivalence classes of SUD states as appropriate — see section 3.2.1. Finally, *getGuiActions*, which learns the currently available actions, is as described in section section 3.3.2.

To initialise such a value, we provide the callback functions just mentioned, and construct a *Discovery st* with empty *statespace* and *pool*; then *initDiscovery* should immediately call *project* and *getGuiActions* to compute seeds for the state space and pool.

$$\mathit{initDiscovery} :: \mathit{IO}\ \mathit{st} \rightarrow (\mathit{st} \rightarrow \mathit{IO}\ ()) \rightarrow (\mathit{st} \rightarrow \mathit{st} \rightarrow \mathit{Bool}) \rightarrow (w \rightarrow \mathit{IO}\ [\mathit{GuiAction}]) \rightarrow \mathit{Discovery}\ \mathit{st}$$

Given a state, we need to know if it has been seen before, i.e. is it in the state space (using *eq* to compare states):

$$\mathit{isStateNew} :: \mathit{Discovery}\ \mathit{st} \rightarrow \mathit{st} \rightarrow \mathit{Bool}$$

When we find new states and new edges, we add them to the state space, returning new *Discovery* values. (We do not have to check for new edges because we only add a state's actions to the pool once, when we first meet the state; see details below.)

$$\mathit{addState} :: \mathit{Discovery}\ \mathit{st} \rightarrow \mathit{st} \rightarrow \mathit{Discovery}\ \mathit{st}$$
$$\mathit{addEdge} :: \mathit{Discovery}\ \mathit{st} \rightarrow (\mathit{st}, \mathit{GuiAction}, \mathit{st}) \rightarrow \mathit{Discovery}\ \mathit{st}$$

Finally, an auxiliary function checks if the pool is empty:

$$\mathit{finished} :: \mathit{Discovery}\ \mathit{st} \rightarrow \mathit{Bool}$$

3.3.5 Model Discovery API Summary

Figure 3.3 summarises the API, pointing out (in the third column) which parts may be re-used for multiple SUDs written in the same GUI toolkit, and which parts need to be defined once per SUD. Most of the work can be done just once per toolkit: only the data type *st* and the parameters passed to *initDiscovery* ever need to be tuned for a specific SUD (but see section 3.4.3).

Notes:

1. As described in section 3.4.3, several *pickFromPool* implementations may be desirable to

Item	**Purpose**	**Re-use**
data *GuiControl*	Data type for GUI controls	toolkit
data *GuiValue*	Data type for GUI values	toolkit
data *GuiAction*	Data type for GUI actions	toolkit
setControlValue	Set a GUI control to a given value	toolkit
getGuiActions	Learn available actions in SUD's current state	toolkit
doGuiAction	Perform some GUI action on some GUI control	toolkit
data *Pool st*	Data type for pool of (state, action) pairs to explore	toolkit
addToPool	Add a discovered state and its actions to a pool	toolkit
pickFromPool	Pick the next item to explore from a pool	see note 1
data *st*	Data type for SUD states	see note 2
project	Project SUD state from SUD	SUD
reset	Given an SUD state, reset SUD to that state	SUD
eq	Compare two SUD states for equality	SUD
data *Discovery st*	Data type encapsulating state space and pool	toolkit
initDiscovery	Initialise model discovery algorithm	toolkit
isStateNew	Check if a state is new or has been seen before	toolkit
addState	Add a newly-discovered state to the state space	toolkit
addEdge	Add a newly-discovered edge to the state space	toolkit
finished	Check if the pool is empty or not	toolkit

Figure 3.3: API summary

implement various exploration strategies; however, in each case, the strategy need only be implemented once per toolkit.

2. As described in section 3.2.1, the data type *st* used to represent SUD states may be definable just once in a generic way, which can then be re-used for many SUDs implemented using the same GUI toolkit; if it is not possible or desirable to do so, then a per-SUD representation can be used. For example, in section 4.2 we consider the previously-introduced air conditioning control system example in more depth; that system is implemented using the wxHaskell[1] GUI toolkit, and in section 4.2.2.3 we describe a *GuiControl* type for tracking the state of radio buttons and sliders in that toolkit; this data type could be re-used for any wxHaskell SUD consisting of just those controls, though it would need extension to handle other widget kinds.

[1]`http://www.haskell.org/haskellwiki/WxHaskell`

3.4 Model Discovery Algorithm

Given the API, algorithm 1 shows pseudocode for the UI model discovery algorithm.

Algorithm 1 The UI model discovery algorithm.

```
initDiscovery
while $\neg$ finished do
    $s, a \leftarrow$ pickFromPool
    reset SUD to SUD state $s$
    doGuiAction $a$
    $s' \leftarrow$ project new SUD state
    if $\neg[s \; eq \; s']$ then
        if isStateNew $s'$ then
            addState $s'$ to statespace
            $a' \leftarrow$ getGuiActions for state $s'$
            addToPool $s', a'$
        end if
        addEdge $(s, a, s')$ to statespace
    end if
end while
```

In practice the algorithm need not be implemented exactly as shown here. Thus, for the air conditioning control described in section 3.1.2, a single cycle of the **while** loop is enacted by clicking "Step Discovery," while "Start Discovery" starts it cycling to completion (though it may be paused). There is no while loop directly visible in its code — but the overall strategy above is encoded faithfully.

3.4.1 Worked example

We now illustrate the algorithm by stepping through its operation on a trivial (two state) example. Consider a picture viewer program, with $\boxed{\text{next}}$ and $\boxed{\text{prev}}$ buttons to show the next and previous pictures; suppose that it does not loop, so $\boxed{\text{prev}}$ and $\boxed{\text{next}}$ are disabled on the first and last pictures respectively. Thus this UI has one piece of state, which we will call *picture*, and two possible actions, not both necessarily available at all times.

Suppose that the program has been loaded with just two pictures, which we call 1 and 2. The model discovery algorithm proceeds as follows:

1. *initDiscovery* — initialise the discovery algorithm, which includes the following steps:

   (a) Call *project* to project the initial state, $\{ \mathit{picture} \mapsto 1 \}$.

   (b) Call *getGuiActions* to discover the possible actions in that state, which is just: $\{ \boxed{\text{next}} \}$.

   (c) Initialise the state space with the initial state (figure 3.4(a)).

   (d) Initialise the pool with that state/action pair: $\mathit{pool} = \langle\, (\{ \mathit{picture} \mapsto 1 \}, \boxed{\text{next}}) \,\rangle$

2. The pool is not empty, so *finished* is false, so enter the while loop.

3. *pickFromPool* picks the first element[2] from the pool, yielding $s = \{ \mathit{picture} \mapsto 1 \}$ and $a = \boxed{\text{next}}$ (and emptying the pool).

4. Reset the SUD's state to $\{ \mathit{picture} \mapsto 1 \}$, i.e. display the first picture (no change).

5. Call *doGuiAction* $\boxed{\text{next}}$ — i.e. simulate pressing the $\boxed{\text{next}}$ button, moving to picture 2.

6. *project* the system's state again, yielding $s' = \{ \mathit{picture} \mapsto 2 \}$.

7. $s \neq s'$ and state is new (i.e. not in state space already), so add s' to the state space.

8. Call *getGuiActions* to discover the possible actions in the current state, yielding: $a' = \{ \boxed{\text{prev}} \}$.

9. Add newly discovered state/actions to pool: $\mathit{pool} = \langle\, (\{ \mathit{picture} \mapsto 2 \}, \boxed{\text{prev}}) \,\rangle$

10. Add $\boxed{\text{next}}$-labelled edge from s to s' in state space (figure 3.4(b)).

    (End of first iteration through algorithm.)

11. The pool is not empty, so *finished* is false, so stay in the while loop.

12. *pickFromPool* picks $s = \{ \mathit{picture} \mapsto 2 \}$ and $a = \boxed{\text{prev}}$ (emptying the pool again).

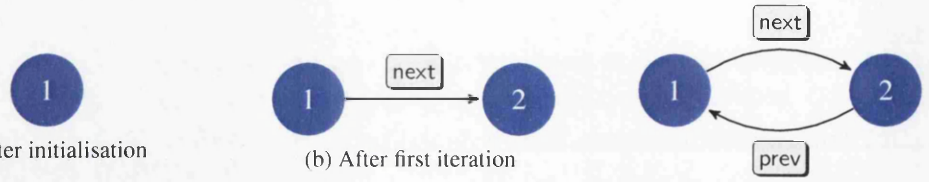
13. Reset the SUD's state to $\{ \mathit{picture} \mapsto 2 \}$, i.e. display the second picture (no change).

14. Call *doGuiAction* $\boxed{\text{prev}}$ — i.e. simulate pressing the $\boxed{\text{prev}}$ button, moving to picture 1.

15. *project* the system's state again, yielding $s' = \{ \mathit{picture} \mapsto 1 \}$.

16. $s \neq s'$, but s' is not new (i.e. it's already in the state space), so *don't* add it again or discover its actions.

[2]For the sake of this example we will suppose it has been implemented as a queue.



(a) After initialisation (b) After first iteration (c) After second iteration

Figure 3.4: State space growth in the model discovery algorithm for a 2-state example.

17. Add $\boxed{\text{prev}}$-labelled edge from s to s' in state space (figure 3.4(c)).

    (End of second iteration through algorithm.)

18. Pool is now empty, so algorithm has finished.

Note that this example doesn't really demonstrate backtracking, as there is only one action to explore in each state.

3.4.2 Nondeterminism

The above algorithm always produces a deterministic model: in a given state, a given action is only ever explored once, so the model *cannot* contain states with multiple identically-labelled actions leading to different destinations. (In section 4.5 we describe an extension to the algorithm that *can* produce nondeterministic models, under certain circumstances.)

However, the use of a state projection may cause the *algorithm* to operate nondeterministically: if some aspect of state is not projected (and thus not reset upon backtracking), but influences the effect of an action, then multiple runs of the algorithm with different exploration orders might produce differing models. We argue that this would be a sign of an ill-formed/unsound projection, that fails to capture some essential element of state, and propose *stochastic checking* of models to detect such cases. An obvious approach here is to perform a full stochastic exploration (see section 3.4.3.1) after discovery is complete, and check that the two models thus produced are isomorphic; a more practical approach is to interleave stochastic checks with model discovery by revisiting discovered state/action pairs at random, allowing early detection of ill-formed projections.

'Uninteresting' (but necessary, in the above sense) aspects of state may, of course, be filtered from the complete discovered model, and *this* can lead to a nondeterministic model. Such models may be meaningful and provide valuable insight [Dix91].

3.4.3 Variations and Extensions

The API and algorithm are deliberately generic. We now describe some useful extensions and variations to the basic picture. Note that with the possible exception of directed exploration (section 3.4.3.7), each of these extensions requires modification only of API elements: the structure of the algorithm remains identical. The basic theme of these extensions is to increase the flexibility of the algorithm, and in particular to allow more focused model discovery in order to explore more tightly specified parts of a system than is possible given the basic algorithm.

3.4.3.1 Exploration order

The order in which the state space is explored is determined by the implementation of the *Pool* data type and its associated functions. It is possible to implement almost any desired strategy without modifying the rest of the algorithm. In particular, using a stack for the pool will yield *depth-first* exploration whereas a FIFO queue yields *breadth-first*. *Stochastic* exploration can be implemented using any collection data type supporting random access, and this may be useful for checking for hidden modes or for inadequate state abstractions. If the state space is fully explored, each of these strategies eventually produces the same result, albeit in different orders; however, some of the extensions described below can break that assumption.

3.4.3.2 Conditional exploration

In *conditional exploration* we ignore particular states or actions; we need only modify *addToPool* or *pickFromPool*. For example, we might introduce a predicate on $(st, GuiAction)$ pairs to *addToPool*, so it only explores pairs for which the predicate is true (compare this with the version in section 3.3.3):

$$addToPool :: ((st, GuiAction) \to Bool) \to Pool\ st \to st \to [GuiAction] \to Pool\ st$$

By modifying *pickFromPool* similarly, we can achieve the same effect at a later stage. If the filtering condition is fixed for the entire exploration, there is no difference between these two approaches; if it can vary (depending on state space size, say) then there is a real difference, and one approach might be preferable to the other.

See section 4.3 for several examples of this approach.

3.4.3.3 Filtering SUD state items

As described in section 3.2.1, *project* dictates which aspects of SUD states are projected into the model and, as such, it strongly influences the contents and size of the discovered model. Thus in the air conditioning control example, ignoring the temperature slider reduces the discovered model to just 12 states. An obvious way to increase the flexibility of model discovery, then, is to allow some *run-time* control over this. A simple way to do this is to add a parameter to *project*, listing the model state members to be retained — and to expose that list to user control via the discovery control interface itself.

How SUD state members are identified for such filtering depends on the representation used: if the SUD state is flat, simple names suffice; if it is a JSON-like tree, a path language such as *XPath*[3] is required. Assuming the existence of a *StateItem* type fulfilling this role, the type signature of *project* then becomes:

$$project :: [StateItem] \rightarrow IO\ st$$

3.4.3.4 Controlling action discovery frequency

The algorithm calls *getGuiActions* in *every* state; for full generality this is required, in order to deal with dynamic interfaces in which elements come and go (i.e. most non-trivial GUIs). However, in particular cases it might be unnecessary or undesirable.

If the GUI is static, or if all actions can be discovered up-front, a single call on initialisation will suffice, and this can be implemented without altering the algorithm by *memoising getGuiActions*, i.e. having it

[3]`http://www.w3.org/TR/xpath`

cache the results of its first call and return them immediately for all subsequent calls.

Where automatic discovery of actions is not possible, or as a performance optimisation measure, a last resort is to hard-code the control actions into the discovery tool, i.e. in *getGuiActions* — assuming there is a way to relate these hard-coded references to the actual widgets and actions.

In any case, if the SUD has a dynamic interface, it is necessary to ensure that *doGuiAction* has no effect (and in particular keeps the SUD in the same state) for unavailable actions.

3.4.3.5 Context-sensitive exploration

Suppose there is some target state we are trying to work towards, because we are interested in how the user gets from A to B, say. We might wish to implement a hill-climbing strategy or similar, in which case we need to know the current state and the global context, so we can try to pick an appropriate action leading in the right direction.

In *context-sensitive exploration*, then, exploration may be influenced by both the current state and the current picture of the state space; thus, it is necessary to add these as parameters to *pickFromPool*:

$$pickFromPool :: st \rightarrow Gr\ st\ GuiAction \rightarrow Pool\ st \rightarrow (Maybe\ (st, GuiAction), Pool\ st)$$

3.4.3.6 Initialisation

As described above, model discovery starts with a single state in the state space, and that state's actions in the pool — where that state is just whatever state the SUD is in when discovery is begun. There are two possible variants here. One is simply to allow some control over exactly what that initial state is (as opposed to just whatever state the SUD happens to be in when discovery starts); see section 4.3.3.1 for an example of this approach. Another possible extension, if there is a programmatic way to specify SUD states, is to seed discovery with more than one. Such an approach could be useful for the kind of search mentioned in section 3.4.3.5: concurrent searches starting from several points may lead to a desired result state faster (with appropriately implemented communication between the concurrent searches). In order to be effective, some sort of scripted control over exploration would be desirable, for example allowing seed collections to be defined easily — see section 3.4.3.7.

3.4.3.7 Directed exploration and scripting

An interesting avenue of future work is *directed exploration*, in which, rather than proceeding entirely automatically, discovery is consciously directed, either interactively or programmatically, by the analyst. That is, a finer level of control is offered to the interaction programmer, allowing the focused and flexible application of the ideas already presented in this section.

For example, while the case studies described in sections 4.3 and 4.5 involve the use of conditional exploration implemented by allowing the programmer to filter explored actions before discovery begins, the set of actions explored is fixed while it is running. Instead, we might offer the ability to interrupt discovery and modify that set, so that different parts of the model involve different actions (this could be one way to tackle *modes*).

The following aspects are involved:

- ability to interrupt automated discovery, either manually or using a breakpoint-like approach (defined conditionally, rather than locationally);
- ability, when paused, to modify discovery criteria such as exploration order, action/state filtering conditions, and SUD state projection;
- ability to perform discovery step-by-step, possibly with fine control over *pickFromPool*'s behaviour, to allow choice of state and/or action explored — perhaps graphically, via the state space preview;
- support for all of these tasks via the discovery control GUI and/or programatically, for instance via a domain specific language (DSL) [MS05].

As discussed in section 3.1, we see model discovery as suitable for integration into existing development workflows, and it is clear that a DSL for discovery control is an essential component of such efforts. For without such a capability, manual intervention would be required for all but the simplest of cases, and smooth integration into iterative development workflows based on automated regression testing would be (nearly) impossible. Thus, we consider directed exploration, and particularly language support for same, to be an important area of future research.

Chapter 4

Model Discovery Case Studies

Contents

4.1 Introduction

In this chapter we present four case studies of model discovery in use. The first two, considered in some detail, represent the author's own work; the third and fourth represent the author's commentary on existing work by others. In each case we consider interesting or particular aspects of the implementation of the model discovery API/algorithm, and describe the discovered models.

4.2 Case study 1: Air Conditioning Control Panel

4.2.1 Overview

First, we return to the example briefly introduced in section 3.1.2/figure 3.1: the air conditioning control panel. The tool, its source code, and a screencast showing its operation, may be found online.[1]

As noted in section 3.1.2, the SUD is a simulation of the control panel of an air conditioning control unit, with the following controls:

- on/off;
- heat/cool;
- fan speed (low, medium, and high);
- target temperature (5–30°C, i.e. 26 settings).

These four controls are entirely orthogonal, in that they may be manipulated entirely independently: interacting with any given control never has any effect on any of the other controls. (This will not be the case in general of course, and it is easy to conceive of a variant of this example where, say, only the on/off switch is accessible when the system is switched off.) The UI's state is thus entirely encapsulated in a collection of four values, with a total state space of $2 \times 2 \times 3 \times 26 = 312$ states. This example is deliberately trivial, and was crafted specifically to facilitate investigation and implementation of the model discovery algorithm in a rigorous manner: there is essentially nothing interesting to say about the control panel itself. Thus, we may concentrate on the specifics of model discovery.

(Of course, an *actual* air conditioning system's state may be considerably more complex. At the very least, we imagine, it will contain one more input (i.e. the current ambient temperature), and also probably the state of some outputs (e.g. compressor and fan activation states). We have simply modelled (and, indeed, only simulated) the possibilities of the *interface*, but the *whole* system will clearly be considerably more complex, and probably quite interesting, including from a UI point of view. For example, suppose the system is in 'cool' mode with a target temperate of 5 degrees, and an ambient temperature of 10 degrees; if the mode is changed from 'cool' to 'heat', what happens? This might be an interesting

[1]`http://www.cs.swan.ac.uk/~csandy/phd`

question for model discovery to consider — and could lead to a redesign of the system's UI, e.g. in a non-orthogonal manner. Modelling such interactions involving a plant and time-varying quantities such as ambient temperature is, however, beyond the scope of the technique as described here.)

4.2.2 Implementation

Here we attempt to explain the key aspects of the implementation in some depth, in particular concentrating on exposing how the API and algorithm have been realised in this example.

The SUD and the model discovery tool were both written in Haskell [Pey03] using the wxWidgets [SHC05] GUI toolkit. Haskell was chosen precisely for its high degree of formality and robustness: as described in section 3.3.1, Haskell is a rich language with a very strong and thoroughgoing type system, which had two major effects here:

1. It gave us high confidence in the correctness and robustness of the implementation (both of the GUI and the model discovery algorithm). Haskell is an unforgiving language to program in, and its type checker immediately identifies many bugs that would go undetected (perhaps causing a crash, which at least has the benefit of being obvious) in a more forgiving language. Of course, run-time problems are still possible (particularly when dealing with imperative third party code as sophisticated as wxWidgets), but overall levels of confidence tend to be higher — once the code compiles.

2. The implementation is, to some extent, self-documented by the type signatures of the functions and data structures involved. It has been said of Haskell that once you work out what your type signatures should be, the implementation is frequently 'obvious'; while this is somewhat frivolous and not, in this author's experience, as true as one might hope, it again illustrates the central nature of the type system to the Haskell worldview. Having implemented model discovery in Haskell, we understood the details of algorithm, and how the various elements fit together, much more clearly than after previous implementations in (say) JavaScript. That deeper understanding led directly to the formal description of model discovery presented in the previous chapter (and also, as it happens, to a clearer and less buggy JavaScript implementation, as described in the next case study).

4.2.2.1 Overview

The SUD and model discovery tool were compiled together (indeed, we embedded the SUD UI in the tool UI, though this is not required — see discussion in section 3.1.2), so the tool's access to the SUD's internals is fixed at compile-time; once the SUD is in the model discovery code's scope, its state can be probed and updated by functions it exposes. Similarly, the wx toolkit provides good runtime access to the UI components, so given a handle on the SUD's panel widget, it is straightforward for the model discovery code to query the available controls and their actions, and to perform those actions.

Let us consider the overall code structure before looking at particular areas of interest in depth. The key modules are as follows:

- `Main.hs` — main GUI/controller, integrating model discovery tool and SUD.
- `Graphics/UI/Aircon.hs` — wxHaskell panel of aircon controls (SUD).
- `Graphics/UI/Aircon/Model.hs` — underlying model/state for aircon control panel (SUD).
- `Graphics/UI/Discovery.hs` — model discovery data types and algorithm.
- `Graphics/UI/Discovery/Pool.hs` — model discovery pool data structure (with stack, queue, and map versions).
- `Graphics/UI/Discovery/WxControls.hs` — facilities to probe and enact wxHaskell GUI components.

The codebase also includes the following modules, covering non-core aspects; unless the reader intends to investigate the codebase deeply, they may safely be ignored, and will not be discussed any further here:

- `AirconGUI.hs` — 'main' wrapper for running the aircon control panel as a standalone app.
- `Graphics/UI/Discovery/FSM.hs` — FSM health checks: is it weakly connected, etc..
- `Graphics/UI/Discovery/GML.hs` — graph output in Graph Modelling Language.
- `Graphics/UI/Discovery/StAn.hs` — state annotations for state spaces (used to produce animations of model growth).

- `Graphics/UI/WX/Imgview.hs` — custom wx widget for drawing an image to a scrolledWindow, used for graph view.
- `Control/Concurrent/CHP/Worker.hs` — asynchronous worker threads; used to render the graph preview in a child thread, in order to not block model discovery while it renders.
- `Config.hs` — configuration file handling, allowing override of GraphViz binaries.

Of these, `Main.hs` is certainly the most complex, containing as it does all of the top-level code for laying out the GUI, dealing with interactions with the user, rendering the graph preview in another thread, and saving the model in various formats. Behind this somewhat intimidating front-end, the actual code doing the work of model discovery is quite clean and well-encapsulated in the modules `Discovery.hs`, `Pool.hs` and `WxControls.hs`.

4.2.2.2 SUD code

The SUD's code, in `Aircon.hs` and `Aircon/Model.hs`, is fairly straightfoward. Its state is represented using a simple record type with four components:

```
data Aircon = Aircon {
  onOff :: OnOff,
  mode :: Mode,
  fans  :: Fans,
  temp  :: Temperature
}
```

where *OnOff*, *Mode* and *Fans* are enumerated types, and *Temperature* is a simple type wrapper around *Integer*.

This inner state is bound to the GUI state using the Observer design pattern [Gam+95] (via the `simple-observer`[2] Haskell package), applied bidirectionally — in the SUD code, not the model discovery tool, of course. The main interface between the SUD and the model discovery code is the function *defaultAirconControlPanel*, which returns a triple containing:

[2]`http://hackage.haskell.org/package/simple-observer`

- The panel widget containing the four controller widgets (for probing by *getGuiActions*, etc.).
- A 'subject' in the sense of the Observer design pattern, i.e. a data structure that allows observers to be informed immediately of changes to the aircon panel's inner state; the model discovery part's *project* function is then a simple wrapper around this component.
- A *reset* function allowing the inner state to be updated from outside, for backtracking. (Since the aircon's widgets are bound to the inner state by the Observer pattern, a call to *reset* will also result in them automatically updating *their* state.)

This is called once when the model discovery tool initialises, and the three return values are used by the various model discovery components as described below.

4.2.2.3 Probing/enacting widgets

The module `Graphics/UI/Discovery/WxControls.hs` contains the code for dealing with the GUI widgets of interest. First, we have two data types and one function for referring to/setting GUI widgets:

- Data type *GuiControl*, with two cases: *WxRadio* and *WxSlider*, each of which carries a string for the widget name and a reference to the actual widget.
- Data type *GuiValue*, with two cases: *WxRadioValue* and *WxSliderValue*, each of which carries an integer value.
- Function *setControlValue*, which takes a *GuiControl* and a *GuiValue* and sets the given widget to the given value as a side effect; this is called by *doGuiAction*, below.

Then, we have two data types and one function related to simulating user actions:

- Data type *UpDown* representing slider actions, with the two cases *Up* and *Down*.
- Data type *GuiAction* representing a user action, with two cases: *WxRadioAction*, carrying a *WxRadio GuiControl* value and an integer; and *WxSliderAction*, carrying a *WxSlider GuiControl* value an a *UpDown* value.
- Function *doGuiAction*, which takes a *GuiAction* value and enacts it as a side effect.

Finally, three functions related to learning the SUD's available widgets and actions at any given moment:

- *learnGuiControls* — given the wx panel widget returned by *defaultAirconControlPanel* (see above), returns a list of *GuiControl* values corresponding to the widgets in that panel. It actually walks the panel recursively (i.e. if it contains subpanels), though the returned list is always flat. Of course, it happens that in the aircon example, this need only be called once, because the set of widgets never changes — though this optimisation is not in fact implemented.
- *getControlActions* — given a *GuiControl* value, return a list of available *GuiAction* values for that widget; the possibilities are hard-coded here, but not including actions that we would expect to lead to a self-loop — e.g. attempting *Up* on a slider in its maximum position.
- *getGuiActions* — combines *learnGuiControls* and *getControlActions* to compute a list of *GuiAction*s for the panel widget returned by *defaultAirconControlPanel.*

4.2.2.4 The pool

The module `Graphics/UI/Discovery/Pool.hs` contains data structures for the pool of unexplored state/action pairs. There are actually three pool implementations here, unified by a *typeclass*. A Haskell typeclass is analogous to a Java interface, in that it encapsulates a contract regarding the capabilities of some datatype — in particular, a list of functions that a data type implementing that typeclass must implement. In this case, the typeclass *Pool* contains seven functions:

- *emptyPool* — create an empty pool.
- *poolNull* — check if the pool is empty.
- *poolStates* — count the states in the pool.
- *poolActions* — count the actions (actually state/action pairs) in the pool.
- *addToPool* — add a state and all its actions to the pool.
- *pickFromPool* — pick/remove the next state/action pair from the pool.
- *isStateInPool* — check if a given state is anywhere in the pool.

There are then three implementation of this typeclass:

- *PoolStack* — a stack of (state, action) pairs, implemented using the *BankersDequeue* (double-ended queue) datatype from the `dequeue` package[3]. This results in a depth-first exploration strategy.
- *PoolQueue* — a queue of (state, action) pairs, again implemented as a *BankersDequeue*. This results in a breadth-first exploration strategy.
- *PoolMap* — a mapping from states to lists of actions, implemented using the *Data.Map* datatype from the standard Haskell library[4] (an obvious first choice for a straightforward and reasonably fast implementation). Picking from the pool involves picking the first state in the map, and then picking its first action (and if that state now has no actions remaining, removing it from the map's index). As the order of items in a *Data.Map* is undefined, this results in an unpredictable exploration order.

The choice of which of these is used is hard-coded in the model discovery tool, rather than being exposed via the GUI — so to change it requires recompiling the tool. A video showing the different exploration orders in action can be found online.[5]

4.2.2.5 Core discovery algorithm and data structures

Finally, the module `Graphics/UI/Discovery.hs` contains the core model discovery data structure and algorithm implementation. The *Discovery* data structure is essentially as described in section 3.3.4, and as noted there, is polymorphic over the type of the state being projected from the SUD. This is why the *initDiscovery* function, which sets up a *Discovery* value ready to start model discovery, takes most of the value's elements as parameters:

- A *Pool* value, as described above.
- The *project* function; in our implementation this is a simple wrapper, constructed in `Main.hs` around the aircon observer subject returned by *defaultAirconControlPanel*. As has already been

[3]`http://hackage.haskell.org/package/dequeue`
[4]`http://hackage.haskell.org/package/containers`
[5]`http://www.cs.swan.ac.uk/~csandy/phd`

noted several times, in this example we have simply projected the entire inner state of the SUD — a simple bundle of 4 values. If we had wished to restrict this in some way (e.g. ignoring the temperature value), we could simply have plugged in an alternative *project* implementation between the SUD and the *Discovery* value.

- The *reset* function, exactly as returned by *defaultAirconControlPanel.*
- An *eq* comparator for project states; like *project*, this is simple in this example, but any modification to *project* would necessitate a corresponding modification here.
- The *getGuiActions* function from `Graphics/UI/Discovery/WxControls.hs`, as described above.

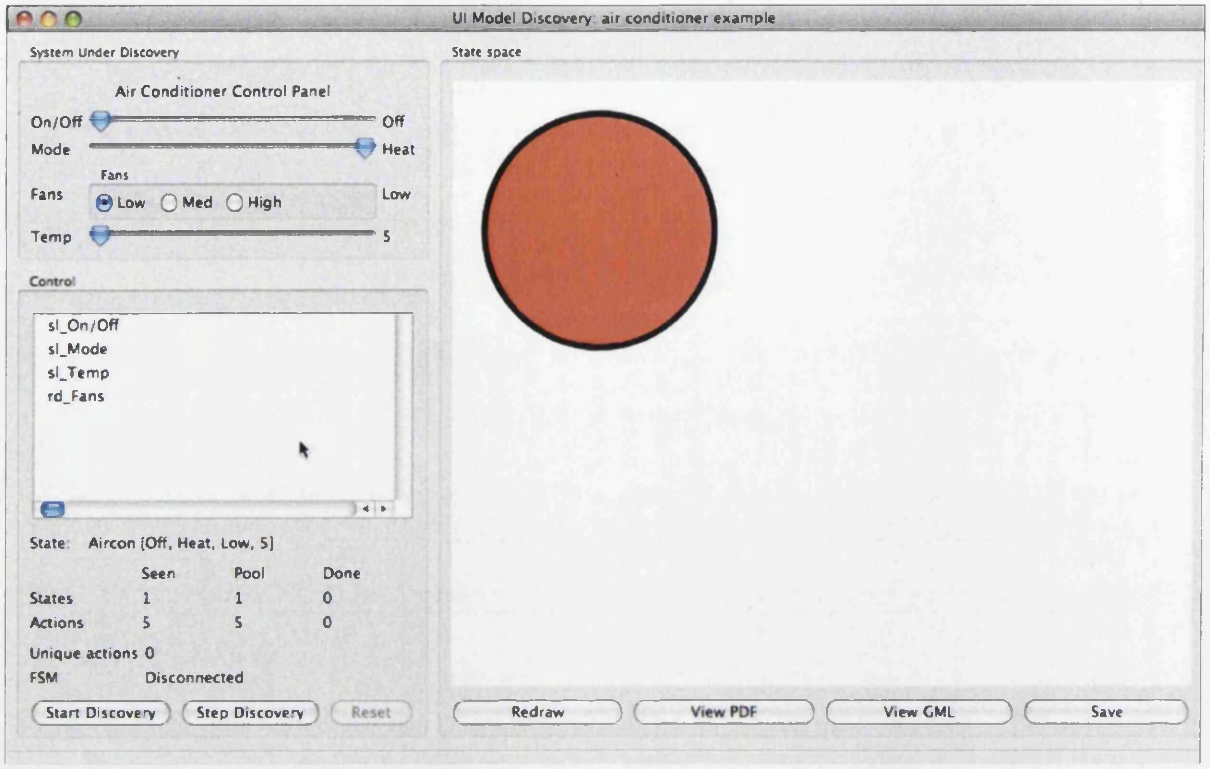
The *Discovery* value's *stateSpace* is a graph mapping projected states to *GuiAction* values, implemented using the `fgl`[6] ('functional graph library') package's *Inductive.Tree* data type.

The model discovery algorithm itself is implemented in a collection of functions over this *Discovery* value, essentially following the descriptions given in section 3.3.4, along with a few helper functions and some extras for housekeeping tasks such as reporting the number of states in the state space so far.

4.2.3 Example models

At initialisation, the *Discovery* value projects the SUD's current state as the start state of model discovery; the situation is shown in figure 4.1: there is one state in the state space, with 5 actions to be explored (in this case they are *on*, *cool*, *fans_medium*, *fans_high*, and *temperature_up*). After a single step of the model discovery algorithm the situation is as shown in figure 4.2: one action has been explored, leading to a new state, whose 5 actions have also been added into the pool. (From the displayed projected state, we can infer that the action explored was *fans_high.*) After a few more steps, all five actions available in the initial state have been explored (figure 4.3), and a few steps later, another state has been fully explored, including some edges back to the initial state (figure 4.4). Finally, after 1848 actions have been explored, the complete state space of 312 states has been discovered (figure 4.5).

[6]`http://hackage.haskell.org/package/fgl`


UI Model Discovery: air conditioner example
System Under Discovery
State space
Air Conditioner Control Panel
Redraw
View PDF
View GML
Save

On/Off Off
Mode Heat
Fans: Low [✓] Med High — Low
Temp 5

Control
sl_On/Off
sl_Mode
sl_Temp
rd_Fans

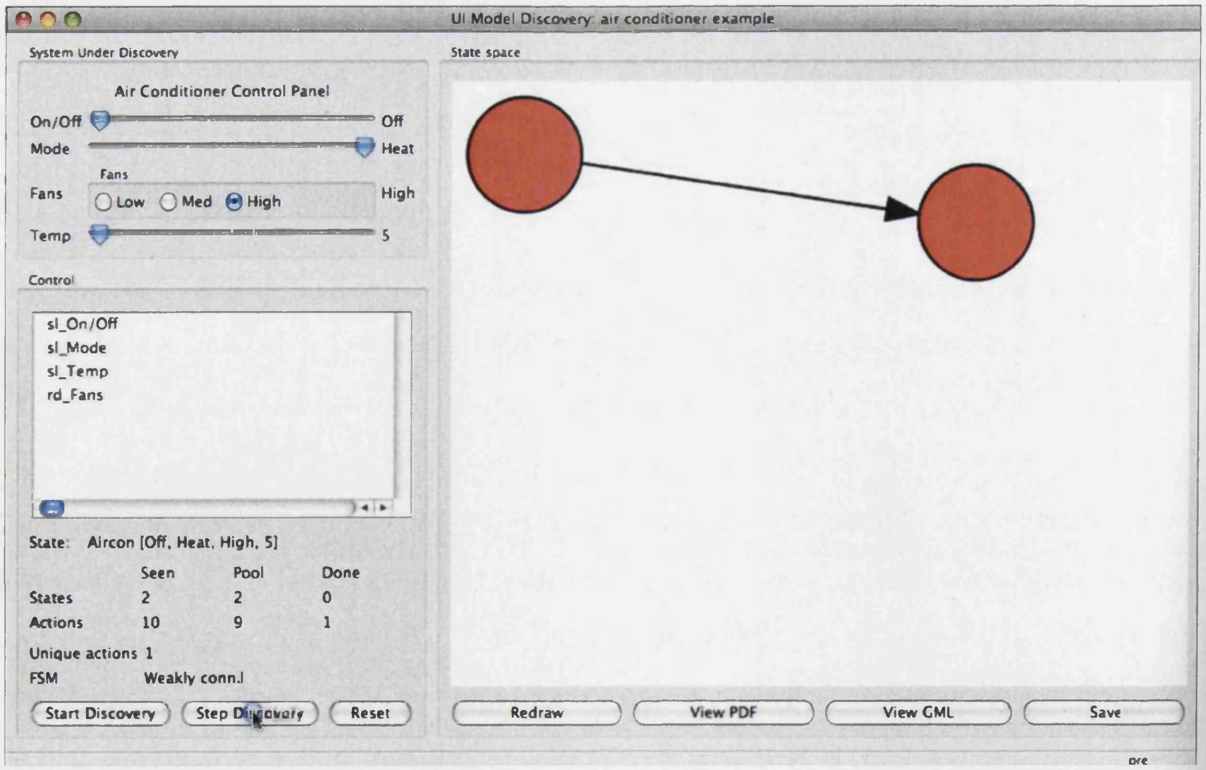
State: Aircon [Off, Heat, Low, 5]

	Seen	Pool	Done
States	1	1	0
Actions	5	5	0

Unique actions 0
FSM Disconnected

Start Discovery | Step Discovery | Reset

Figure 4.1: Initial state of model discovery on the aircon example


UI Model Discovery: air conditioner example
System Under Discovery
State space
Air Conditioner Control Panel
Redraw
View PDF
View GML
Save

On/Off Off
Mode Heat
Fans: Low Med High [✓] — High
Temp 5

Control
sl_On/Off
sl_Mode
sl_Temp
rd_Fans

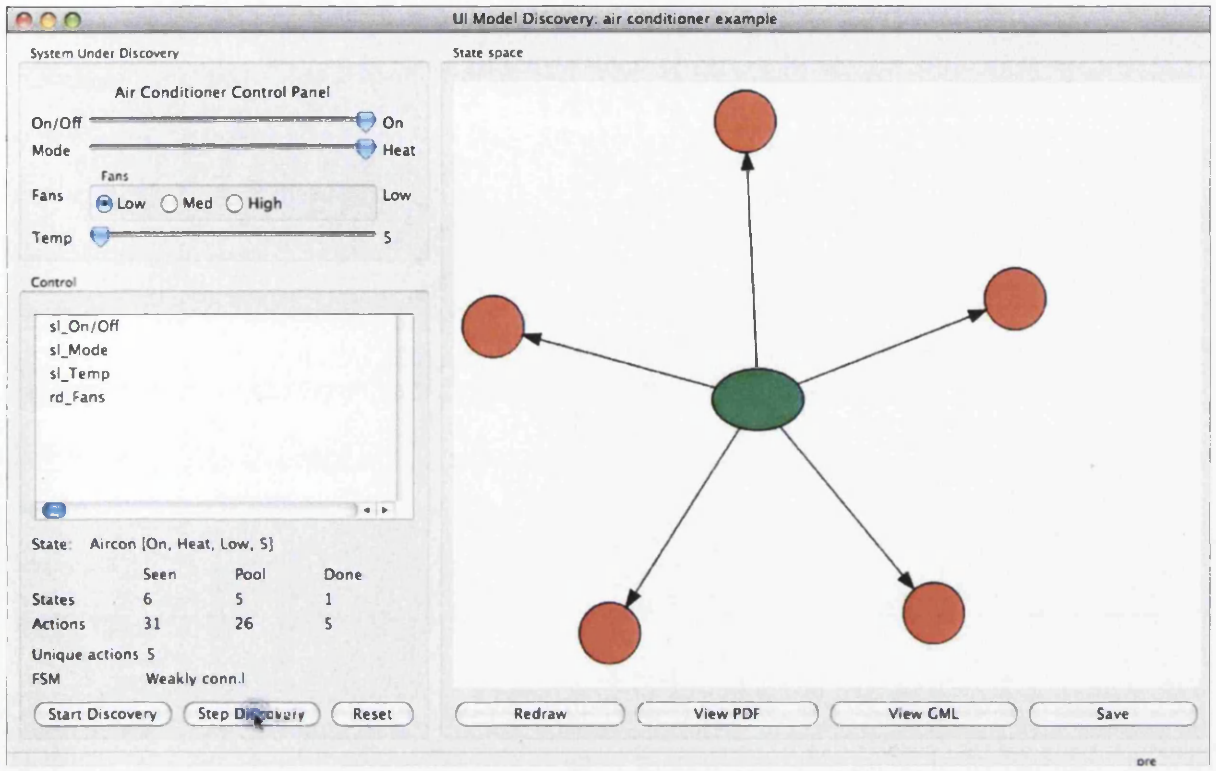
State: Aircon [Off, Heat, High, 5]

	Seen	Pool	Done
States	2	2	0
Actions	10	9	1

Unique actions 1
FSM Weakly conn.l

Start Discovery | Step Discovery | Reset

Figure 4.2: Aircon discovery after a single step: discovery of a new state



On/Off: On
Mode: Heat
Fans: [✓ Low] Med High — Low
Temp: 5

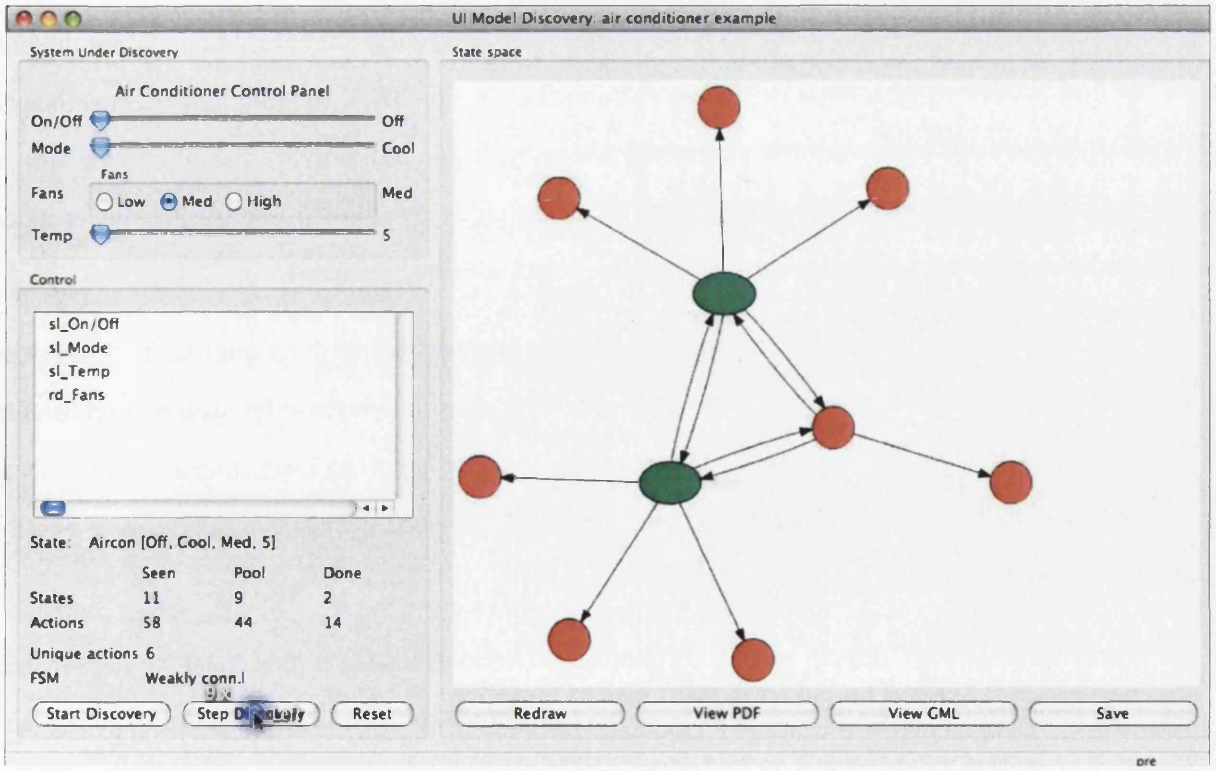
sl_On/Off
sl_Mode
sl_Temp
rd_Fans

State: Aircon [On, Heat, Low, 5]

	Seen	Pool	Done
States	6	5	1
Actions	31	26	5

Unique actions 5
FSM Weakly conn.l

Figure 4.3: Aircon discovery after all actions of first state have been explored



On/Off: Off
Mode: Cool
Fans: Low [✓ Med] High — Med
Temp: 5

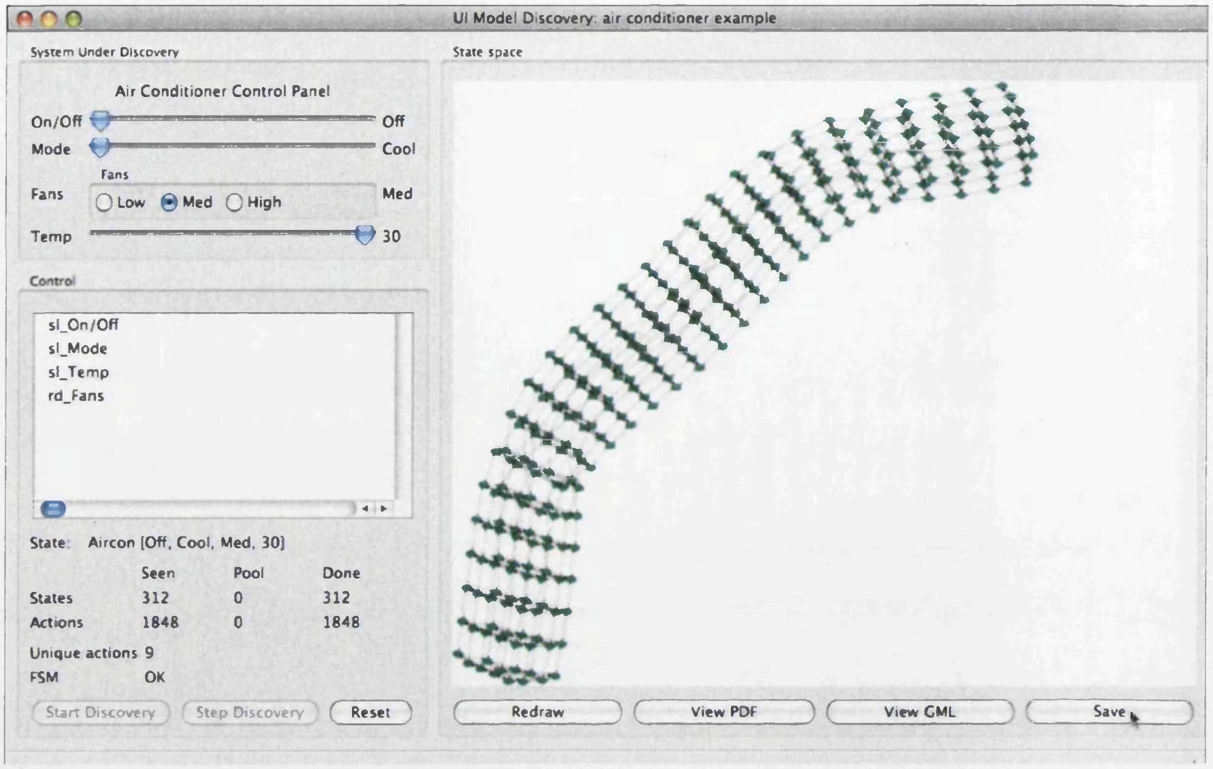
sl_On/Off
sl_Mode
sl_Temp
rd_Fans

State: Aircon [Off, Cool, Med, 5]

	Seen	Pool	Done
States	11	9	2
Actions	58	44	14

Unique actions 6
FSM Weakly conn.l

Figure 4.4: Aircon discovery a little later: backlinks appearing



```
sl_On/Off
sl_Mode
sl_Temp
rd_Fans

State:  Aircon [Off, Cool, Med, 30]

          Seen   Pool   Done
States    312    0      312
Actions   1848   0      1848

Unique actions 9
FSM            OK
```

Figure 4.5: Aircon discovery upon completion: 312 states

The SUD's interface is simple and orthogonal enough that components may be dropped from the projection freely without incurring any hazards mentioned in section 3.4.2, so while this example helped us clearly understand the mechanisms of model discovery, it does not provide insight into the task of choosing an appropriate projection function. The algorithm as described in section 3.4 was encoded essentially directly, except that it does not cycle freely, but under user control.

The main value of this implementation was in providing a clear, rigorous and quite formal picture of the requirements for model discovery, and of how the basic algorithm ought to be implemented. Before this implementation, model discovery had been implemented *ad hoc* as an interesting and fruitful idea, without being studied in and of itself. The aircon example provided a context for performing exactly that kind of study.

For this purpose, Haskell's rigour was highly valuable; conversely, however, that rigour perhaps makes experimentation with new ideas, variations and extensions more difficult than in a more forgiving language. As a small example, to introduce *conditional exploration* would (at the least) require modifying the *addToPool* function (and each of its implementations, and calls) as described in section 3.4.3.2. This would then raise the question of how the control predicate is to be implemented, leading in turn to questions of scope, and probably necessitating rewriting a large number of functions and their types in order make new data available to that function. This process can be managed; for example, rather than explicitly passing around ever-growing lists of variables they may be explicitly collected together in a data structure, allowing elements to be added or removed over time more easily (the *Discovery* structure in section 3.3.4 is one example of this strategy). This can make code more opaque, however.

By contrast, in our next case study (implemented in JavaScript), it was very easy to introduce conditional exploration, as we shall see — just a small modification to one function, accessing some globally accessible data. Of course, it's *possible* to make data globally visible in Haskell too, but doing so is far less natural there: Haskell promotes the kind of side-effect free programming which led to our fairly rigid implementation — but again, for the purpose of this example, that was exactly what was required.

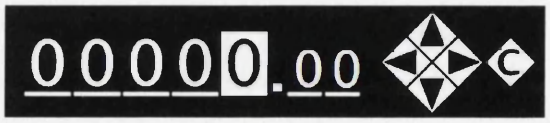
4.3 Case study 2: Independent Digit / '5-key' Number Entry

4.3.1 Overview

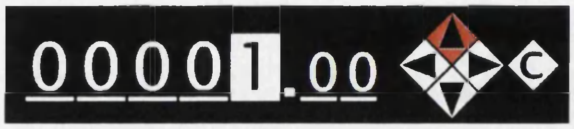
This case study concerns interactive number entry, focussing on an interaction style characterised in [Ola12] as *independent digit entry*, where the user controls the value of each digit separately and (conceptually at least) in any order, navigating around the space of possibilities using up, down, left and right buttons (which we write as $\boxed{\blacktriangle}$, $\boxed{\blacktriangledown}$, $\boxed{\blacktriangleleft}$, $\boxed{\blacktriangleright}$ respectively): the $\boxed{\blacktriangle}$ / $\boxed{\blacktriangledown}$ buttons modify the value of the digit currently being edited, and the $\boxed{\blacktriangleleft}$ / $\boxed{\blacktriangleright}$ buttons move a cursor around, thereby selecting the digit for modification.

Figure 4.6 shows a few steps in this process, starting from an initial zero state, and entering the key sequence $\boxed{\blacktriangle}\boxed{\blacktriangleleft}\boxed{\blacktriangle}\boxed{\blacktriangleleft}\boxed{\blacktriangledown}$. In this example, the number entry system is of a type we call *Simple Spinner*, which may be characterised by these two facts:

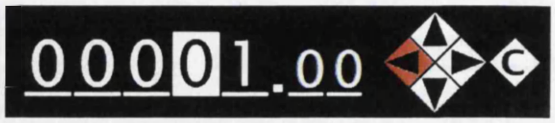
- Each digit in the number is completely independent of the others, so that pressing $\boxed{\blacktriangle}$ or $\boxed{\blacktriangledown}$ only ever affects the value of the current digit.
- The digits 'wrap around', so that pressing $\boxed{\blacktriangledown}$ on a value of 0 changes it to 9 (as seen in figure



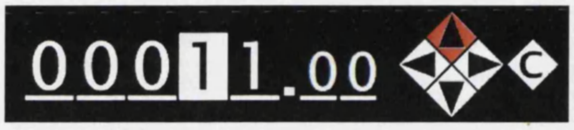
(a) Initial state



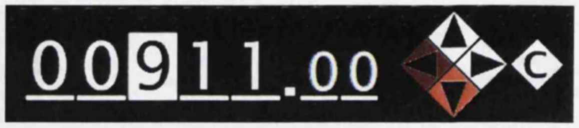
(b) ▲ pressed: current digit incremented.



(c) ◄ pressed: cursor moves one space to left.



(d) ▲ pressed: current digit incremented.



(e) ◄ ▼ pressed: digit wraparound from 0 to 9.

Figure 4.6: 5-key/independent digit number entry in action, using the 'Simple Spinner' entry routine.

4.6(e)), and pressing ▲ on a value of 9 changes it to 0.

A Simple Spinner system is, thus, highly regular and predictable — but it is not the only possibility. In this case study we compare and contrast the behaviour of a Simple Spinner system with two other modes of operation: one where an action on some digit can sometimes affect other digits according to the rules of arithmetic, and a real-world example which extends this arithmetic behaviour with some interesting irregularities.

5-key number entry systems are found on a range of hardware devices, but in particular are becoming popular on medical devices such as infusion pumps, with several manufacturers releasing devices utilising this technique. A key point here is that (as alluded to in the previous paragraph), there is a wide variety in how 5-key number entry systems behave in practice, and a given input sequence may lead to wildly varying results on two apparently similar systems — even on two identical pieces of hardware, if they happen to be running different firmware. In [Thi+12] we explored this space systematically, investigating a range of possible implementations and measuring their resilience to unnoticed keying errors (e.g. key bounces, missed keys and transposition); while that work did not involve model discovery, here we describe the result of extending one of the implementations with model discovery capabilities. A running version of the code used in this case study, and its source code, may be found online.[7]

[7]`http://www.cs.swan.ac.uk/~csandy/phd`

The three number entry routines examined in this case study are as follows:

4.3.1.1 Simple Spinner

This is the routine pictured in figure 4.6 and described above. The ▲ / ▼ buttons increment/decrement the current digit's value, wrapping around between 0 and 9 (hence 'spinner'), and the ◄ / ► buttons move the cursor left and right. Note that in this example, the ◄ / ► buttons *do not* cause horizontal wrap around — and this is the case for all three routines described here. Also note that in order to keep this routine as simple as possible, every digit position always displays a value, even if it is just 0.

4.3.1.2 Simple Arithmetic

Here, a change to a digit does not always only change that digit; rather, it results in an arithmetical change to the number displayed, according to the position of the cursor. For example, if the display is currently 190.00 (with the cursor on the 'tens' column), then ▲ increases the overall displayed value by 10, yielding 200.00, and ▼ decreases it by 10, yielding 180.00.

This leads to the question of how to handle operations that would take the value beyond its maximum and minimum bounds, to which there are two possible responses: reject the operation, or clamp the value. For example, suppose the display is 045.00 and the user hits ▼; since that would 'ideally' subtract 100 yielding a value of −55, and assuming the system cannot display negative numbers, either the operation would be rejected leaving the display unchanged (or perhaps displaying an error and/or beeping — see [Thi+12] for a discussion of why this is valuable), or the value would be clamped at its minimum, say zero: 000.00. This is how it is implemented in our case study.

4.3.1.3 BBraun v686E VTBI

This is a simulation of a number entry routine found on a real-world medical device; the device in question is the *BBraun Infusomat Space*, an infusion pump, and the routine in question is for entering the *Volume To Be Infused* or **VTBI** — specifically, the VTBI routine found on devices running firmware version *686E*. This routine is generally interesting as a real-world example, but also in particular because it exhibits some structural irregularities not found in the other two routines, and reflects the truth that real-world systems are indeed often less ‘clean’ and easy to specify or analyse than ‘ideal cases’ — either because they have grown organically, or because particular domain-specific requirements have been identified and targetted.

Note well that our analysis here is not intended in any way as a judgement of the value of the BBraun VTBI routine as compared to the other routines (or to any other example); we recognise that industrial software engineering must necessarily take into account unpredictable (and from our point of view, unknowable) requirements and constraints, and in particular we remark that the irregularities seen in the BBraun example may well be of great benefit to the users of that system. Our intention here is simply to demonstrate that model discovery can produce models that can provide insight into the existence and extent of such irregularities, in an example of real-world complexity.

The basic behaviour of the routine is arithmetical, as described above, without horizontal/cursor wrap-around. There are three areas in which this basic behaviour is extended.

First, it admits entry of numbers over a range of magnitudes. Its display can show up to five digits, but the range displayed is a sliding window between hundredths and ten-thousands (i.e. seven orders of magnitude). However, the sliding window does not move uniformly and in fact has three possible positions: *hundredths to tens*, e.g. `99.99`; *tenths to hundreds*, e.g. `999.9`; and *units to ten-thousands*, e.g. `99999`. Note that the first two ranges are four digits wide, whereas the third is five wide; it is not, for example, possible to enter `9999.9`.

Second, it has range-dependent non-zero minimum values, which can prevent the entry of certain syntactically valid values. For example and in particular, in the *hundredths to tens* range, the lowest non-zero value the device allows to be displayed is `0.1`, which is *not* the lowest value in that range, i.e. `0.01`. To be clear: the hundredths column *is* accessible, and (say) `0.11` is allowed — but no value smaller than 0.1 can be accessed. This means that, for example, from the initial display of `0.__` the key sequence ▶ ▶ ▲ yields the display `0.10` — a case where an ▲ action has *no effect* on the current digit!

Third and finally, it has a simple memory facility around its maximum values, apparently intended to allow users to easily undo accidentally hitting the maximum value under some circumstances. For example, if the display is 90000 and the user presses ▲, the display is then clamped at the maximum value of 99999. If the user's next action is to press ▼, the previous display is recalled from memory, i.e. it returns to 90000. However, the memory is short: the key sequence ▲ ▶ ◀ ▼ would yield the display 89999 in this case, the ▶ action having cleared the memory.

We wrote a set of 138 unit tests in order to document the behaviour of this routine and to test our simulation against that behaviour; the tests probe both the 'normal' arithmetic behaviour of the routine, and the various corner cases mentioned above, and in combination provide high assurance that our simulation is indeed a faithful reproduction of the actual device's behaviour. The unit tests may be found (and run) with the online version of this example — see above.

4.3.2 Implementation

4.3.2.1 Overview

In this section we describe the implementation of the number entry system and its three routines in more detail, concentrating on the aspects of interest from the point of view of model discovery, and then we describe our model discovery implementation. This case study was implemented as a JavaScript client-side system, i.e. the whole thing runs inside a web page and there is no server-side component.

4.3.2.2 SUD: `Digits.js` and `vtbi/*.js`

A detailed description of the SUD's architecture and internals is beyond the scope of this thesis, but from the point of view of model discovery, there are five modules of interest:

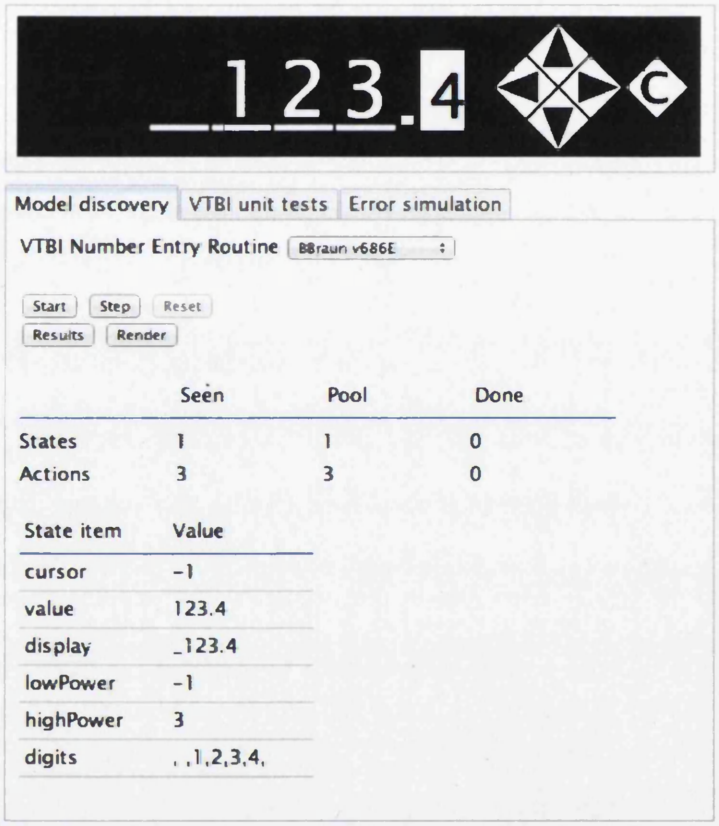
- `Digits.js` — defines a *Digits* object encapsulating the digits display. Its public interface includes functions such as *getCursor* and *setCursor* to query/move the cursor; *getDigit* and *setDigit* to get/set particular digits; *getDigits* and *setDigits* to get/set all digits at once; *getContent* and *setContent* to get/set all as a string, and several functions related to getting/setting the minimum/maximum accessible and visible powers of ten.
- `vtbi/VTBIEntry.js` — multiplexing interface/facade for the three VTBI number entry

routines; the top-level UI responds to button-clicks by calling functions here, that in turn call the requisite functions in whichever routine is actually selected. From our point of view, the key ones are *left*, *right*, *up*, *down* and *clear* (hooked up to the UI's buttons), and *set* (to set to a particular value).

- `vtbi/SimpleSpinner.js` — simple spinner implementation. Here *left* and *right* move the cursor using the functions *Digits.getCursor* and *Digits.setCursor*, whereas *up* and *down* increment/decrement the selected digit (with wraparound) using the functions *Digits.getCursor*, *Digits.getDigit*, and *Digits.setDigit* — and that's basically it: about 35 lines of code.
- `vtbi/SimpleArithmetic.js` — simple arithmetic implementation. Here *up* and *down* take into account the value of the whole display, and so use *Digits.getContent* and *Digits.setContent*; this is another very simple module: about 50 lines of code.
- `vtbi/BBraun_686E.js` — BBraun implementation. At about 165 lines of code, this is considerably more complex than either of the other routines, in order to accommodate the routine's irregularities. In particular, *up* and *down* contain special logic for dealing with upper and lower bounds respectively, all the public functions have some awareness of the memory behaviour, and after any update an *updateDisplay* function is called which takes care of the display range corner cases.

4.3.2.3 Model discovery: `Discovery.js` and `VTBIDiscovery.js`

The extension of the UI for model discovery is shown in figure 4.7, with the SUD embedded at the top. Other than the number entry routine selector at the top (which is not strictly part of model discovery but is placed here for convenience while experimenting), it is very similar to the control interface for the aircon UI in the previous case study: in particular there are start/stop, step, and reset buttons, rendering facilities (to be discussed in section 4.3.3), and readouts of the current pool and state space sizes, and the current projected state (with a bit more detail than in the aircon example). The details of the UI's implementation (in `discovery/DiscoveryUI.js` and the front-end HTML) are beyond the scope of this thesis. There are two modules of interest to us: `discovery/Discovery.js` and `discovery/VTBIDiscovery.js`



VTBI Number Entry Routine: BBraun v686E

	Seen	Pool	Done
States	1	1	0
Actions	3	3	0

State item	Value
cursor	-1
value	123.4
display	_123.4
lowPower	-1
highPower	3
digits	, ,1,2,3,4,

Figure 4.7: The 5-key number entry interface and associated model discovery tool

```
project : function() {
  return {
    __keys : ["cursor", "value", "display", "lowPower", "highPower", "digits"],
    cursor : Digits.getCursor(),
    value : Digits.getValue()/100,
    display : Digits.getContent().replace(/ /g, "_"),
    lowPower : Digits.getPowVisLo(),
    highPower : Digits.getPowVisHi(),
    digits : Digits.getDigits()
  };
},
```

Figure 4.8: The *VTBIDiscovery.project* function from `VTBIDiscovery.js` — a fixed projection of the full visible state of the number entry system.

The module `discovery/Discovery.js` contains the generic model discovery code, with no number entry/VTBI-specific parts. It is a straightforward implementation of the generic algorithm from section 3.4, based around three objects:

- *DiscoveryPool* implements the pool as a queue of state/action pairs using JavaScript's built in `Array` type.
- *DiscoveryStatespace* implements the state space as a JavaScript object mapping source states to maps from action names to destination states, along with some supporting data.
- *Discovery* is the module's top-level interface; `discovery/DiscoveryUI.js` uses this to initialise and run model discovery, to query its current status, and to make the results available for further processing. Its main function of interest is *step* which performs a single step of the basic model discovery algorithm. The details of state projection and widget query/enaction are generic: initialisation of the *Discovery* object includes a *target* parameter, which in our case will be the *VTBIDiscovery* object discussed below.

The module `discovery/VTBIDiscovery.js` is more interesting, as it is parameterised in order to allow conditional exploration.

Its main member is a *VTBIDiscovery* object encapsulating the parameterisation of model discovery for the number entry context, in particular functions *project*, *compare*, and *reset* for state projection, comparison and backtracking, and *getGuiActions* to probe the currently available actions.

```
var fullExploration = {
  name : "Whole state space",
  comments : "No restrictions",
  prefix : "",
  getGuiActions : function() {
      return [
        ["up"   , VTBIEntry.up],
        ["down"  , VTBIEntry.down],
        ["left"  , VTBIEntry.left],
        ["right", VTBIEntry.right]
      ];
  },
  validState : function(state) {
    return true;
  }
};
```

Figure 4.9: A *VTBIDiscovery* configuration object from `VTBIDiscovery.js` — full exploration of all actions.

Here *project*, *compare* and *reset* are fixed for all discovery tasks, i.e. we simply project all aspects of system state in this case study — see figure 4.8 for *project*, for example; this choice was made on the basis that the state space is regular in terms of what state variables are available — there are no variables that become live only in particular states.

However, *getGuiActions* is itself parameterised, and comes from a configuration object passed to *VTBIDiscovery* upon initialisation, in order to allow the investigating programmer to restrict exploration to some subset of the full (and very large) state space of the system. Figure 4.9 shows an example of such a configuration object, in particular the configuration for full exploration of the entire state space. It contains the following items:

- *name* — a name by which to refer to this configuration.
- *comments* — on the details of the configuration, as a reminder to a human reader.
- *prefix* — a string of actions to perform before model discovery begins, as a restricted case of *directed exploration* (see section 3.4.3.7). In figure 4.9, this is empty.
- *getGuiActions* — the function called every time a new state is discovered in order to learn its possible actions; it returns a list of JavaScript Array objects, each of which has two elements: a readable name (used to label the induced edge in the graph) and the function to call in order to

perform the action (here, this is always one of the *VTBIEntry* facade methods wrapping whatever number entry routine has been chosen). In figure 4.9, the set of actions is fixed to 'all actions', whatever the state.

- *validState* — a boolean function called when a new state is discovered, guarding addition of that state to the state space, and of its actions to the pool. Thus, if this function returns *false* for some discovered state, it will be ignored by model discovery. This provides a further level of control over conditional exploration, beyond what can be easily achieved with *getGuiActions*. In figure 4.9, all states are allowed; see section 4.3.3.2 for an example where this is not the case.

Realistically, running this full exploration would not be advisable: the state space produced is huge (10^7 states and 4×10^7 edges for the Simple Spinner) and takes hours to compute. As such, the module `discovery/VTBIDiscovery.js` contains a number of these configuration objects, covering a number of restricted exploration tasks. The choice of which configuration object is used is hard-coded into the module's code; thus, to switch model discovery strategy, the code needs to be altered and the model discovery tool reloaded. The choice could of course be exposed via the model discovery UI, e.g. as a drop-down list of available strategies.

In the next section we look at some of these model discovery configurations, and their results.

4.3.3 Example models

4.3.3.1 Tenths, hundredths only; digits 0–2 only

Figure 4.10 shows a configuration that uses *getGuiActions* in order to explore only the tenths/hundredths range of the state space, restricting the digits explored to just the set $\{0, 1, 2\}$. In any given state:

- *up* is to be explored only if the current digit is 0 or 1;
- *down* is to be explored only if the current digit is not 0 (avoiding vertical wraparound in the Simple Spinner, and arithmetic subtraction in the other routines);
- *left* is to be explored only if the cursor is in position -2 (cursor values correspond to powers of ten, and -2 is the lowest possible, i.e. hundredths);

```
var vtbidcMinimalHundredths = {
  name : "minimal hundredths",
  comments : "Tenths, hundredths only; digits 0-2 only",
  prefix : "R",
  getGuiActions : function() {
      var cursor = Digits.getCursor();
      var digit = parseInt(Digits.getDigit(cursor));
      var guiActions = [];
      if (digit < 2) {
        guiActions.push(["up" , VTBIEntry.up]);
      }
      if (digit > 0) {
        guiActions.push(["down" , VTBIEntry.down]);
      }
      if (cursor < -1) {
        guiActions.push(["left" , VTBIEntry.left]);
      }
      if (cursor > -2) {
        guiActions.push(["right", VTBIEntry.right]);
      }
      return guiActions;
  }
}
```

Figure 4.10: Model discovery configuration object: exploring only the digits $\{0, 1, 2\}$ in the tenths/hundredths range.

- *right* is to be explored only if the cursor is in a position left of −2 (as none of our routines implement horizontal wraparound, a *right* action when the cursor is in position −2 would only result in a self-loop anyway).

Note the absence of *validState* in this example: this implicitly accepts all discovered states, i.e. this configuration's only conditional exploration aspects are those arising from *getGuiActions*.

Also note the *prefix* value: upon initialisation, all three number entry routines put the cursor in position 0 (units); according to this configuration's rules, we would then never explore *left* or *right* actions, and not get the state space we're interested in: following the approach suggested in section 3.4.3.6, a single *right* action before starting exploration takes us into the desired area, however.

The results of performing this model discovery task on the three number systems are shown in figures 4.11 and 4.12. In each of these figures, nodes are labelled with the projected state's numerical value (i.e. *state.value*, programmaticaly); cursor position is not explicitly given but can usually be inferred by looking at adjacent states and the edges between them. (For example, at the top of figure 4.11 a 0 state leads to a 0.01 state with an *up* action; clearly in both states the cursor is in position −2, i.e. the hundredths column.)

Figure 4.11 shows the discovered state space for both Simple Spinner *and* Simple Arithmetic — a highly regular space in both cases; clearly in this corner of the total state space, arithmetical behaviour adds nothing new. Probably the most interesting thing to note is that (as you would expect) for each value there is an adjacent pair of states, linked by *left* / *right* edges.

Figure 4.12 shows the BBraun state space; there are two things to note here:

1. It has less states. As described in section 4.3.1.3, this routine has range-dependent minima, excluding the entry of certain values. For example and in particular, as noted there, 0.01 is not accessible in this routine, and we see its absence here.

2. It has some one-way edges. Again, this reflects the behaviour around the minimum allowed value in the display range being explored, i.e. 0.1 — so for example a *down* operation on 0.12 with the cursor in the tenths column clamps the value at 0.1 rather than leading to 0.02 — but then an immediate *up* operation leads to 0.20, *not* back to 0.12 (understandably, though this is *not* how

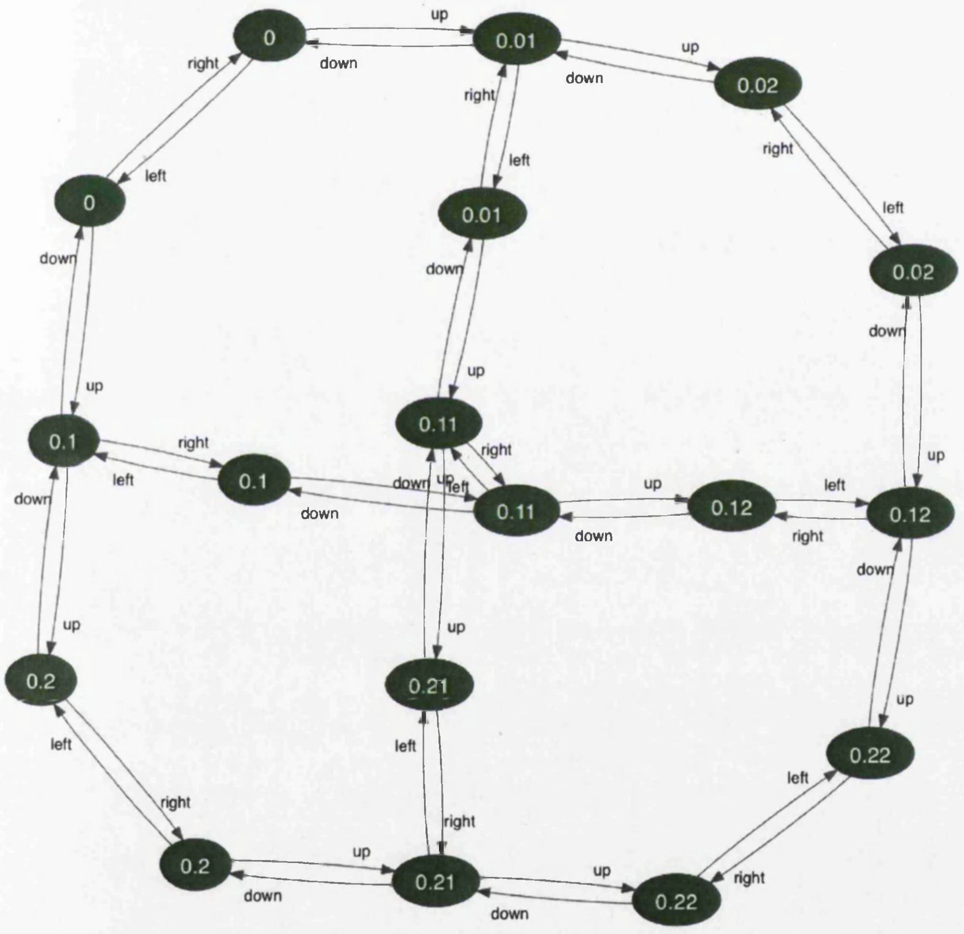


Figure 4.11: Tenths & hundredths, digits 0-2 only, Simple Spinner *and* Simple Arithmetic

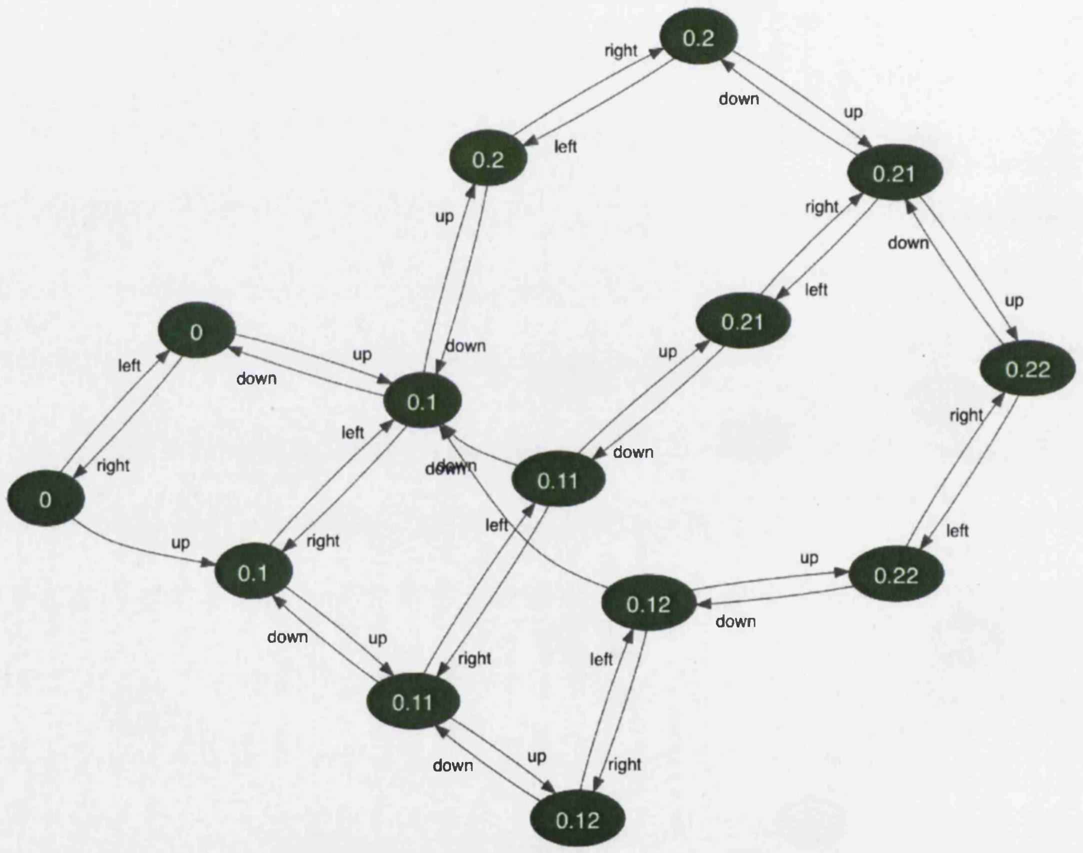


Figure 4.12: Tenths & hundredths, digits 0-2 only, BBraun v686E

```
var vtbidcHundredthsWith9 = {
  name : "hundredths with 9",
  comments : "Tenths, hundredths only; digits 0-2 and 9 only",
  prefix : "R",
  getGuiActions : function() {
      var cursor = Digits.getCursor();
      var digit = parseInt(Digits.getDigit(cursor));
      var guiActions = [];
      if ([0,1,9].indexOf(digit) > -1) {
        guiActions.push(["up" , VTBIEntry.up]);
      }
      if ([0,1,2].indexOf(digit) > -1) {
        guiActions.push(["down" , VTBIEntry.down]);
      }
      if (cursor < -1) {
        guiActions.push(["left" , VTBIEntry.left]);
      }
      if (cursor > -2) {
        guiActions.push(["right", VTBIEntry.right]);
      }
      return guiActions;
  },
  validState : function(state) {
    return Array.checkAllDigits(state.digits, [0, 1, 2, 9]);
  }
};
```

Figure 4.13: A *VTBIDiscovery* configuration object from `VTBIDiscovery.js` — exploring only the digits $\{0,1,2,9\}$ in the tenths/hundredths range.

the BBraun operates around *maximum* values).

This example illustrates immediately that not only can model discovery automatically probe the details of corner cases, but that often visual inspection of the models can yield immediate insights. The models in this case are small enough that we can readily take in their entirety, but as we will see, even with larger models it is possible to get a 'big picture' sense of the difference between the routines. Of course, more detailed analysis such as theorem discovery or model checking can then provide deeper and more certain insights than this initial impression allows.

4.3.3.2 Tenths, hundredths only; digits 0-2 and 9 only

Figure 4.13 shows a configuration aimed at extending the previous example to include exploration of the 9 digit. There are two key differences from figure 4.10:

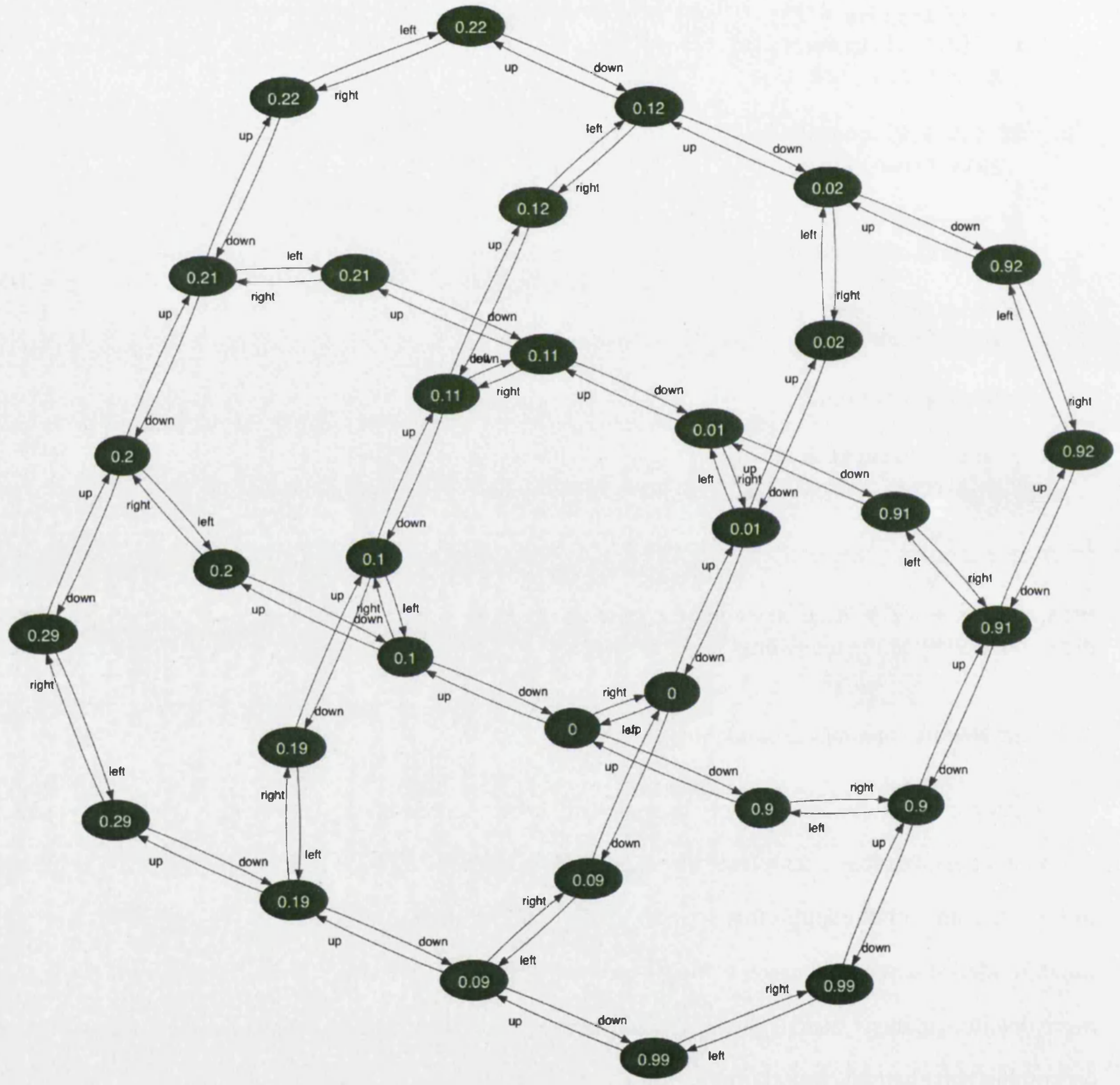


Figure 4.14: Tenths & hundredths only, digits 0-2 and 9 only, Simple Spinner

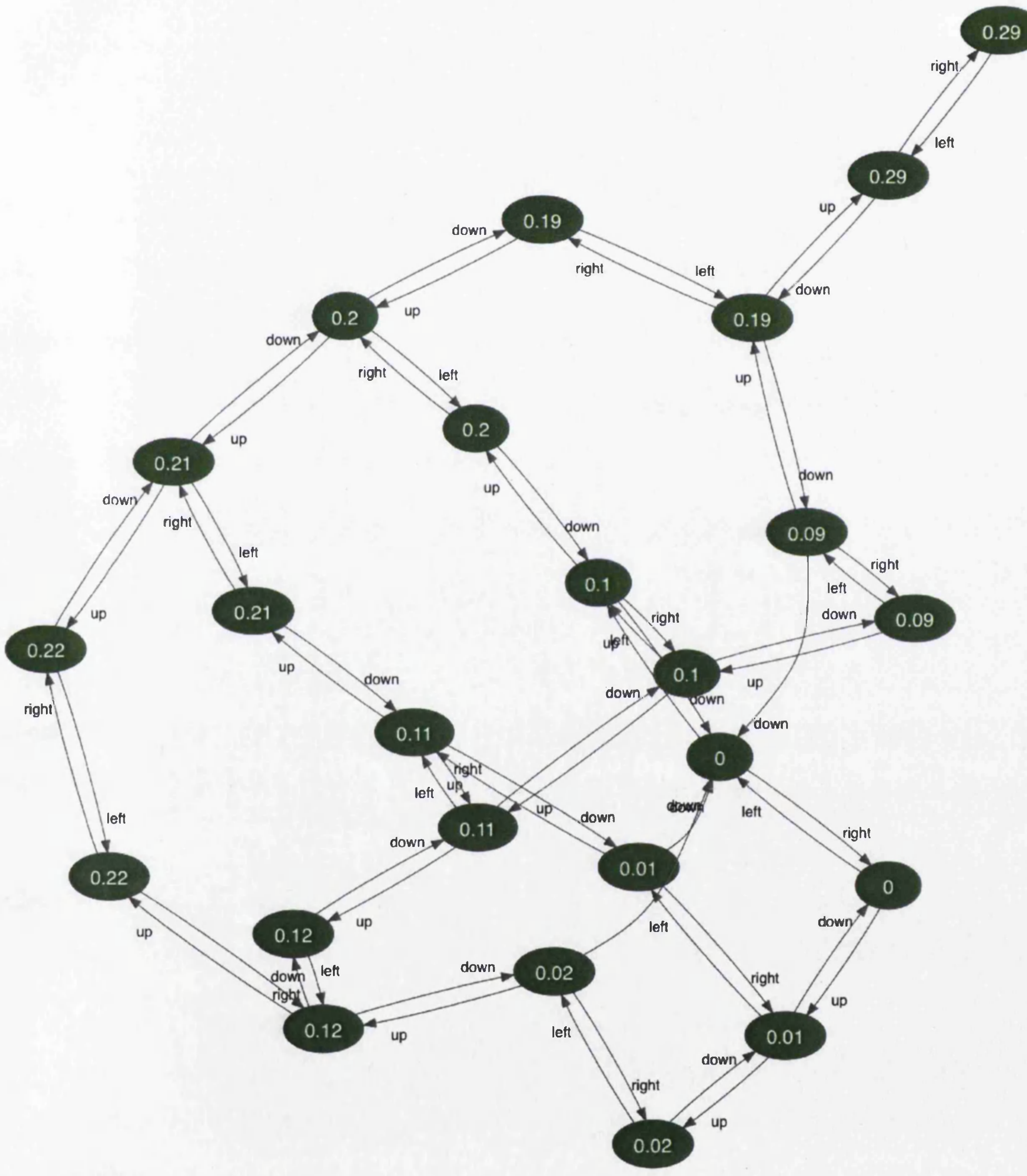


Figure 4.15: Tenths & hundredths only, digits 0-2 and 9 only, Simple Arithmetic

- The guards on addition of *up* and *down* have been modified so that *up* can also be pressed if the digit under the cursor is 9, and *down* can also be pressed if the digit under the cursor is 0; this allows vertical wraparound in the Simple Spinner routine, and arithmetic operations in the Simple Arithmetic and BBraun routines.

- It includes a *validState* implementation, filtering out states whose digits are not in the set $\{0,1,2,9\}$ (using a purpose-built utility function, *Array.checkAllDigits*). Without this, *getGuiActions* on its own is not in fact strong enough to limit our state space to the area we're interested in — see below.

The results of performing this model discovery task on the three number systems are shown in figures 4.14, 4.15 and 4.16, respectively.

Figure 4.14 once again clearly displays the regularity of the Simple Spinner implementation: a highly symmetrical, regular structure; the structure is very similar to the one seen in figure 4.11, but with the addition of 14 new states for the values 0.09, 0.19, 0.29, 0.90, 0.91, 0.92, and 0.99.

Figure 4.15 shows that in this configuration, the Simple Arithmetic routine behaves differently from the Simple Spinner — as we would expect; its behaviour is still fairly regular, except for the existence of two isolated states that deal with exceptional cases, i.e. for the value 0.29 in the top-right of the model. The 'innermost' of these states represents the display $\texttt{0.}\underline{\texttt{2}}\texttt{9}$, and the other is for $\texttt{0.2}\underline{\texttt{9}}$. Looking at these states, there are two obvious questions:

1. Why is there no *up* action edge from the innermost 0.29 state? The answer is simple: the display here is $\texttt{0.}\underline{\texttt{2}}\texttt{9}$, i.e. the cursor is on 2, and *getGuiActions* blocks *up* in that case.

2. Why is there no *up or down* edge leading from the outermost 0.29 state? Here the display is $\texttt{0.2}\underline{\texttt{9}}$, so *getGuiActions* rightly blocks *down*, but it should allow *up*. The answer is that an *up* action is indeed performed in that state, but it leads to a display of $\texttt{0.3}\underline{\texttt{0}}$, which is blocked by the *validState* guard: that state is not added to the state space, so there is no edge leading to it. This will be further discussed below.

Finally, figure 4.16 shows the model for the BBraun under this configuration; as it happens, visual comparison with figure 4.12 is enough to see that this is a simple extension of that state space, with just

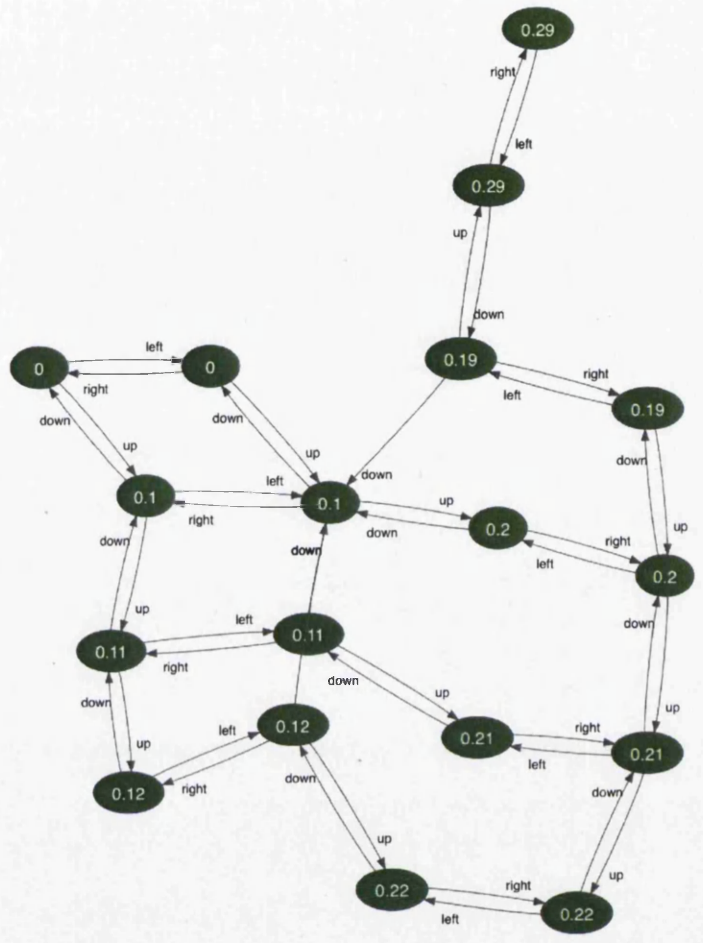


Figure 4.16: Tenths & hundredths only, digits 0-2 and 9 only, BBraun v686E

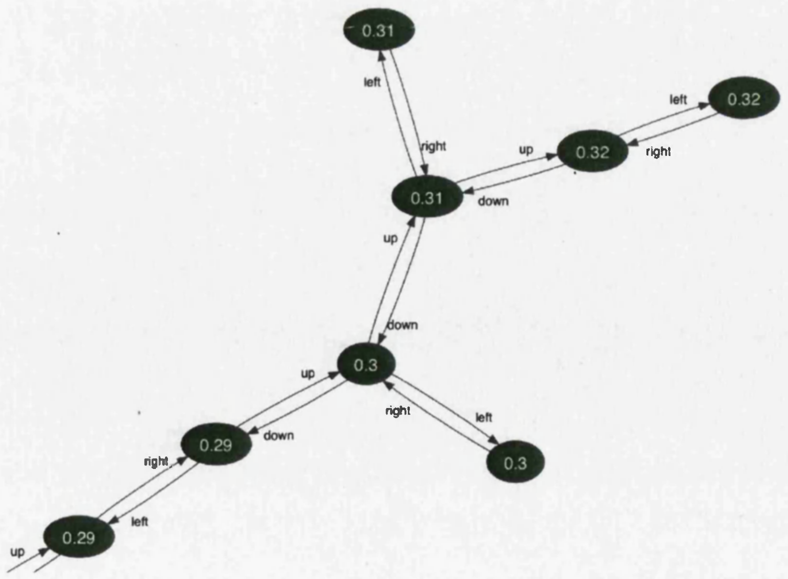


Figure 4.17: Erroneous parts of 'broken' version of previous example (seen in both Simple Arithmetic and BBraun routines).

four more states, for the values 0.19 and 0.29. (In contrast, while figure 4.15 is, similarly, an extension of figure 4.11, that is much harder to see in that case because of the greater number of states.) And, like with the Simple Arithmetic example, we have the irregular protrusion of two states for 0.29, with the same explanation.

To conclude this example, we illustrate the importance of the *validState* mechanism for conditional exploration by considering what happens if it is omitted in this case. It happens that the Simple Spinner case remains unchanged, i.e. *getGuiActions* is a sufficient restriction, but the Simple Arithmetic and BBraun cases both get (the same set of) new states, as shown in figure 4.17. This shows just one part of the model, starting with the innermost 0.29 state, and includes states for the values 0.3, 0.31, and 0.32. As has already been explained, if the display is 0.29, an *up* action leads to 0.30; here the current digit is 0, so we explore *up* and *left* leading to states for displays 0.3 and 0.31, etc. This example illustrates that conditional exploration can be a subtle matter, and requires careful consideration of the behaviour of the system being explored — and that model discovery is thus necessarily something of an iterative, and interactive, process, much like programming itself.

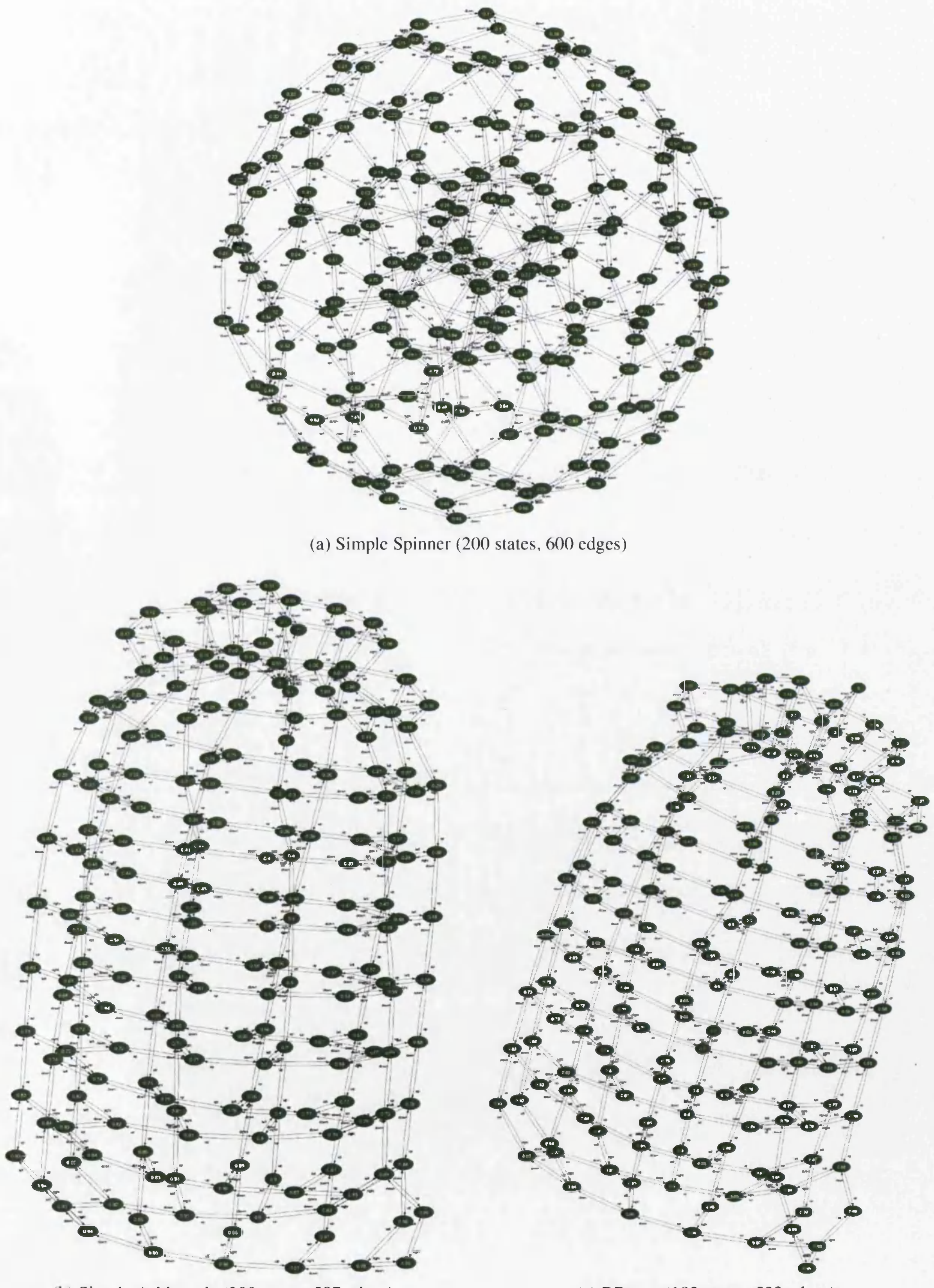
4.3.3.3 Hundredths, full

Figure 4.18 shows the models produced by exploring the full state space of tenths and hundreths, allowing all digits (but only including values less than 1.00, via *validState*). Clearly at this level of complexity, visual inspection of these models can only provide indicative insight, but three things stand out:

1. The Simple Spinner retains something of its regularity at this scale; it's not as obvious as on the smaller models, but it's certainly clear in comparison to the other two models; this is underlined by noticing that it has the full set of 200 states (100 possible values, 2 cursor positions each), and 4 actions per state.

2. The Simple Arithmetic model is clearly less regular, particular around the 'top' of the model (closer visual inspections shows that this is the area around 0); furthermore, it has thirteen less edges than the Simple Spinner.

3. The BBraun model is similar in overall impression to the Simple Arithmetic Model, but it has



(a) Simple Spinner (200 states, 600 edges)

(b) Simple Arithmetic (200 states, 587 edges)

(c) BBraun (182 states, 533 edges)

Figure 4.18: Tenths and hundredths only: full state spaces (i.e. all digits).

less states and edges (due to unreachable states around minima).

At this scale there is little more to say, but these models are ripe for further analysis using, say, social network analysis [TO09], a model checker or our theorem discovery tool — and we shall return to this example later.

4.3.3.4 Ceiling

We conclude this case study with a brief examination of its behaviour around the maximum attainable value, i.e. 99999, partially in order to draw out a limitation to our basic form of model discovery which will be addressed in the case study in section 4.5. The configuration is shown in figure 4.19. Its *prefix* value means model discovery starts at the value 80000; only the digits 0, 8 and 9 are explored, and only cursor values 3 and 4 (thousands and tens-of-thousands respectively); a *validState* guard is used to remove any 'stray' states with any other digits that might be discovered.

The results of this exploration are shown in figure 4.20.

Figure 4.20(a) shows the discovered model for the Simple Spinner number entry routine; it has exactly the same structure as figure 4.11 — only the node labels have changed. On reflection, this is unsurprising: in both cases we are exploring a subspace two digits wide and 3 digits 'deep', in a highly regular overall state space; it is, at least, reassursing that the subspaces have the same shape.

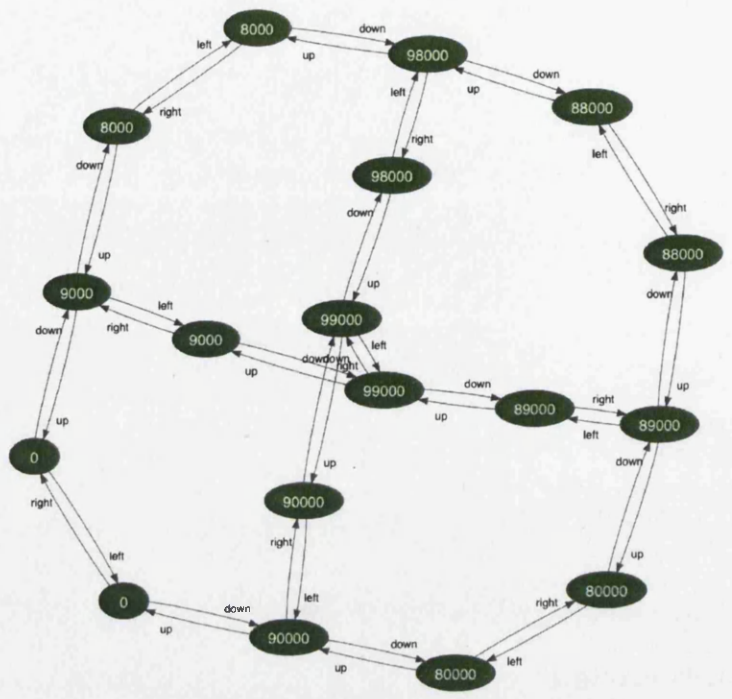
Figure 4.20(b) shows the discovered state space for both the Simple Arithmetic and BBraun routines. One is immediately struck by the symmetry of the model, but on closer inspection, it is not, in fact, quite symmetrical. In particular, none of the nodes in the bottom half of the model that have edges leading *to* 99999 also have edges leading *from* there, whereas in the top half this is not the case (in particular, nodes for 89999 and 98999 have edges in both directions).

This is the case precisely because 99999 is the maximum value, and *up* operations leading to that value are clamped there: so many states have *up* edges leading to a 99999 state — but each of the 99999 states (for two different cursor values) only has one *down* edge leading from it.

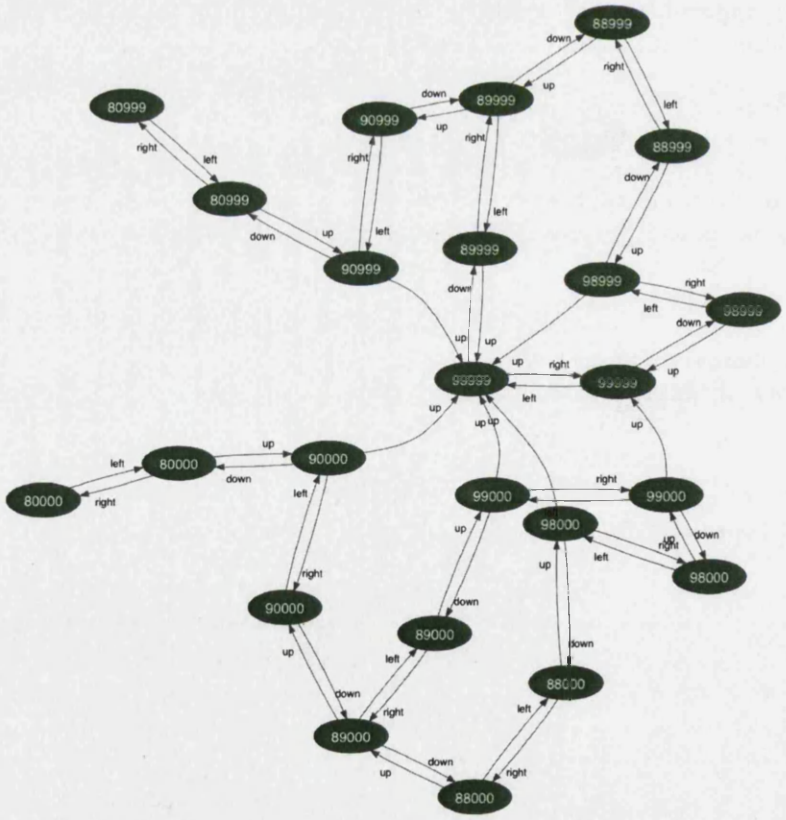
In the case of the Simple Arithmetic routine, this is perfectly correct, and exactly what we would expect; however, in the case of the BBraun routine, a *down* operation on a value of 99999 can in fact lead to one

```
var vtbidcCeiling = {
  name : "ceiling",
  comments : "thousands, tens of thousands; 0, 8, 9 only",
  prefix : "LLLLLUUUUUUUU",
  getGuiActions : function() {
      var cursor = Digits.getCursor();
      var digit = parseInt(Digits.getDigit(cursor));
      var guiActions = [];
      if ([8,9].indexOf(digit) > -1) {
        guiActions.push(["up" , VTBIEntry.up]);
      }
      if ([0,9].indexOf(digit) > -1) {
        guiActions.push(["down" , VTBIEntry.down]);
      }
      if ([3].indexOf(cursor) > -1) {
        guiActions.push(["left" , VTBIEntry.left]);
      }
      if ([4].indexOf(cursor) > -1) {
        guiActions.push(["right", VTBIEntry.right]);
      }
      return guiActions;
  },
  validState : function(state) {
    return Array.checkAllDigits(state.digits, [0, 8, 9]);
  }
};
```

Figure 4.19: Model discovery configuration object: exploring the region around 99999.



(a) Simple Spinner: compare with figure 4.11



(b) BBraun and Simple Arithmetic: note vertical near-symmetry and lack of memory behaviour.

Figure 4.20: Results of model discovery around the region 80000-99999 (0, 8, 9 only).

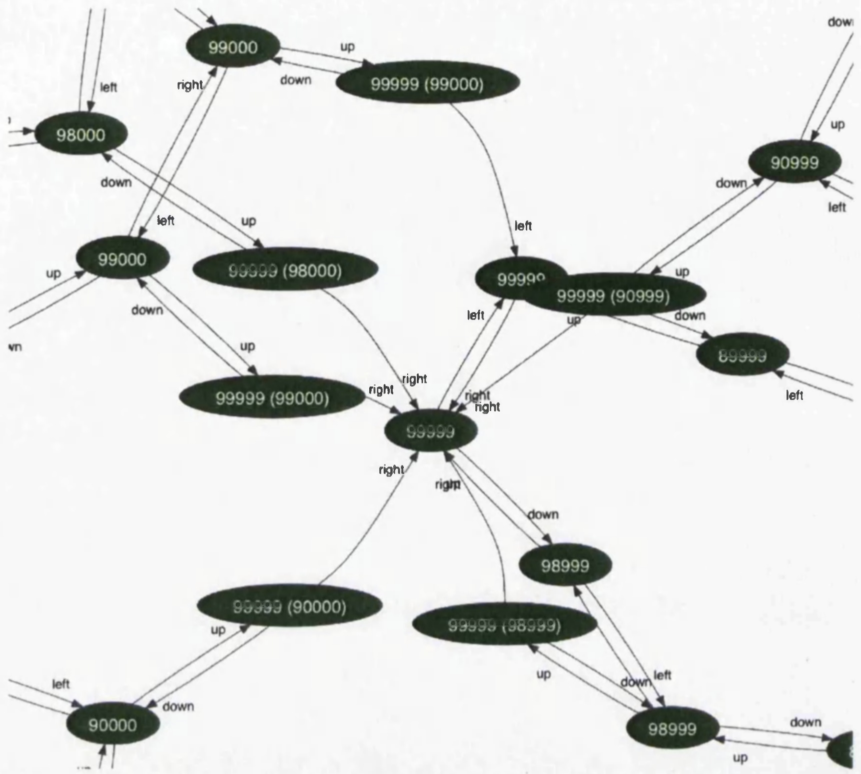


Figure 4.21: Model discovery around the region 80000-99999, with memory projection.

of several different values, depending on the recent history of the device and in particular the status of its *memory* feature (see section 4.3.1.3). The problem is, our model completely fails to take account of this.

There are (at least) two possible responses to this problem:

1. Add the memory's status to the projected state. That is, we can interpret this 'failure' of model discovery as a sign that our chosen projection (figure 4.8) is inadequate. This means rewriting *project* and *reset* in *VTBIDiscovery* and, in fact, also requires that we modify our actual BBraun number entry code, in order to expose the memory status which has been heretofore encapsulated and inaccessible.

   The results of doing so can be seen in figure 4.21, which shows an excerpt of the model thus discovered. Here nodes are labelled not only with the displayed value but also the memory contents in parentheses (if and only if there is anything in memory). Thus we have 'exploded' the two 99999 states from figure 4.20(b) into eight states in this model, six of which have something in memory — and each of those memory-bearing states has a *down* action leading back against its incoming *up* action.

2. Alternatively, it is in fact possible to extend model discovery in a manner that (unlike the basic algorithm) allows for the possibility of multiple identically-labelled edges leading from a particular node; this approach could, if applied here, result in a version of figure 4.20(b) in which each of the two 99999 nodes has several *down* edges leading from it, modelling the system's memory behaviour. We describe this approach, and see it in action, in our final model discovery case study — see section 4.5.

4.4 Case study 3: Alaris GP infusion pump

Thimbleby & Oladimeji [TO09] discuss model discovery and analysis of a simulation they wrote of the Alaris GP infusion pump, a device for controlling drug delivery to hospital patients. The simulation and model discovery tool were written in *ActionScript* on the *Flex* platform[8]. Here we briefly comment on their implementation and its relation to our formalisation of model discovery. The model discovery algorithm is not encoded directly (their work predates the formulation given in this thesis) but its essential strategy is followed. There are two main differences.

First, the *Pool* is a queue of unexplored states, with all of a state's actions explored in sequence, per state — a quite natural way to implement breadth-first discovery.

Second, the tool will actually explore some of the actions in each state *twice*, redundantly. In a given state, the tool first makes a pass where it explores every one of a set of hard-coded actions in order to see which ones actually change the state. Then it makes a second pass, in which only those actions that lead to a new state are explored, and it is *this* exploration that is used to build the model. Thus, each 'useful' button is pressed twice per state. It is easy to see how such an inefficiency could arise in an *ad hoc* implementation of model discovery; of course, it would be avoided by following our algorithm more directly.

The result of *getGuiActions*, i.e. the list of all *possible* button presses, is hard-coded; we suggest that in

[8] `http://www.adobe.com/products/flex`

many cases — particularly for research purposes — this is a reasonable approach, but that for full applicability and integration into software development workflow, the more general approach advocated in this thesis is clearly beneficial. Then *doGuiAction* is simply implemented via Flex's *Button.performClick* method.

An SUD-specific inner state representation was used: a flat bundle of some 20 strings and booleans (e.g. *volumeUnit*, *pressureLevel*, *infusionMethod*), with *eq* defined naturally and no restriction (in any of the experiments they performed) on which variables were projected.

The SUD and the models produced in this example are structurally more complex than in the other three case studies considered in this chapter. In particular, in this example model discovery experiments targetted not only number entry (as in case study 2), but also the device's menu structure and usage workflow. This example thus demonstrates that model discovery is applicable to structurally complex systems; one of the figures in [TO09] shows how the device's mode structure may be exposed in a post-processing step by taking a discovered model and removing the *Off* state from it, leading to a fairly hierarchical view of the device's overall operation.

One consequence of the SUD's complexity, with many dependent variables, more untracked aspects of state, and a bigger state space, is that model discovery is quite slow — a situation compounded by Flash's slower running speed than natively compiled Haskell, and the implementation inefficiency mentioned above.

The SUD involves number entry, so for some analyses, an ad-hoc version of conditional exploration (section 3.4.3.2) was used, with some numerical variables (e.g. 'flow rate') restricted to subsets of their possible values using simple bounds. This helped keep the state space (and model discovery) tractable, which was important not only for the model discovery process itself, but also for the subsequent analyses performed on the models; different restrictions were used at different times, depending on what was being investigated, by modifying model the discovery code directly.

4.5 Case study 4: Casio HS-8V

The Casio HS-8V (figure 4.22) is a popular desktop calculator of a very familiar kind: it has a keypad for entry of decimal digits, a screen with space for numbers up to eight digits long, basic arithmetic operations such as addition and multiplication, and a simple memory facility. Devices like this are common in healthcare where they are routinely used for drug dosage calculations, and so their correct use and operation is clearly critical; as such they are potentially interesting examples for researchers concerned with general methods of improving the safe use of interactive devices. Note that there is nothing particularly special about the HS-8V *per se*, and we do not seek to single it out for attention in and of itself: we could just as easily be talking about any one of a wide range of desktop and scientific calculators, from many different manufacturers including Canon, Citizen, HP, and Sharp, and generic brands; the HS-8V is simply the instance we happened to study in detail first.

Harold Thimbleby wrote an HS-8V simulation using Mathematica, and implemented model discovery against that simulation. This particular instance of model discovery is interesting for two reasons, both of which we will consider further in this section:

1. It is an example of very focused conditional exploration aimed at keeping the discovered model at a tractable size while still providing valid and interesting insights into the device's operation. (In this regard, it is similar to our second case study, in section 4.3.)

2. More uniquely, it uses a very interesting extension of the basic model discovery algorithm that has the effect in one case of producing a nondeterministic model. (In section 6.5 we revisit this example, and describe a particularly interesting and novel insight into the device's operation arising from that nondeterministic model.)

The full state space for the calculator is about 10^{17} states, as there is a display of 8 decimal digits, a memory of 8 decimal digits, and a few modes. This is too large to generate an explicit model for, and such a model would anyway be computationally expensive to analyse — certainly beyond our current theorem discovery implementation, and also probably beyond any of the techniques described in [TO09], though a model checker might be able to handle it. Instead, model discovery was focused on a particular aspect of interest, i.e. number entry, by ignoring the arithmetical/operational keys ($\boxed{+}$, $\boxed{=}$, $\boxed{\text{M+}}$,

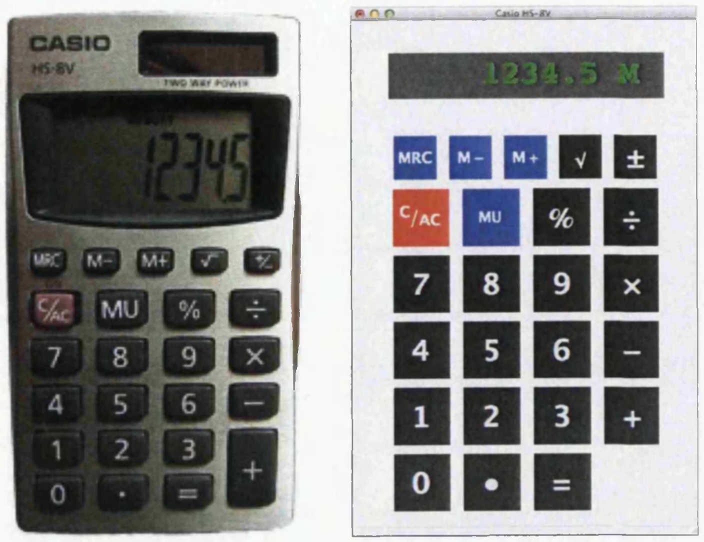
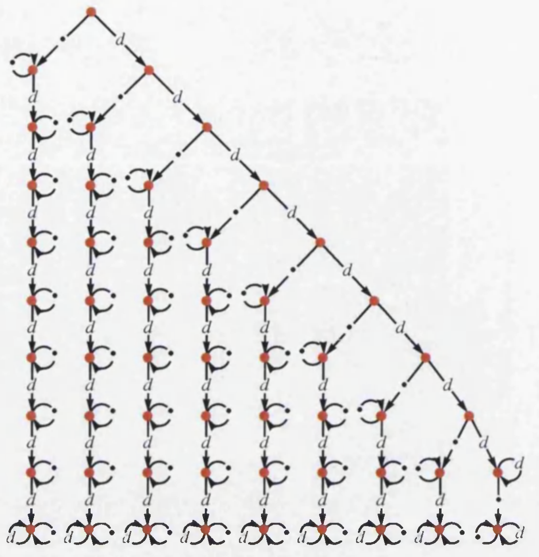


Figure 4.22: Casio HS-8V: actual device and simulation

etc.). Going further, there is good reason to believe that every non-zero digit behaves the same way, and therefore the key $\boxed{1}$ is used as a proxy for all non-zero digits (displayed in figure 4.23 as *d*). The final alphabet of user actions for model discovery in this example thus consisted of only $\boxed{1}$ and $\boxed{\bullet}$. (Thimbleby also produced and analysed some models where $\boxed{0}$ and $\boxed{\text{AC}}$ were included, but we do not discuss them here.)

Thimbleby produced two models, distinguished by the state projection used, and pictured in figure 4.23. In the first, which we call the *device* model, the full internal state of the simulation is projected, including not only the display but also a value indicating whether the $\boxed{\bullet}$ key has been pressed recently; in the second, which we call the *user* model, only the display contents are projected — what the user can see. (In contrast with the more common use of this term, we do not present this as a model of the user's behaviour; we have simply chosen this term to denote that this is what the user can know about simply by inspecting the device.)

The device model (figure 4.23(a)) is a fairly standard product of model discovery as described in this chapter — and in particular it is deterministic; probably the most interesting thing to note about it (without doing further analysis — see section 6.5.2) is that it contains self-loops: after the $\boxed{\bullet}$ key has been pressed once, subsequent presses of that key have no effect (see each left-hand branch in the figure).


(a) 'Device' model: display/decimal projected.

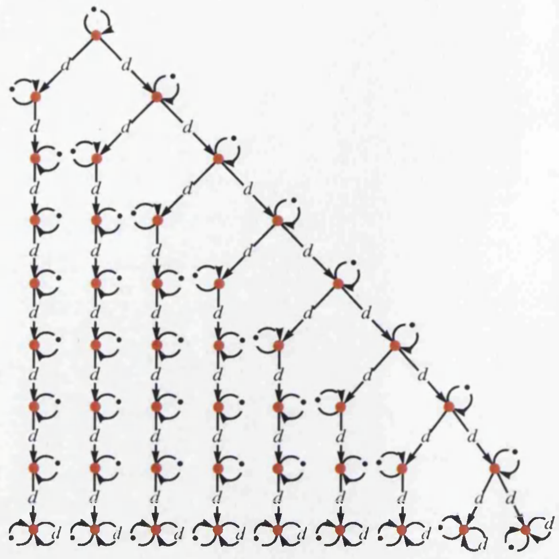

(b) 'User' model: only display is projected.

Figure 4.23: Two models of number entry on the Casio HS-8V desktop calculator.

The user model (figure 4.23(b)) is more immediately interesting, however, as it is *nondeterministic*. For example, consider the root state, at the top of the figure: we have a $\boxed{\bullet}$ labelled self-loop (i.e. the decimal point doesn't change the display), and *two* edges labelled *d* (represented a $\boxed{1}$ keypress). Now, as described in section 3.4.2, the basic model discovery algorithm *cannot* produce a nondeterministic model — so what's going on here?

The basic model discovery algorithm is unable to produce nondeterministic models for the following two reasons:

1. Each state is only *discovered* once: if any subsequent actions lead to that same state, it will be recognised as such (using the equivalence check, *eq*), and a new edge will be added to the model, but *not* a new state. The only way this can 'fail' is if *eq* is too weak, and fails to distinguish two states that really *should* be distinguished, in which case the projection used should be reconsidered — but even if that is the case, you don't get the same state added to the model twice: you get two different states added that ought to be unified. In any case, the upshot is that each state in the model is only discovered once, and (critically for our current discussion) *that* means the set of each state's actions are only added to the pool once.

2. When the set of a state's actions are added to the pool, they're each only added once. This is an

obvious design choice: you discover a state, you list its actions, and you add each action to the pool — obviously only once, because unless our projection is broken, multiple explorations of the same edge are redundant. (As such, it might in fact be worth adding multiple copies of the same action to the pool, as a sanity check: if multiple explorations of the same action *do* lead to different states, that would tend to indicate that the state projection used is insufficiently rich to allow proper backtracking, and some aspect of the system's inner state is causing problems. However, that's not what's happening here.)

In the case of Thimbleby's HS-8V model discovery, this last design choice has been subverted. The novel extension is to add to the pool some (state, action) pairs whose actions are *compound*, i.e. sequences of atomic user actions, where previously only atomic actions have been used.

In particular, it is hard-coded into the model discovery implementation that at every state, *four* actions are to be explored:

- $\boxed{1}$
- $\boxed{\bullet}$
- $\boxed{1}\boxed{\bullet}$
- $\boxed{\bullet}\boxed{1}$

The first two are the simple atomic actions we would expect to explore anyway; the last two are the new compound actions. So from each state, model discovery investigates the effect not only of hitting $\boxed{1}$, but *also* the effect of hitting $\boxed{\bullet}$ followed immediately by $\boxed{1}$. The effects of these compound actions appear in the model in 'small step'/unfolded form, where every intermediate state/edge visited during the compound action is added to the model (if not already present) — so there are no edges labelled $\boxed{\bullet}$ $\boxed{d}$, for example.

Now we can understand the nondeterminism visible in figure 4.23(b). Consider again the root state, at the top of the figure:

- The edge labelled *d* going to the left represents the (second step of the) $\boxed{\bullet}\boxed{1}$ compound action: the $\boxed{\bullet}$ keypress induces the self-loop on the root state (as does exploration of the $\boxed{\bullet}$

atomic action), and then the *d* edge leads to a branch in which $\boxed{\bullet}$ never has any further effect. (Note that there is no visual distinction, in the figure, between edges induced by atomic actions, and edges induced by compound ones.)

- The edge labelled *d* going to the right represents the $\boxed{1}$ case: here, $\boxed{\bullet}$ is yet to be pressed. (It also, thus, represents the first step of $\boxed{1}\boxed{\bullet}$ action.)

This pattern is repeated to the right: *each* left-hand branch is a world where $\boxed{\bullet}$ has just been pressed, and only $\boxed{1}$ has any effect on the model; and each right-hand branch is a world where $\boxed{\bullet}$ is yet to be pressed.

Without compound actions, model discovery as described in [GT10] and chapter 3 could not possibly produce such a model, as it only allows one outgoing edge per action per state.

That a compound action (such as, in this case, $\boxed{\bullet}\boxed{1}$) adds a new state or edge to the model during the model discovery process should, arguably, raise a warning: it indicates either that the projection being used is not rich enough (as described above), or that an observer (e.g. the end user) cannot predictably differentiate between states that should rightfully be distinct as is the case here. In either case, it would be worth making the investigator aware of this as soon as it is noticed.

In the case of the HS-8V user model, this non-determinism is bound up in the fact that $\boxed{\bullet}$ never changes the display, as exposed to us via theorem discovery and considered further in section 6.5 — but an earlier warning, raised during model discovery, could have been useful.

Chapter 5

Theorem Discovery

Contents

5.1 Introduction

In this chapter we present the second of the two techniques at the core of this thesis: **Theorem Discovery**. Whereas the first technique, Model Discovery, targets the problem of *producing* a model of an interactive system, Theorem Discovery targets the problem of *analysing* such models. In particular, theorem discovery automatically and systematically seeks sequences of user (or system) actions that are in some sense — that we define precisely — equivalent in their effects on the system, or nearly so. The core operations, which we describe in detail, are:

1. To compute sequences of actions.

2. To compute the effects of each sequence of actions.

3. To compare those effects with each other.

By doing so, theorem discovery produces two primary output kinds:

1. *Total equivalence theorems* — where two or more action strings are found to have identical effects on the system, throughout its entirety.

2. *Partial equivalence theorems* — where two or more action strings are found to have very similar effects on the system, i.e. identical up to some threshold.

Such theorems are, we argue, interesting because they can embody salient usability properties arising from the system's logical structure. For example, total theorems can embody redundancy, shortcuts, and undo-like behaviour (we consider examples in the next chapter). Conversely, partial theorems represent sequences of actions that are *nearly* equivalent — which can be potential sources of confusion for the user, who may believe they are exactly equivalent, unaware of the divergence.

These 'atomic' theorem outputs are often individually interesting, but may also lead to the discovery of *'metatheorems'*, arising from the interpretation of the combination of a set or *family* of atomic theorems together. The systematic production and interpretation of metatheorems is beyond the scope of this thesis and a matter for future work; here we lay the ground work upon which metatheorems may be built, as well as considering some small examples and proposing some avenues for specific future work in this direction.

The approach to theorem discovery presented in this thesis was first described in [GTC06], within the context of a formal model of inconsistency-related mode confusion; there, the emphasis was on identifying modes and ensuring consistent behaviour within them, and what we refer to here as Theorem Discovery was one component of that approach. The contributions in relation to theorem discovery in this thesis are:

- To generalise the equivalence-finding algorithm beyond the context of that paper to a more general standalone technique, to formalise that in detail, and to implement it (the implementation is described in section 6.2).

- To extend that formalisation, algorithm, and implementation to include partial equivalence (also implemented).

- To begin exploring meta-theorems and the interpretation of families of theorems (section 5.5).
- To provide case studies (in chapter 6) of theorem discovery in action on a range of systems, including ones of real-world complexity.

Thus, we make the technique described in [GTC06] more rigorous and more general, we show it working in realistically complex analyses, and we pave the way for a systematic approach to the application and exploration of the technique.

The scope of our current approach is as described in section 3.1.1: reactive systems with discrete interfaces and finite state spaces, subject to some assumptions (described in that section). The key idea is that a user performs actions and thereby changes the state of the system they are using; we assume the user's actions are discrete, for example pressing buttons. Variations such as holding a button down for 2 seconds, say, may be modelled as different discrete actions and many continuous actions can be approximated as a sequence of discrete actions (e.g. turning a knob could be represented by the two actions "turn right 1 degree" and "turn left 1 degree"); similarly, internal (i.e. non-user) actions such as clock ticks may be modelled and treated identically with user actions, without loss of generality.

Our models, as described in chapter 3, are directed graphs whose nodes represent (sets of) system states, and whose edges represent user actions. This class of model is easily comprehended, and is not particularly specialised or esoteric. Such models may be readily used to gain real insights into UI behaviour that might otherwise go unnoticed, as we shall show. (We believe, however, that the core idea of theorem discovery could be fruitfully applied to other classes of model, including ones with richer structure such as statecharts; doing so is beyond the scope of this thesis.)

A note on terminology: the discussion in this chapter concerns related entities in three separate domains, namely the interactive system whose model is being investigated, the model of that system (i.e. a graph as just described), and the formalism found in section 5.3. The following table relates the terms used in each of these domains to discuss related concepts:

System/UI	Graph model	Formalism
Interactive system	Graph	Finite state machine
User action (or just 'action')	Edge	Action / symbol
System state / set of states	Node	State
Sequence of user actions	Path	String

5.2 Informal discussion

As noted above, the core concept of theorem discovery is as follows: given a directed graph that notionally represents a model of some user interface (i.e. its nodes are system states and its edges are user actions), we systematically compute *strings* of actions, and the *effects* of those strings over the whole model, and look for equivalences between the effects (and thus the action strings). The key question is: what exactly do we mean by the 'effect' of a string of actions 'over the whole model'? That is, how are such effects to be represented, computed and compared? In this section we provide an informal overview of our solution to this problem; in section 5.3, we present a formal treatment.

5.2.1 Effects

The effect of an action is considered to be *where it leads to in the model*, i.e. in the graph. In any particular given state, the effect of an action is easily conceived: if a user performs that action while the system is in that state, what state does the system end up in? In terms of the graph, we can state that as follows: in a given node, if we follow the edge(s) labelled with a given action, what node(s) do we end up in? The effect *across the whole system*, then, is simply the aggregation of those per-state effects, indexed somehow by state (node). (For the rest of this discussion, we generally prefer the terms 'state' and 'action' to 'node' and 'edge'.)

For example, consider the simple model in figure 5.1; we have four states labelled, $\{0,1,2,3\}$ and an alphabet of two actions $\{x,y\}$. Writing $[\![x]\!]$ and $[\![y]\!]$ for the effects of the actions x and y, we can represent

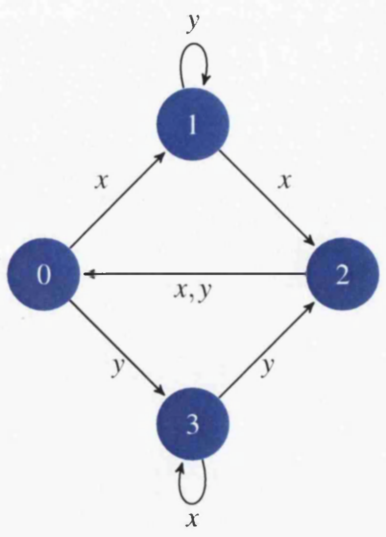


Figure 5.1: A simple model, with four states and two actions, x and y

each effect at each state as a function from source state to destination state:

$$\begin{aligned}
[\![x]\!](0) &= 1 \quad , \quad [\![y]\!](0) = 3 \\
[\![x]\!](1) &= 2 \quad , \quad [\![y]\!](1) = 1 \\
[\![x]\!](2) &= 0 \quad , \quad [\![y]\!](2) = 0 \\
[\![x]\!](3) &= 3 \quad , \quad [\![y]\!](3) = 2
\end{aligned}$$

Then (as our states are numbered incrementally from 0) an effect over the whole model can be simply written as an ordered list of destination states, where source state is encoded as list position:

$$[\![x]\!] = \langle 1,2,0,3 \rangle \quad , \quad [\![y]\!] = \langle 3,1,0,2 \rangle$$

Other representations are possible, of course — and below we extend this notation for the nondeterministic case. [Thi04a] explores the use of another representation, namely adjacency matrices:

$$[\![x]\!] = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} , \quad [\![y]\!] = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

While these have some appealing properties, as explored in that paper, in this thesis we favour the list based representation, as it is more compact and arises naturally out of the formalisation given in section 5.3.

5.2.2 Nondeterminism

The list-of-destination-states representation given above is only viable in this example because the model happens to be deterministic, so there is only one destination state for each action in each source state. In general we need to allow for the possibility of nondeterministic models, with multiple destination states for each action in each source state. (Graph-theoretically, we need to allow for multiple identically-labelled edges leading from the same node.)

Thus, in general, we write an effect as a list of *sets* of destination states:

$$[\![x]\!] = \langle\{1\},\{2\},\{0\},\{3\}\rangle \quad , \quad [\![y]\!] = \langle\{3\},\{1\},\{0\},\{2\}\rangle$$

Where it is clear that we are dealing with a deterministic model, the non-set version is preferred for reasons of clarity.

5.2.3 Strings of actions, and their effects

All of these representations may be used not only to represent the effects of atomic actions, but also to represent the effects of *strings* of actions. In each case, the representation of a string's effect is easily computed, given the representations of its component actions. For example, given the list representations of the effects $[\![x]\!]$ and $[\![y]\!]$ above, it is easy to compute the effect $[\![xy]\!]$ by composing $[\![x]\!]$ and $[\![y]\!]$ at each state:

$$\begin{array}{lcl}
[\![x]\!](0) = 1 \;\wedge\; [\![y]\!](1) = 1 & \Rightarrow & [\![xy]\!](0) = 1 \\
[\![x]\!](1) = 2 \;\wedge\; [\![y]\!](2) = 0 & \Rightarrow & [\![xy]\!](1) = 0 \\
[\![x]\!](2) = 0 \;\wedge\; [\![y]\!](0) = 3 & \Rightarrow & [\![xy]\!](2) = 3 \\
[\![x]\!](3) = 3 \;\wedge\; [\![y]\!](3) = 2 & \Rightarrow & [\![xy]\!](3) = 2
\end{array}$$

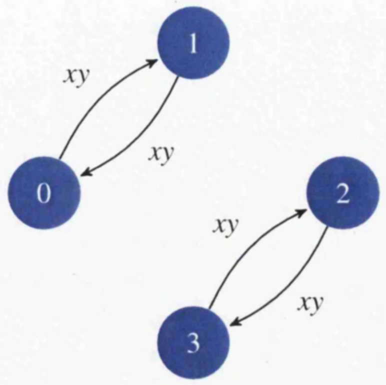


Figure 5.2: The effect of the action string xy

So xy takes you from 0 to 1, from 1 to 0, etc., as is easily checked by inspection. Then, in the concise list notation:

$$[\![xy]\!] = \langle 1, 0, 3, 2 \rangle$$

Another view of this computation is to take the list representations of $[\![x]\!]$ and $[\![y]\!]$, i.e. $\langle 1,2,0,3 \rangle$ and $\langle 3,1,0,2 \rangle$, and to use the components of $[\![x]\!]$ as indices into $[\![y]\!]$:

$$\begin{array}{rcccccccccc}
[\![xy]\!] & = & \langle & [\![xy]\!](0) & , & [\![xy]\!](1) & , & [\![xy]\!](2) & , & [\![xy]\!](3) & \rangle \\
 & = & \langle & [\![y]\!]([\![x]\!](0)) & , & [\![y]\!]([\![x]\!](1)) & , & [\![y]\!]([\![x]\!](2)) & , & [\![y]\!]([\![x]\!](3)) & \rangle \\
 & = & \langle & [\![y]\!](1) & , & [\![y]\!](2) & , & [\![y]\!](0) & , & [\![y]\!](3) & \rangle \\
 & = & \langle & 1 & , & 0 & , & 3 & , & 2 & \rangle
\end{array}$$

This is, in fact, how our implementation computes such compositions (generalised to the nondeterministic set-of-destinations representation described above.)

This effect, $[\![xy]\!]$, is illustrated in figure 5.2. By inspection of that diagram, it is easy to see that xy is its own inverse: two applications of xy in a row have no effect, whatever state you start in — it is the identity action, which we write as $\mathbb{I}$ (see section 5.3.9.1):

$$[\![xyxy]\!] = \langle 0, 1, 2, 3 \rangle = \mathbb{I}$$

This is exactly the kind of thing that theorem discovery can discover automatically.

5.2.4 Total and partial equivalence

Two actions are *totally equivalent* if their effects are identical; obviously checking for such identity is computationally trivial: in the list representation, we simply check the destination (in general, destination set) for each state, and require that they are all the same.

For example, we find that:

$$[\![yyxx]\!] = \langle 1,0,3,2\rangle = [\![xy]\!]$$

and thus xy and $yyxx$ are totally equivalent: whatever state the system is in, performing either of those actions takes the system to the same new state.

Partial equivalence generalises this notion further, by introducing an *effect similarity* metric that gives some measure of how similar two actions are to each other, in terms of their effect; then it is possible to set a threshold — 95%, say — and report all actions whose effect similarity is above that threshold.

How, then, should effect similarity be measured? There are a number of possible ways to compare two effects structurally (e.g. by looking at the symmetric difference of their sets of induced edges), but the notion we are specifically trying to capture here is about *what happens when the action is performed.* As such, we aim to compute the following:

The probability that, given some (e.g. uniformly) randomly chosen state in the system, performing both actions in that state leads to the same destination state.

In the deterministic case, this is very simple: count the number of states where the two actions' destinations are identical, and divide it by the total number of states. This clearly gives us the desired probability. For example, consider x and y again:

$$[\![x]\!] = \langle 1,2,0,3\rangle \quad , \quad [\![y]\!] = \langle 3,1,0,2\rangle$$

Here there is only one state where x and y have the same effect: state 2, where both actions lead to state 0. Thus, picking a state at random, the probability that these two actions lead to the same state is $\frac{1}{4}$.

The nondeterministic case is slightly more complicated: here we can't just count identical destination sets, because by doing so we may be discarding some similarity from our measure. For example, con-

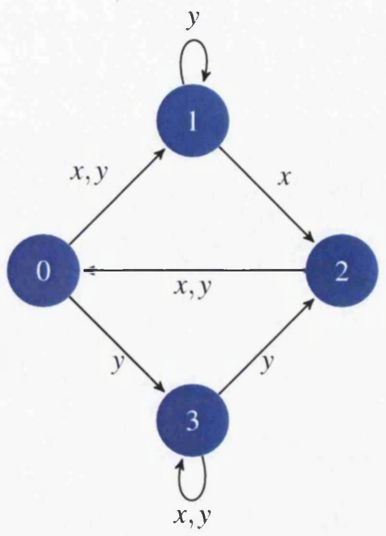


Figure 5.3: A simple nondeterministic model, based on the model shown in figure 5.1.

sider figure 5.3, a slightly modified version of our familiar example from figure 5.1. Now we have:

$$[\![x]\!] = \langle\{1\},\{2\},\{0\},\{3\}\rangle \quad , \quad [\![y]\!] = \langle\{1,3\},\{1\},\{0\},\{2,3\}\rangle$$

...with two additional y-labelled edges, from states 0 and 3. The simple state similarity measure described above still gives us $\frac{1}{4}$ here: only at state 2 are the destination sets identical. However, if we pick a random state, we might in fact now pick one where x and y *could* have the same effect, namely states 0 and 3, with a 50/50 chance in each case. So if we consider the four states in turn, we get the following probabilities (see section 5.3.11.1 for workings):

$$\langle\frac{1}{2},0,1,\frac{1}{2}\rangle$$

Their total is 2, so dividing by the number of states, we get an overall probability, and thus similarity, of $\frac{1}{2}$.

Generalising from this example, we see that the probability at each state is computed by taking the destination sets for each of the two actions, and dividing the size of their intersection by the size of their union. In the above example this indeed gives us the probabilities $\langle\frac{1}{2},0,1,\frac{1}{2}\rangle$, and it is easy to see that this continues to give the right solution as more destinations are added to any of those sets; furthermore, the deterministic case is just the limit of this, where the union is either the empty set or the intersection.

It is worth noting that this interpretation of effect similarity for nondeterministic actions hinges on the exact semantics we choose for nondeterminism in our models. If we are in state 0 in figure 5.3, and perform action y, where does that take us? There are two possible ways to interpret this:

- To *either* state 1 *or* state 3.
- To *both* state 1 *and* state 3.

When computing the effects of action strings, the second interpretation is the proper one, in order to include all possible effects and not throw information away. (This is captured in our formalisation of the *string transition function* in section 5.3.5.) However, when computing effect similarity, the first interpretation is the correct one: ultimately our models are intended to provide insight into the behaviour of a reactive system as it responds to user interaction; in that context, the first interpretation is clearly the right one because when a user presses a button that may have one of two effects on the system, *only one effect actually takes place*. Thus, at each state, we need to compute the probability that randomly choosing a destination state for each of the actions yields the same state — and that probability is as described above.

This measure of effect similarity assumes that states are visited with equal probability, which may not, in practice, be the case. A more sophisticated treatment might weight the probabilities at the various states, informed (say) by the use of a Markov model (e.g. following [TCJ01]) and/or empirically collected usage data. Exploration of such weighting is beyond the scope of this thesis.

We conclude this section with a few words on the algorithm to find total and partial equivalences, which is explained in detail in section 5.4. It simply computes ever-longer action strings and their effects, and compares those effects with the ones previously computed; where total and partial equivalences (modulo some threshold) are discovered, they are reported. The algorithm reduces redundant computation and reporting by grouping totally equivalent actions into equivalence classes by effect, and by not checking strings that have a prefix that has already been the subject of a total equivalence.

5.3 Formalisation

5.3.1 Definition: finite state machine

We model a discrete user interface as a *finite state machine* (FSM): a tuple $(S, \Sigma, \curvearrowright)$ where:

- S is a finite set (of states);
- Σ is a finite set (of symbols) called the system's **alphabet**; and
- $\curvearrowright : S \times \Sigma \rightarrow \mathscr{P}(S)$ is a total function, called the machine's **transition function**.

States correspond to projections of the modelled device's internal state; symbols consist of possible user (or internal) actions; the transition function encodes the behaviour of the modelled device in terms of how actions move about the state set. Defining $\curvearrowright$ as a total function to the powerset of states unifies the treatment of deterministic and nondeterministic machines.

5.3.2 Definition: transition set

Given a state $p \in S$ and a symbol $a \in \Sigma$, we call the set of states in $\curvearrowright(p, a)$ the **transition set** for a at p, written $[\![a]\!](p)$. We introduce an infix shorthand for single **transitions**:

$$\forall\, p, q \in S \bullet \forall\, a \in \Sigma \bullet p \overset{a}{\curvearrowright} q \iff q \in [\![a]\!](p)$$

Here, p and q are called the **source** and **destination** states of the transition, respectively.

5.3.3 Definition: nondeterminism

An FSM $(S, \Sigma, \curvearrowright)$ is **nondeterministic** (an NFSM) if the transition set for some symbol at some state has more than one member; otherwise the machine is **deterministic** (a DFSM). Equivalently, an FSM is nondeterministic if and only if:

$$\exists\, p, q, r \in S \wedge \exists\, a \in \Sigma \bullet p \overset{a}{\curvearrowright} q \wedge p \overset{a}{\curvearrowright} r \wedge q \neq r$$

5.3.3.1 Example: finite state machine

Consider the model shown in figure 5.1. As an FSM $(S, \Sigma, \curvearrowright)$ we have:

$$\begin{aligned} S &= \{\, 0,1,2,3 \,\} \\ \Sigma &= \{\, x,y \,\} \\ \curvearrowright &= \{\, 0\overset{x}{\curvearrowright}1 \,,\, 0\overset{y}{\curvearrowright}3 \,,\, 1\overset{x}{\curvearrowright}2 \,,\, 1\overset{y}{\curvearrowright}1 \,,\, 2\overset{x}{\curvearrowright}0 \,,\, 2\overset{y}{\curvearrowright}0 \,,\, 3\overset{x}{\curvearrowright}3 \,,\, 3\overset{y}{\curvearrowright}2 \,\} \end{aligned}$$

Note the slight abuse of notation in enumerating $\curvearrowright$ here; properly it would written as:

$$\curvearrowright \;=\; \{\, (0,x) \mapsto \{1\} \,,\, (0,y) \mapsto \{3\} \,,\, (1,x) \mapsto \{2\} \,,\, (1,y) \mapsto \{1\} \,,\, \cdots \,\}$$

5.3.4 Definition: string

We adopt the standard definition of **strings** over an alphabet. Specifically, if Σ is an alphabet then Σ^*, the set of strings over Σ is the smallest set such that:

1. Σ^* contains the empty string: $\lambda \in \Sigma^*$

2. $w \in \Sigma^* \wedge a \in \Sigma \implies wa \in \Sigma^*$

5.3.5 Definition: string transition function

The transition function $\curvearrowright : (S \times \Sigma) \to \mathscr{P}(S)$ may be lifted to the **string transition function** (to a **destination set**), $\twoheadrightarrow : (S \times \Sigma^*) \to \mathscr{P}(S)$ as follows:

1. $\forall\, p \in S \bullet \; \twoheadrightarrow(p, \lambda) = \{\, p \,\}$

2. $\forall\, p \in S, w \in \Sigma^*, a \in \Sigma \bullet \; \twoheadrightarrow(p, wa) = \bigcup \left\{\, \llbracket a \rrbracket(q) \;\middle|\; q \in \twoheadrightarrow(p, w) \,\right\}$

Note that this definition takes the view of nondeterministic actions as being 'all taken together' rather than 'just one chosen and taken', as discussed in section 5.2.4.

5.3.5.1 Example: string transition function

Consider the nondeterministic model shown in figure 5.3. To compute $\twoheadrightarrow(3, yxy)$:

$$\begin{aligned}
\twoheadrightarrow(3,yxy) &= \bigcup\left\{ [\![y]\!](q) \,\middle|\, q \in \twoheadrightarrow(3,yx) \right\} \\
&= \bigcup\left\{ [\![y]\!](q) \,\middle|\, q \in \bigcup\left\{ [\![x]\!](r) \,\middle|\, r \in \twoheadrightarrow(3,y) \right\} \right\} \\
&= \bigcup\left\{ [\![y]\!](q) \,\middle|\, q \in \bigcup\left\{ [\![x]\!](r) \,\middle|\, r \in \bigcup\{ [\![y]\!](p) \mid p \in \twoheadrightarrow(3,\lambda) \} \right\} \right\} \\
&= \bigcup\left\{ [\![y]\!](q) \,\middle|\, q \in \bigcup\left\{ [\![x]\!](r) \,\middle|\, r \in \bigcup\{ [\![y]\!](p) \mid p \in \{3\} \} \right\} \right\} \\
&= \bigcup\left\{ [\![y]\!](q) \,\middle|\, q \in \bigcup\left\{ [\![x]\!](r) \,\middle|\, r \in [\![y]\!](3) \right\} \right\} \\
&= \bigcup\left\{ [\![y]\!](q) \,\middle|\, q \in \bigcup\left\{ [\![x]\!](r) \,\middle|\, r \in \{2,3\} \right\} \right\} \\
&= \bigcup\left\{ [\![y]\!](q) \,\middle|\, q \in [\![x]\!](2) \cup [\![x]\!](3) \right\} \\
&= \bigcup\left\{ [\![y]\!](q) \,\middle|\, q \in \{1\} \cup \{3\} \right\} \\
&= \bigcup\left\{ [\![y]\!](q) \,\middle|\, q \in \{1\,,3\} \right\} \\
&= [\![y]\!](1) \cup [\![y]\!](3) \\
&= \{1\} \cup \{2,3\} \\
&= \{1,2,3\}
\end{aligned}$$

5.3.6 Definition: destination set

Given a state $p \in S$ and a string $w \in \Sigma^*$, we call the set of states in $\twoheadrightarrow(p,w)$ the **destination set** for w at p, and we write it as $[\![w]\!](p)$. Note the overloading of notation here: we write $[\![a]\!](p)$ for the transition set induced by an atomic action at a state p, and $[\![w]\!](p)$ for the destination set induced by a string w at that state.

As with single symbols, we introduce an infix shorthand for single **string transitions**:

$$\forall\, p,q \in S \bullet \forall\, w \in \Sigma^* \bullet p \overset{w}{\twoheadrightarrow} q \iff q \in [\![w]\!](p)$$

5.3.7 Definition: destination equivalence

Given a state $p \in S$, two strings are said to be **destination equivalent** at p, written $\sim_p$, if their destination sets at p are identical:

$$\forall\, p \in S \bullet \forall\, w,x \in \Sigma^* \bullet w \sim_p x \iff [\![w]\!](p) = [\![x]\!](p)$$

5.3.7.1 Examples: destination sets and destination equivalence

As shown in the example in section 5.3.5.1, in the model shown in figure 5.3 we have that:

$$[\![yxy]\!](3) = \{1,2,3\}$$

so we can also write:

$$3 \overset{yxy}{\twoheadrightarrow} 1,\ 3 \overset{yxy}{\twoheadrightarrow} 2, \text{ and } 3 \overset{yxy}{\twoheadrightarrow} 3$$

We can also see that $[\![xxy]\!](0) = [\![xyxx]\!](0) = \{\,0\,\}$ and thus we have that:

$$xxy \sim_0 xyxx$$

5.3.8 Definition: effects

Given an FSM $(S, \Sigma, \curvearrowright)$, for every string $w \in \Sigma^*$, w's **effect**, written $[\![w]\!] \subseteq S \times \mathscr{P}(S)$ captures its behaviour across every state p in S, as a set of destination sets for w, indexed by source state:

$$[\![w]\!] = \left\{ (p, [\![w]\!](p)) \,\middle|\, p \in S \right\}$$

An effect is **deterministic** if, for every one of its pairs, the destination set found in the pair's second element has only a single member; otherwise it is **nondeterministic**. We state without proof that an FSM is deterministic if and only if all of its effects are deterministic.

In the common case where S is a contiguous range of integers $0 \ldots n$ for some n, we consider explicitly indexing by state to be redundant, and prefer to write effects using the list shorthand described in section 5.2 (see below for example).

5.3.8.1 Example: effects

Consider again the FSM in figure 5.1. Here are some effects over this machine:

$$\begin{aligned}
[\![x]\!] &= \{\, (0,\{1\})\,,\,(1,\{2\})\,,\,(2,\{0\})\,,\,(3,\{3\}) \,\} \\
[\![y]\!] &= \{\, (0,\{3\})\,,\,(1,\{1\})\,,\,(2,\{0\})\,,\,(3,\{2\}) \,\} \\
[\![xx]\!] &= \{\, (0,\{2\})\,,\,(1,\{0\})\,,\,(2,\{1\})\,,\,(3,\{3\}) \,\} \\
[\![xy]\!] &= \{\, (0,\{1\})\,,\,(1,\{0\})\,,\,(2,\{3\})\,,\,(3,\{2\}) \,\}
\end{aligned}$$

As the set of the states is the contiguous range of integers $0 \ldots 3$, we may omit the explicit state indexing, and instead write (as before):

$$\begin{aligned}
[\![x]\!] &= \langle 1,2,0,3 \rangle \\
[\![y]\!] &= \langle 3,1,0,2 \rangle \\
[\![xx]\!] &= \langle 2,0,1,3 \rangle \\
[\![xy]\!] &= \langle 1,0,3,2 \rangle
\end{aligned}$$

5.3.9 Definition: total equivalence

Two strings are said to be **totally equivalent**, written $\sim$, if their effects are identical:

$$\forall w,x \in \Sigma^* \bullet w \sim x \iff [\![w]\!] = [\![x]\!]$$

5.3.9.1 Definition: identity effect

An action or sequence of actions may in fact do nothing, i.e. (from an operational point of view) it may simply take the system back to whatever state it was in before the action occurred. In such cases, the action has the same effect as the empty string λ, and we write its effect as $\mathbb{I}$.

$$\forall w \in \Sigma^* \bullet w \sim \lambda \iff [\![w]\!] = \mathbb{I} \iff \forall p \in S \bullet [\![w]\!](p) = \{ p \}$$

5.3.9.2 Examples: total equivalence

The full set of total equivalences for the deterministic model in figure 5.1, as computed by our algorithm (see section 5.4) is as follows:

$$\begin{aligned}
xxx &\sim \lambda \\
xyx &\sim yy \\
yxy &\sim xx \\
yyy &\sim \lambda \\
xxyy &\sim yx \\
yxxy &\sim xyyx \\
yyxx &\sim xy
\end{aligned}$$

We have already seen the last of these, $yyxx \sim xy$, in section 5.2.4. In section 5.2.3 we also saw that $xyxy \sim \lambda$ — why isn't that reported here? The answer is that before considering strings of length 4, the algorithm found that $xyx \sim yy$; as such, it doesn't check any strings whose prefix is xyx, as they will trivially induce total equivalences with the corresponding strings whose prefix is yy. In this case, then, 'obviously' $xyxy \sim yyy$ — but the algorithm will never report this. See section 5.4 for more information on how the algorithm prunes its search space.

5.3.10 Definition: similarity

Given two strings $w,x \in \Sigma^*$, their **similarity at state** p, written $\sigma_p(w,x) \in [0,1] \subset \mathbb{R}$, is defined as follows:

$$\forall\, p \in S \bullet \forall\, w,x \in \Sigma^* \bullet \sigma_p(w,x) \;=\; \frac{\left|\, [\![w]\!](p) \cap [\![x]\!](p) \,\right|}{\left|\, [\![w]\!](p) \cup [\![x]\!](p) \,\right|}$$

That is: we compute each string's destination set at that state, $[\![w]\!](p)$ and $[\![x]\!](p)$, and we divide the size of intersection of those two sets by the size of their union, yielding the probability *at that state* that the two actions have the same effect. In the case of deterministic effects, where both destination sets have a single member, this value will always be either 0 or 1.

Then, the **similarity** of the two strings *across the whole model*, written $\sigma(w,x) \in [0,1] \subset \mathbb{R}$, is simply the mean of the similarities at all of the states of the model:

$$\forall\, w,x \in \Sigma^* \bullet \sigma(w,x) \;=\; \frac{\sum_{p \in S} \sigma_p(w,x)}{|\,S\,|}$$

5.3.11 Definition: partial equivalence

Two strings $w,x \in \Sigma^*$ are said to be **partially equivalent with threshold** t, written $w \approx_t x$ if and only if their similarity is greater than or equal to t:

$$\forall\, w,x \in \Sigma^* \bullet \forall\, t \in [0,1] \subset \mathbb{R} \bullet w \approx_t x \iff \sigma(w,x) \geq t$$

5.3.11.1 Examples: similarity and partial equivalence

Here we revisit the example in section 5.2.4: a nondeterministic model (shown in figure 5.3) with four states and two actions:

$$[\![x]\!] = \langle \{1\}, \{2\}, \{0\}, \{3\} \rangle \quad , \quad [\![y]\!] = \langle \{1,3\}, \{1\}, \{0\}, \{2,3\} \rangle$$

To compute $\sigma(x,y)$, the similarity of x and y, we first need to compute their similarities at each state.

$$\begin{aligned}\sigma_0(x,y) &= \frac{|[\![x]\!](0) \cap [\![y]\!](0)|}{|[\![x]\!](0) \cup [\![y]\!](0)|}\\ &= \frac{|\{1\} \cap \{1,3\}|}{|\{1\} \cup \{1,3\}|}\\ &= \frac{|\{1\}|}{|\{1,3\}|}\\ &= \frac{1}{2}\end{aligned}
\qquad
\begin{aligned}\sigma_1(x,y) &= \frac{|[\![x]\!](1) \cap [\![y]\!](1)|}{|[\![x]\!](1) \cup [\![y]\!](1)|}\\ &= \frac{|\{2\} \cap \{1\}|}{|\{2\} \cup \{1\}|}\\ &= \frac{|\emptyset|}{|\{1,2\}|}\\ &= 0\end{aligned}$$

$$\begin{aligned}\sigma_2(x,y) &= \frac{|[\![x]\!](2) \cap [\![y]\!](2)|}{|[\![x]\!](2) \cup [\![y]\!](2)|}\\ &= \frac{|\{0\} \cap \{0\}|}{|\{0\} \cup \{0\}|}\\ &= \frac{|\{0\}|}{|\{0\}|}\\ &= 1\end{aligned}
\qquad
\begin{aligned}\sigma_3(x,y) &= \frac{|[\![x]\!](3) \cap [\![y]\!](3)|}{|[\![x]\!](3) \cup [\![y]\!](3)|}\\ &= \frac{|\{3\} \cap \{2,3\}|}{|\{3\} \cup \{2,3\}|}\\ &= \frac{|\{3\}|}{|\{2,3\}|}\\ &= \frac{1}{2}\end{aligned}$$

Then to compute their overall similarity over the whole model:

$$\begin{aligned}\sigma(x,y) &= \frac{\sum_{p \in \{0,1,2,3\}} \sigma_p(x,y)}{|\{0,1,2,3\}|}\\ &= \frac{\frac{1}{2}+0+1+\frac{1}{2}}{4}\\ &= \frac{2}{4}\\ &= \frac{1}{2}\end{aligned}$$

5.4 Theorem Discovery Algorithm

Given the ability to represent and compute effects as described so far in this chapter, it is then straightforward to construct an algorithm that computes equivalence theorems. In essence, the algorithm systematically computes ever-longer action strings and their effects, grouping the strings into equivalence classes by effect. When a string is added to an existing class (rather than forming a new one), the algorithm reports a total equivalence theorem between that string and the canonical/first member of the class it has been added to.

The algorithm is essentially similar to that given in [GTC06], but generalised beyond the matrix representation of effects, and extended to include partial equivalence checking. In particular, the algorithm is parameterised by a similarity threshold t; if $t = 1$, the algorithm only looks for/reports total equivalence, but for $0 \leq t < 1$, each computed string is compared against every equivalence class' canonical member in search of partial equivalence theorems.

We have split the algorithm into two parts for ease of presentation and discussion:

- Algorithm 2 shows the top level structure. It repeatedly iterates over a set of *bases* (section 5.4.2), extending each of them with every atomic action a in the alphabet Σ in turn to produce a new string w. If that string is not *suffix-pruned* (section 5.4.3), it is checked against the equivalence classes, as described in algorithm 3. The n^{th} iteration of the algorithm checks strings of length n. The algorithm stops at the end of an iteration where no new bases have been discovered, which is the same as saying that no new equivalence classes were discovered on that iteration.

- Algorithm 3 focuses on the operation of checking a single string w against the set of equivalence classes. It iterates once over the set of *canonical members* (section 5.4.1) of the equivalence classes, looking for any whose effect is the same as the effect of w; if one is found, that is a total equivalence theorem; otherwise, and if the threshold t is less than 1, it checks the two effects for similarity, possibly reporting a partial equivalence theorem. When w has been checked against all of the canonicals, if no total equivalences have been found (i.e. the flag *found* is **false**), a new equivalence class containing just w is created, and w is added to the set of *bases* for the next iteration.

Algorithm 2 Algorithm to discover total and partial equivalence theorems

```
1: $E \leftarrow$ initialise()
2: addNewClass($E, \lambda$)
3: $bases \leftarrow \{ \lambda \}$
4: while $bases \neq \{\}$ do
5:    $bases' \leftarrow \{\}$
6:    for all $b \in bases$ do
7:       for all $a \in \Sigma$ do
8:          $w \leftarrow ba$          // concatenate atomic action onto base
9:          if $w$ is not suffix-pruned then
10:            check $w$ against $E$ — see algorithm 3
11:         end if
12:      end for
13:   end for
14:   $bases \leftarrow bases'$
15: end while
```

Algorithm 3 Inner part of algorithm: check a string w against existing equivalence classes

```
1: $found \leftarrow$ false
2: for all $c \in$ canons($E$) do
3:    if not $found$ and $[\![w]\!] = [\![c]\!]$ then
4:       Report total theorem: $w \sim tc$
5:       addToClass($E, w, c$)
6:       $found \leftarrow$ true
7:    else if $t < 1$ and $\sigma(w, c) \geq t$ then
8:       Report partial theorem: $w \approx_t c$
9:    end if
10: end for
11: if not $found$ then
12:    addNewClass($E, w$)
13:    $bases' \leftarrow bases' \cup \{ w \}$
14: end if
```

Three aspects of the algorithm warrant further explanation: the *equivalence classes* (and their associated operations), the sets *bases* and *bases'*, and *suffix-pruning*.

5.4.1 Equivalence classes

The collection of equivalence classes, written E, is the central data structure of this algorithm. Each equivalence class contains strings whose effects are identical, i.e. every time a total equivalence theorem is found, a string is added to some already existing equivalence class. Conceptually, E is just a set of sets of strings, though in section 6.2 we discuss optimisations. Critically, each equivalence class has a *canonical* member, against whose effect each new string's effect is compared: obviously it would be pointless to compare against every member of the class; the choice is free, but as the algorithm checks ever-longer strings, an obvious strategy is to pick the first string entered into the equivalence class, as it will be shortest. The algorithm uses the following operations involving the collection of equivalence classes:

- initialise() — create a new empty collection of equivalence classes.
- addNewClass(E, w) — add a new equivalence class to E, with one member, w.
- addToClass(E, w, c) — add string w to the existing class in E that has c as its canonical member.
- canons(E) — get the set of canonical members of the equivalence classes in E (so if E contains n equivalence classes, this will return a set of n strings).

5.4.2 The *bases* sets

The set *bases* is the set of strings to be extended by the letters in Σ on every iteration through the algorithm. A naïve version of the algorithm would simply check every string in Σ^*, but this is highly redundant (and indeed produces many redundant theorems). For example, suppose we find that $xyx \sim yy$ (in which case we add xyx to the equivalence class containing yy). Now, if we subsequently check the string $xyxy$, we would find xyx-prefixed theorems corresponding to every yy-prefixed theorem we discover. So, if we had also found that $xxxx \sim yy$ (say), then we would *also* find that $xxxx \sim xyx$, which is (we argue) totally redundant because we already know that $xyx \sim yy$, and total equivalence is (obviously) transitive.

The sets *bases* and *bases'* are used to avoid these redundant theorems (and computations) arising from the transitivity of total equivalence. In particular, on each iteration we build a set *bases'*, that at the end of the iteration contains exactly those strings that were checked in that iteration and found not to be members of any existing equivalence class (alternatively, we can say that it contains exactly those strings for which new equivalence classes were produced). This is then used as the set *bases* on the next iteration, each member being extended in turn with every letter in Σ to produce a new set of strings to check. On the n^{th} iteration of the algorithm, *bases* contains strings of length $n-1$, and *bases'* contains strings of length n. When an iteration of the algorithm produces no new equivalence classes, i.e. an empty *bases'* (which becomes an empty *bases* for the next iteration), the algorithm has found every total equivalence theorem it is going to find, and can stop.

5.4.3 Suffix-pruning

Beyond the redundancy-reduction encapsulated in the *bases* mechanism just described, the algorithm also skips checking a newly-computed string if any of its suffices have already been added to an existing equivalence class. For example, if we consider once again the model in figure 5.1, one of the total equivalence theorems that the algorithm finds for that model is:

$$yyy \sim \lambda$$

So, *yyy* is added to the equivalence class whose canonical member is λ. Now consider the string *xyyy*. First, we remark that its prefix *xyy* is not found to be totally equivalent to any other string, and so is added to the set of *bases* to be extended on the 4th iteration of the algorithm; thus, the string *xyyy* is *not* pruned by the *bases* mechanism described above. However, it is still worth pruning, for without pruning, the algorithm will produce the theorem:

$$xyyy \sim x$$

which is redundant because we already know:

$$yyy \sim \lambda$$

Without such pruning, the algorithm produces many such suffix-similar theorems which can be considered simply as redundant.

The exact mechanism of suffix-pruning is:

1. Compute all suffices of the input string w.
2. Compute the union of all of the equivalence classes, *minus* their canonical members.
3. If the intersection of the set of suffices and the set of non-canonical members of equivalence classes is non-empty, prune the string.

Note that we exclude canonical members from the set of strings to check the suffices against; without doing so, we would automatically prune *all* strings of length > 1 and never find *any* theorems.

5.4.4 Approximate worst-case peformance analysis

On the n^{th} iteration of the algorithm:

- There will be $W \leq |\Sigma|^n$ strings (of length n) to check (algorithm 2, lines 6–8).
- There will be at most $|\Sigma|^0 + |\Sigma|^1 + |\Sigma|^2 + \cdots |\Sigma|^n$ strings in E; in general this will be dominated by the term $|\Sigma|^n$, i.e. $|E| \in O(|\Sigma|^n)$
- Those strings will be in $C \leq |E|$ equivalence classes, with that many canonicals.
- Each of the W strings being checked will have $Z = n - 1$ suffices to examine during suffix pruning; each of those suffices will need to be checked against the $< |E|$ non-canonical strings.
- If it is not pruned, checking a given string for total and partial theorems will require C effect comparisons.
- Comparing two effects (naïvely) will require $X \leq |S|^2$ operations.

Putting all of this together, the n^{th} iteration of the algorithm will involve at most:

$$\begin{aligned} & W \times Z \times |E| \times C \times X \\ \leq\ & |\Sigma|^n \times (n-1) \times |\Sigma|^n \times |\Sigma|^n \times |S|^2 \\ \leq\ & |\Sigma|^{3n} \times |S|^2 \times (n-1) \end{aligned}$$

operations, i.e. the algorithm is cubic in the size of the alphabet.

This is a worst-case analysis; in practice:

- Many strings will be pruned.
- C will be considerably smaller than $|E|$, $|E|$ will be considerably smaller than the subset of Σ^* up to length n, etc.
- Actual implementations will be able to apply their own optimisations, e.g. to improve the speed of the frequently used effect computation and comparison operations; in the next section we describe one such optimisation.

In practice, we have found that there are diminishing returns (in terms of insight) as n grows, due to two factors:

1. As n grows, the strings in the theorems get longer, and more effort is required to understand the implications of the theorem; put another way: shorter theorems are easier to interpret.
2. As n grows, the number of theorems discovered can in some cases grow rapidly, making it harder to pick out interesting and salient ones from what might be considered 'noise'.

Neither of these factors is definitive, and in the next chapter we will see counterexamples; even so, we have (so far, at least) not found any interesting theorems outside the range $n \leq 8$, say, and we believe that in practice theorem discovery will prove most useful within this space: behaviour expressed in a theorem involving strings of 20 actions, say, will never be noticed by actual users [ADK03].

5.5 Meta-theorems: patterns and families

Theorem discovery as described so far in this chapter is a technique for finding total and partial equivalences between the effects of individual pairs of action strings. The question of how to interpret those theorems in order to obtain insight into a system's interaction behaviour is of course a general one, and the details will vary between systems, and will depend on the semantics of the actions concerned. However, it is also possible to look at theorems and sets of theorems *structurally* — that is, it is sometimes possible to say something about what a theorem means simply by looking at its syntactic structure, without reference to the meanings of the actions involved.

In this section we begin to explore this notion by giving a few simple examples, and we also consider the more general question of dealing with *families* of theorems, where in order to make an interpretation it is necessary to consider more than one theorem together — again, often structurally. These approaches extend theorem discovery beyond the setting described so far in this chapter, into a realm of 'meta-theorems'.

Our implementation (described in section 6.2) does not deal with any of the concepts discussed in this section — rather, it simply produces lists of individual theorems involving pairs of strings. The case studies in chapter 6 do include some examples of the concepts from this section, but their identification and analysis there is done entirely by hand, by visual inspection of the lists of basic theorems produced by the tool. While it is quite straightforward to see how some of the patterns in this section could be automatically identified by the tool, the question of how to automatically interpret families of theorems in general remains future work.

5.5.1 Idempotent action

A very simple theorem pattern is the **idempotent action**, where we have a theorem of the form

$$\exists\, a \in \Sigma \bullet aa \sim a$$

for some $a \in \Sigma$. Here we have an action that sets some aspect(s) of the system's state to some particular value, so that subsequent applications of that action have no further effect.

For example, in section 6.3.1 we have the theorems

$$\boxed{\text{off}}\boxed{\text{off}} \sim \boxed{\text{off}}$$
$$\boxed{\text{on}}\boxed{\text{on}} \sim \boxed{\text{on}}$$

indicating that once the system in question has been switched off (or on), it remains off (or on).

5.5.2 Action groups

The 'idempotent action' pattern can be generalised into families of theorems, to identify **groups of related actions which set some variable**. Here we have a set of actions, each of which affects the same aspect(s) of system state, so that later actions override earlier ones. Such families may be recognised via the following formalism, in which the set A is the group of actions:

$$\exists A \subseteq \Sigma \bullet \forall (a,b) \in A \times A \bullet ab \sim b$$

For example, in section 6.3.1 we have, with $A = \{\boxed{\text{off}}, \boxed{\text{on}}\}$:

$$\boxed{\text{on}}\boxed{\text{off}} \sim \boxed{\text{off}}$$
$$\boxed{\text{off}}\boxed{\text{on}} \sim \boxed{\text{on}}$$

indicating (along with the two theorems above) that the actions $\boxed{\text{off}}$ and $\boxed{\text{on}}$ both affect the same aspect of system state — in this case simply a flag indicating if the system is on or off.

5.5.3 Inverse actions

Another very simple theorem pattern, but a profound one where it occurs, is that of **inverse actions**, i.e. actions that undo each other:

$$\exists a,b \in \Sigma \bullet ab \sim ba \sim \lambda$$

Now, where such theorems exist, it is not guaranteed that theorem discovery will find them: the only strings that the theorem discovery algorithm is guaranteed to check are the atomic actions in Σ; ordinarily we'd expect those to all have different effects, in which case all length-2 strings will also be checked, but this need not be the case. As such, and given the obvious importance to users of inverse actions, it might be worth adding explicit checks against this pattern when implementing theorem discovery, and in particular looking for partial equivalences of this form with high similarity, as those will represent cases where two actions are nearly but not quite inverses, representing a potentially important area of surprise for the user.

Similarly, there are some cases where a given action *string* is its own inverse — as with *xyxy* in section 5.2.3. As seen in section 5.3.9.2, these can also be missed due to pruning in the algorithm, so again, it might be worth adding this pattern as a special case to check for, perhaps optionally.

5.5.4 Undo

Generalising the idea of inverse actions further, we might find that a given action is, in fact, an **undo**, always undoing the previous action (but not, in all cases, *vice versa*):

$$\exists\, u \in \Sigma \bullet \forall\, b \in \Sigma \backslash \{u\} \bullet bu \sim \lambda$$

Note the exclusion of u from Σ on the right hand side of this definition. Undo is a subtle concept, the subject of much research in the 1980s but these days generally rather taken for granted — and still often implemented inconsistently. In this definition we aim for a universal undo (i.e. one that undoes all actions in the system), but follow Dix's observation [Dix91; Dix95; Dix13], formally grounded in PIE theory, that a completely universal undo, that operates even on itself, is not in fact possible.

To briefly recap the argument: Suppose a system is in some state s_0 in which two actions are possible: a leading to s_a and b leading to s_b. Now, in either of those states, undo must clearly lead back to s_0, but what then happens if there is another undo immediately, i.e. to undo the first undo? The system must return to either s_a or s_b — but which? The only way in which this can be properly handled is if the system's state includes some extra history information enabling the undo (perhaps there are actually two copies of the state, allowing flip-flopping between two states, or perhaps there is a stack of such information), but if that is the case then undo after s_a and after s_b actually lead to two different states, and not in fact s_0 (since in s_0 the undo information component of the state would be different).

Thus we must acknowledge that a fully universal undo is not possible: there will always be some actions in the system (at the very least undo itself) on which undo cannot act in this manner. Of course, there may be other actions that we don't expect undo to work on (cursor movements, say, which have inverses but are perhaps best excluded from undo), in which case they would also need to be removed from Σ before looking for undo in this way.

Undo is clearly an important kind of metatheorem to know about, and so once again we suggest that it might be worth making it a special case and explicitly checking for it (as it only involves looking for equivalences on length-2 strings). (Special-casing is necessary because of the exclusion of u from Σ noted above: normal theorem discovery does not produce theorems that follow this pattern.)

Finally we note that it is also possible to conceive, similarly, of multi-step undo *strings*, though it is unclear at this time the degree to which this makes sense or is useful:

$$\exists\, w \in \Sigma^* \bullet \forall\, x \in \Sigma^* \backslash \{w\} \bullet xw \sim \lambda$$

5.5.5 Safe action

Some action or action string might always take us to the same state, in which case the user might consider it 'safe' in that it gives them reliable behaviour without their having to pay attention. An $\boxed{\text{off}}$ button is an obvious example — or pressing the $\boxed{\text{AC}}$ key on a desktop calculator repeatedly.

$$\forall\, w \in \Sigma^* \bullet \text{safe}(w) \iff \exists\, q \in S \bullet \forall\, p \in S \bullet [\![w]\!](p) = \{q\}$$

Unlike the other meta-theorem concepts discussed in this section, this cannot be learnt only by looking at the simple theorems produced by theorem discovery: checking for safe action means looking explicitly at the effect of an action string in each state, i.e. at the structure of the effect itself. As such, we propose it as a potentially useful extension to the normal theorem discovery process, or a possible task to perform using the theorem discovery infrastructure on which the algorithm is built.

We also note that this is another case where partiality is particularly interesting: if a user believes that some action sequence always takes them to a safe state, but actually in a few states that's not true, that represents a potential surprise and possibly even a hazard.

5.5.6 Families of theorems

There will frequently be cases where some aspect of a system's interactive behaviour is expressed as a set of theorems rather than just one. Action groups and undo, as described above, are two examples of this phenomenon; in order to recognise their presence automatically, some post-processing is required beyond the basic theorem discovery algorithm described in section 5.4. In the cases of action groups and undo, we have identified particular patterns of theorems to look for; in general, however, families can arise with unpredictable structure, as emergent properties of the particular system being analysed.

We see an example in section 6.5.2, with theorems on the 'device' model of the Casio HS-8V simulation considered in section 4.5. There are two families of nine theorems each, where each family corresponds to some (different) aspect of the system's behaviour. To produce that analysis it was necessary to visually examine the eighteen theorems that theorem discovery found on that model, and look for patterns — by applying human intelligence it was then quite straightforward then to group the theorems together and interpret them as described later. It is unclear at this time to what degree it ought to be possible to automate or support this process; the question of how to do this in general is beyond the scope of this thesis, and is proposed as future work; see also section 6.2.3.

Chapter 6

Theorem Discovery Implementation and Case Studies

Contents

6.1 Introduction

In this chapter we consider theorem discovery in action. We describe our theorem discovery implementation in the form of the `fsmActions` library and `fsmKit` tool, and follow this with three case studies where we apply that tool to models of actual interactive systems. The case studies in question correspond to three of the case studies in chapter 4, i.e. we describe the application of theorem discovery to the models produced in those examples. We survey the theorems discovered, discuss the implications or reasons behind particular ones, and pay particular attention to theorems that are surprising or that, we argue, represent a potential to surprise the user.

6.2 Implementation

In this section we describe our implementation of theorem discovery, which has been used to produce the theorems discussed in the case studies in the rest of this chapter. A detailed examination of the implementation is beyond the scope of this thesis; rather we provide an overview of its general structure and components, and look more closely at certain specific aspects, and consider how it might be improved and extended.

This software has been implemented in the programming language Haskell. Our reasons for using Haskell are much as described in section 4.2.2; to recap and summarise, using Haskell gives us (mainly thanks to its strong and thoroughgoing type system, and brevity of code):

- Deeper understanding of the exact requirements and details of the algorithm and required data structures, including in corner cases.
- High confidence in the correctness and robustness of the implementation.

The source code[1] is divided between two major components:

1. The `fsmActions` package (section 6.2.1) — a library of basic facilities for representing and manipulating finite state machines and effects as formalised in section 5.3.
2. The `fsmKit` tool (section 6.2.2) — a command-line tool built on `fsmActions`, implementing the theorem discovery algorithm described in section 5.4.

6.2.1 The `fsmActions` package

`fsmActions` is a Haskell package (publically available on the *hackage* database[2]) for representing and manipulating finite state machines, action strings, and effects as formalised in section 5.3 of this thesis. Its key contents are described in the following sections.

[1] `http://www.cs.swan.ac.uk/~csandy/phd/`
[2] `http://hackage.haskell.org/package/fsmActions`

6.2.1.1 The `Data.FsmActions` module

This is the top-level module of the package, in which its primary data structures and operations are defined. The key data types are:

- *State* — simply a synonym for *Int*. In `fsmActions` an FSM's states are assumed to form a contiguous range starting at 0.
- *DestinationSet* — a list of *States*. This is the implementation of *destination set* as defined in section 5.3.5.
- *Action* — a list of *DestinationSets*. This is the implementation of *effect*[3] as defined in section 5.3.8, specifically the 'common case' described there where the set of states is simply numbered $0 \ldots n$.
- *FSM sy* — a finite state machine, parameterised over alphabet/symbols type *sy*. In practice we have exclusively used FSMs over textual types, i.e. *FSM Char* and *FSM String*, but the library is generic in this regard. It is implemented as a *Map*[4] from *sy* to *Action*, allowing fast lookup by symbol.
- *Word sy* — a string of symbols, simply implemented as a list of *sy*.

There are various operations for constructing, deconstructing, querying and combining values of these data types; in particular:

- $append :: Action \rightarrow Action \rightarrow Action$ — to compose two effects as described in sections 5.2.3 and 5.3.5.
- $action :: FSM\ sy \rightarrow Word\ sy \rightarrow Maybe\ Action$ — to compute the effect (the *Maybe Action*) of some string (the *Word sy*) in some FSM; the return value is encased in a *Maybe* to catch the case where the word contains symbols outside the FSM's alphabet.
- $actionEquiv :: Action \rightarrow Action \rightarrow Bool$ — to test if two effects are totally equivalent.

[3]The choice of name is somewhat confusing: it would (now) be better if this data type was called *Effect*, but at the time of implementation *Action* was chosen and as the software is now publically available it has not been changed.

[4]From the standard `Data.Map` module, `http://hackage.haskell.org/package/containers`.

```
{{0->1,x},
 {0->3,y},
 {1->2,x},
 {1->1,y},
 {2->0,x},
 {2->0,y},
 {3->2,x},
 {3->3,y}}
```

Figure 6.1: An example of the `fsmEdges` format: the FSM shown in figure 5.1.

6.2.1.2 I/O and serialialised representations

The package is capable of reading and writing FSMs serialised in a variety of formats. The modules involved are:

- `Data.FsmActions.IO` — 'top level' module for performing I/O in any of the three formats described below. Provides functions *loadFsm* which takes a path to a file on disk and a list of formats to try, and returns the corresponding *FSM*, and *saveFsm* which saves an *FSM* in a specified format to a file on disk.
- `Data.FsmActions.FsmEdges` — to load and save FSMs represented as lists of labelled edges. For example, see figure 6.1.
- `Data.FsmActions.ActionMatrix` — to load and save FSMs represented as collections of binary adjacency matrices (as exemplified briefly in section 5.2.1).
- `Data.FsmActions.FsmMatrix` — to load and save FSMs represented in another matrix representation, where the whole FSM is represented in a single matrix, and each row contains space-separated destination sets (as lists of comma-separated destination states) for one state.
- `Data.FsmActions.Graph` — to render and save FSMs in the DOT format of the GraphViz tool[5], for offline rendering to graphical formats.

[5]`http://graphviz.org`

```
loadFsm a "examples/aircon/aircon.fsmEdges"
isDFSM a
stateCount a
edgeCount a
actionCount a
actionNames a
partials a 1 3 95/100
dotGraph a "aircon.dot"
```

Figure 6.2: Example `fsmKit` input, for the air conditioning control system (section 6.3).

```
Loaded a from examples/aircon/aircon.fsmEdges
a is deterministic
a has 312 states
a has 2808 edges
a has 9 actions
a actions are:["cool","down","heat","high","low","med","off","on","up"]
Theorems (0):
Theorems (49):
  ["cool","cool"] = ["cool"]
  ["cool","down"] = ["down","cool"]
  ["cool","heat"] = ["heat"]
  ["cool","high"] = ["high","cool"]
...
  96.15% (25/26) : ["down","up"] ~= []
  96.15% (25/26) : ["up","down"] ~= []
```

Figure 6.3: Part of the output produced by feeding the contents of figure 6.2 to `fsmKit`.

6.2.2 The `fsmKit` tool

6.2.2.1 Overview / example

The actual theorem discovery algorithm is implemented in `fsmKit`, a command-line tool built upon the `fsmActions` package just described. The tool reads (from standard input) a list of commands in a small language it defines, and then executes those commands. An example input file is shown in figure 6.2: the input file `aircon.fsmEdges` is read and its content stored in a variable `a`; a few simple measures are taken of `a` (is it deterministic, how many states does it have, etc.), and then the theorem discovery algorithm is executed, looking for theorems on strings of length 1 to 3, and including partial theorems with a threshold of 95% (to only look for total theorems, the command would read "`theorems a 1 3`"). Finally, a DOT graph of the model is written to the file `aircon.dot`.

Part of the output produced by running `fsmKit` against that input is shown in figure 6.3. The algorithm found no theorems on strings of length 1, but 49 theorems on strings of length 2, of which we show the first four and the last two (which happen to be partial); for example:

- The first theorem indicates that multiple instances of the `cool` action have the same effect as a single instance (i.e. it is idempotent — see section 5.5.1).
- The last two theorems shown indicate that the sequences "`down` then `up`", and "`up` then `down`" are 96.15% similar to the identity effect (i.e. 'do nothing'). This example is discussed further in section 6.3.2.

6.2.2.2 The module `Data.FsmActions.Theorems`

The core of `fsmKit` is the module `Data.FsmActions.Theorems`, which implements the theorem discovery algorithm. The implementation is a fairly straightforward transliteration of the algorithm described in section 5.4 into Haskell, with some aspects of note.

While the algorithm is described in this thesis in imperative style, the implementation is in more idiomatic (i.e. functional) Haskell: maps and folds over collections rather than while and for loops, etc.

There are five key data structures involved. They are all parameterised over *sy*, the type of the FSM's alphabet of symbols (see description of the *FSM* type in section 6.2.1.1). They are:

- *EventString sy* — strings of 'events', i.e. user actions. This is actually just implemented using Haskell's built-in list type.
- *EventAction sy* — an $(\mathit{EventString}\ sy, \mathit{Action})$ pair, containing an event string and its corresponding effect (the *Action* value).
- *Theorem sy* — theorems involving strings of type *sy*; an algebraic data type with two cases: *Equivalence* (for total equivalence, carrying the two equivalent *EventString sy* values) and *Partial* (for partial equivalence, which additionally carries the *EventStrings*' similarity as a *Rational* value).
- *Equivs sy* — the collection of equivalence classes, implemented as a map from *Action* to *EventString*. That is, it is a map from effects to lists of strings having that effect; the first member

of each list is taken as the canonical member of the class. The choice to implement *Equiv* as a map has performance implications which are discussed below.

- *Workings sy* — a data structure containing the algorithm's 'workings', i.e. its current state (as embodied in the algorithm in section 5.4 as the variables *bases*, *bases′*, *E*, etc). A value of this type is passed among the various functions involved in the implementation; it is an implementation detail whose exact structure is not important to the current discussion.

The algorithm is run via one of the following functions:

- $fsmTheorems :: Rational \rightarrow FSM\ sy \rightarrow [[Theorem\ sy]]$ — takes a similarity threshold and a finite state machine, and returns all of the theorems discovered by running the algorithm until it can find no more theorems. The returned value is a list of lists of theorems: they are partitioned according to the length of the string that was being checked when the theorem was discovered, (equivalently, according to the iteration of the algorithm).

- $someTheorems :: Int \rightarrow Int \rightarrow Rational \rightarrow FSM\ sy \rightarrow [[Theorem\ sy]]$ — like *fsmTheorems* but also takes two integers x and y, and only returns theorems on strings of length between x and y (though to build the equivalence classes it will still also need to check strings of length $< x$).

- $fsmTheoremsFull :: Rational \rightarrow FSM\ sy \rightarrow [Workings\ sy]$ — like *fsmTheorems* but returns the full set of *Workings* (for debugging and introspection purposes).

The implementation prunes redundant strings according to the two techniques described in section 5.4, i.e. uses of *bases*, and suffix-pruning. Beyond this (and following Knuth's well-known remarks on premature optimisation [Knu74]) it includes only one aspect aimed at improving its performance, namely the use of the *Data.Map* type in the implementation of *Equiv*. In order to check for total equivalence, the algorithm takes the effect of the string currently being checked, and determines whether it is identical to the effect of any of the canonical members of the equivalence classes already computed; as *Equiv* is implemented as a map from *Action* (i.e. effect) to a list of strings having that effect, checking if an effect is already known is simply a matter of performing a *lookup* in the map, and the *Map* type is implemented such that this is an $O(\log n)$ operation (where n is the number of keys in the map, i.e. the number of equivalence classes). Checking explicitly against every class' canonical member's effect is an $O(n)$

operation, so this represents a saving. The saving is modest, but as we get it 'for free' and *Map* is a very natural data structure to use for this purpose anyway, it is worth having.

Note also that this saving only applies for total theorem checking: if the threshold is < 1, the algorithm will look for partial equivalence theorems, which *does* necessitate explicitly checking against the effect of every equivalence class (after all, a string can be partially equivalent to multiple other strings). As such, the implementation explicitly distinguishes between the cases $t = 1$ and $t < 1$. In the first case it uses the *lookup* optimisation described above; otherwise it iterates over all of the equivalence classes (even while looking for total equivalences).

There are, of course, numerous possible optimisations imaginable. For example, using locality-sensitive hashing [GIM99] in the structure of the equivalence classes could improve the performance not only of total equivalence checking, but even *partial* equivalence checking. The algorithm is also a good candidate for concurrent processing: in models with many states, it could be worth dividing the computation involved in composing and comparing effects across many processors; and (particularly for partial equivalence checking), the task of comparing an effect with the effects corresponding to each equivalence class is ripe for parallelisation. Such optimisations are beyond the scope of this thesis, however.

6.2.3 The `fsmKit` user experience

We conclude this section with some thoughts on the experience of using the `fsmKit` tool in order to do theorem discovery. The tool is, clearly, a prototype, and the primary drive behind its creation has been exploration and comprehension of what is actually involved in implementing the theorem discovery algorithm *at all.* Its interface is purely textual and batch-oriented: the user feeds it a 'program' of commands in its input language, and gets (say) a few screenfuls of text in response, in the style seen in figure 6.3. The user must then (typically) closely examine that output in order to interpret the theorems presented, and to work out what is interesting and what is 'noise', which theorems are related to which others, etc.

(Should the user wish to dig deeper, they can construct their own Haskell programs against the `fsmActions` package and the `Data.FsmActions.Theorems` module, bypassing the `fsmKit` and its input language altogether. `Data.FsmActions.Theorems` makes this easier by providing several *pretty printer* routines for nicely formatting *Theorem* and *Workings* values, for example, but obviously such an undertaking still requires good facility with Haskell.)

Overall, using `fsmKit` is quite a 'low level' process requiring a lot of thought and close inspection of fairly ugly plain text output. How might this be improved?

- An obvious improvement would be to simply format the output more sensitively. A GUI of some sort could cut down on visual 'noise' such as the abundance of square brackets, distinguish actions by colour, etc. — and just generally be "easier on the eye".
- Similarly, filtering theorems on particular symbols or substrings is an obvious and fairly simple thing to do.
- To cut down on noise, it could be useful to have an 'ignore' list, containing theorems or parts of theorems that have been considered but discarded as uninteresting, and the tool could simply ignore them.
- Identification of meta-theorems and possible families of theorems, as described in section 5.5 could be of great value. There are still unaddressed computational questions there, but once they are answered, grouping related theorems together is a very natural enhancement to theorem discovery.
- The current implementation is oriented entirely towards *actions* and has essentially no concept of system *state* — that is, FSMs have states, but those states are only labelled with numbers, and there is no way to relate them back to the state of the interactive system being investigated (as produced by model discovery, say). For total equivalence theorems, this is perfectly fine: all that matters is whether the equivalence holds *across all states*.

  For partial equivalence theorems, on the other hand, it is very natural to want to know in *which* states the two actions diverge — and where those divergenes lead. Now, it would be simple to extend the current tool to report that *in terms of state numbers*, but there is currently no mechanism for 'closing the loop' back to a richer model that can tell you something about what's actually going on in the interactive system under investigation in those states — and none of the `fsmActions` serialisation formats described in section 6.2.1.2 contain rich state information. In order to make the most of partial equivalence theorems, this loop needs closing.

All of these enhancements are beyond the scope of this thesis. Our contribution has been to provide a cleanly-defined and rigorously-implemented theorem discovery tool; it could conceivably be wrapped in a more usable interface with relatively little work and little to no modification of the actual tool itself.

We suggest that when designing tools for dependable HCI, a good approach is to start by constructing a well-defined and rigorously-implemented 'engine' (as we have done here), and *then* to work to improve its usability — preferably in a cleanly separated layer of software. Naturally, usability concerns will feed back into the design of the underlying 'engine', but by keeping good separation between these two concerns, confidence in the correctness and thus usefulness of the tool can be maintained.

6.3 Case study 1: Air Conditioning Control Panel

The air conditioning control system was introduced in section 3.1.2, and the application of model discovery to this system was described in section 4.2. As noted in those sections, this is a quite simple system, with a very orthogonal structure; as such, we might expect to see a high degree of regularity in the findings produced by theorem discovery. The model discovery performed on the system was a single application of the basic model discovery algorithm, without any conditional exploration or variation in terms of the aspects of system state that were projected; as such, in this case study we apply theorem discovery to just one model, which is of the entire state space of the system.

As shown in figures 6.2 and 6.3, the model has 312 states, 2808 edges, and 9 actions. Throughout the rest of this chapter, we adopt the convention of writing actions in a 'push button' style, so (for example) we write these 9 actions not as `"on"`, `"off"`, `"up"`, `"down"`, etc. but:

$$\boxed{\text{on}}\,,\ \boxed{\text{off}}\,,\ \boxed{\text{cool}}\,,\ \boxed{\text{heat}}\,,\ \boxed{\text{low}}\,,\ \boxed{\text{med}}\,,\ \boxed{\text{high}}\,,\ \boxed{\blacktriangle}\,,\ \boxed{\blacktriangledown}$$

and we write sequence of actions as (say):

$$\boxed{\text{on}}\boxed{\text{heat}}\boxed{\blacktriangle}\boxed{\blacktriangle}$$

6.3.1 Total equivalence theorems

The theorem discovery tool found no theorems of length 1, i.e. every atomic action has a different effect, and they all have *some* effect (i.e. are not the identity).

Conversely, there are 47 total equivalence theorems of length 2. Seven of these that immediately stand out (because their right-hand sides are atomic actions) are:

$$\begin{array}{c}
\boxed{\text{off}}\boxed{\text{off}} \sim \boxed{\text{off}} \\
\boxed{\text{on}}\boxed{\text{on}} \sim \boxed{\text{on}} \\
\boxed{\text{heat}}\boxed{\text{heat}} \sim \boxed{\text{heat}} \\
\boxed{\text{cool}}\boxed{\text{cool}} \sim \boxed{\text{cool}} \\
\boxed{\text{low}}\boxed{\text{low}} \sim \boxed{\text{low}} \\
\boxed{\text{med}}\boxed{\text{med}} \sim \boxed{\text{med}} \\
\boxed{\text{high}}\boxed{\text{high}} \sim \boxed{\text{high}}
\end{array}$$

That is, each of the 7 atomic actions listed on the right is *idempotent*: multiple applications of those actions have the same effect as just one. This reflects the fact that these actions all set some particular system state variable to some particular value, with no side-effects. Note the absence of $\boxed{\blacktriangle}$ and $\boxed{\blacktriangledown}$ here, because those actions *modify* the value of some part of the system's state, but do not set it to a particular value, so multiple applications do 'stack up' their effects.

On closer inspection, we also find these 10 theorems, involving those same 7 actions:

$$\begin{array}{cc}
\boxed{\text{on}}\boxed{\text{off}} \sim \boxed{\text{off}} & \boxed{\text{off}}\boxed{\text{on}} \sim \boxed{\text{on}} \\
\boxed{\text{cool}}\boxed{\text{heat}} \sim \boxed{\text{heat}} & \boxed{\text{heat}}\boxed{\text{cool}} \sim \boxed{\text{cool}} \\
\boxed{\text{high}}\boxed{\text{low}} \sim \boxed{\text{low}} & \boxed{\text{med}}\boxed{\text{low}} \sim \boxed{\text{low}} \\
\boxed{\text{high}}\boxed{\text{med}} \sim \boxed{\text{med}} & \boxed{\text{low}}\boxed{\text{med}} \sim \boxed{\text{med}} \\
\boxed{\text{low}}\boxed{\text{high}} \sim \boxed{\text{high}} & \boxed{\text{med}}\boxed{\text{high}} \sim \boxed{\text{high}}
\end{array}$$

These reflect the fact that those 7 actions may be divided into three groups (i.e. $\boxed{\text{off}}$ / $\boxed{\text{on}}$, $\boxed{\text{heat}}$ / $\boxed{\text{cool}}$ and $\boxed{\text{low}}$ / $\boxed{\text{med}}$ / $\boxed{\text{high}}$), according to the system state variable that they affect, i.e. according to the component on the (highly orthogonal) UI that they are associated with. Thus, an alternative and more

$\boxed{\text{cool}}\boxed{\text{off}} \sim \boxed{\text{off}}\boxed{\text{cool}}$	$\boxed{\text{off}}\boxed{\blacktriangle} \sim \boxed{\blacktriangle}\boxed{\text{off}}$
$\boxed{\text{cool}}\boxed{\text{on}} \sim \boxed{\text{on}}\boxed{\text{cool}}$	$\boxed{\blacktriangledown}\boxed{\text{off}} \sim \boxed{\text{off}}\boxed{\blacktriangledown}$
$\boxed{\text{cool}}\boxed{\text{low}} \sim \boxed{\text{low}}\boxed{\text{cool}}$	$\boxed{\text{on}}\boxed{\blacktriangle} \sim \boxed{\blacktriangle}\boxed{\text{on}}$
$\boxed{\text{cool}}\boxed{\text{med}} \sim \boxed{\text{med}}\boxed{\text{cool}}$	$\boxed{\blacktriangledown}\boxed{\text{on}} \sim \boxed{\text{on}}\boxed{\blacktriangledown}$
$\boxed{\text{cool}}\boxed{\text{high}} \sim \boxed{\text{high}}\boxed{\text{cool}}$	$\boxed{\text{cool}}\boxed{\blacktriangle} \sim \boxed{\blacktriangle}\boxed{\text{cool}}$
$\boxed{\text{heat}}\boxed{\text{off}} \sim \boxed{\text{off}}\boxed{\text{heat}}$	$\boxed{\text{cool}}\boxed{\blacktriangledown} \sim \boxed{\blacktriangledown}\boxed{\text{cool}}$
$\boxed{\text{heat}}\boxed{\text{on}} \sim \boxed{\text{on}}\boxed{\text{heat}}$	$\boxed{\blacktriangledown}\boxed{\text{heat}} \sim \boxed{\text{heat}}\boxed{\blacktriangledown}$
$\boxed{\text{heat}}\boxed{\text{low}} \sim \boxed{\text{low}}\boxed{\text{heat}}$	$\boxed{\text{heat}}\boxed{\blacktriangle} \sim \boxed{\blacktriangle}\boxed{\text{heat}}$
$\boxed{\text{heat}}\boxed{\text{med}} \sim \boxed{\text{med}}\boxed{\text{heat}}$	$\boxed{\blacktriangledown}\boxed{\text{low}} \sim \boxed{\text{low}}\boxed{\blacktriangledown}$
$\boxed{\text{heat}}\boxed{\text{high}} \sim \boxed{\text{high}}\boxed{\text{heat}}$	$\boxed{\text{low}}\boxed{\blacktriangle} \sim \boxed{\blacktriangle}\boxed{\text{low}}$
$\boxed{\text{low}}\boxed{\text{off}} \sim \boxed{\text{off}}\boxed{\text{low}}$	$\boxed{\blacktriangledown}\boxed{\text{med}} \sim \boxed{\text{med}}\boxed{\blacktriangledown}$
$\boxed{\text{low}}\boxed{\text{on}} \sim \boxed{\text{on}}\boxed{\text{low}}$	$\boxed{\text{med}}\boxed{\blacktriangle} \sim \boxed{\blacktriangle}\boxed{\text{med}}$
$\boxed{\text{med}}\boxed{\text{off}} \sim \boxed{\text{off}}\boxed{\text{med}}$	$\boxed{\blacktriangledown}\boxed{\text{high}} \sim \boxed{\text{high}}\boxed{\blacktriangledown}$
$\boxed{\text{med}}\boxed{\text{on}} \sim \boxed{\text{on}}\boxed{\text{med}}$	$\boxed{\text{high}}\boxed{\blacktriangle} \sim \boxed{\blacktriangle}\boxed{\text{high}}$
$\boxed{\text{high}}\boxed{\text{off}} \sim \boxed{\text{off}}\boxed{\text{high}}$	
$\boxed{\text{high}}\boxed{\text{on}} \sim \boxed{\text{on}}\boxed{\text{high}}$	

Figure 6.4: Thirty commutative theorems on strings of length 2 (air conditioning control panel).

general reading of both these 10 theorems and the 7 discussed above is: "within a given action group, the most recent action always over-rides any earlier actions in the same group" (see section 5.5.2).

There are 30 remaining total equivalence theorems of length 2, and they are all commutative, of form $\boxed{\text{X}}\boxed{\text{Y}} \sim \boxed{\text{Y}}\boxed{\text{X}}$. They are shown in figure 6.4 in two columns: the 16 on the left involve actions from the groups just described, and in each case $\boxed{\text{X}}$ and $\boxed{\text{Y}}$ are in different groups; the 14 on the right are similar, but in each of these, exactly one of $\boxed{\text{X}}$ or $\boxed{\text{Y}}$ is $\boxed{\blacktriangle}$ or $\boxed{\blacktriangledown}$: These theorems represent the orthogonality of the interface's components: between groups (and of course $\boxed{\blacktriangle}$ and $\boxed{\blacktriangledown}$ form their own group), the order of operations never matters, exactly because each group modifies an entirely independent aspect of the system's state.

There are 9 theorems of length 3; the first two, which we return to in section 6.3.2, are:

$$\boxed{\blacktriangledown}\boxed{\blacktriangle}\boxed{\blacktriangledown} \sim \boxed{\blacktriangledown}$$
$$\boxed{\blacktriangle}\boxed{\blacktriangledown}\boxed{\blacktriangle} \sim \boxed{\blacktriangle}$$

The other 7 theorems on strings of length 3 are:

$$
\begin{array}{rcl}
\boxed{\text{heat}}\boxed{\blacktriangledown}\boxed{\text{cool}} & \sim & \boxed{\blacktriangledown}\boxed{\text{cool}} \\
\boxed{\text{off}}\boxed{\blacktriangledown}\boxed{\blacktriangle} & \sim & \boxed{\blacktriangledown}\boxed{\blacktriangle}\boxed{\text{off}} \\
\boxed{\text{on}}\boxed{\blacktriangledown}\boxed{\blacktriangle} & \sim & \boxed{\blacktriangledown}\boxed{\blacktriangle}\boxed{\text{on}} \\
\boxed{\text{heat}}\boxed{\blacktriangledown}\boxed{\blacktriangle} & \sim & \boxed{\blacktriangledown}\boxed{\blacktriangle}\boxed{\text{heat}} \\
\boxed{\text{low}}\boxed{\blacktriangledown}\boxed{\blacktriangle} & \sim & \boxed{\blacktriangledown}\boxed{\blacktriangle}\boxed{\text{low}} \\
\boxed{\text{med}}\boxed{\blacktriangledown}\boxed{\blacktriangle} & \sim & \boxed{\blacktriangledown}\boxed{\blacktriangle}\boxed{\text{med}} \\
\boxed{\text{high}}\boxed{\blacktriangledown}\boxed{\blacktriangle} & \sim & \boxed{\blacktriangledown}\boxed{\blacktriangle}\boxed{\text{high}}
\end{array}
$$

There are two things say about these theorems. The first is that they are all quite comprehensible — if you understand the semantics of the device, there are no surprises here. The second is that given that comprehension, we might expect to see many more, similar, theorems here. For example, $\boxed{\text{cool}}\boxed{\blacktriangledown}\boxed{\text{heat}} \sim \boxed{\blacktriangledown}\boxed{\text{heat}}$ is a 'reflected' form of the first theorem which we would probably expect to be true; indeed, it *is* true — so why isn't it shown?

The answer is that the string $\boxed{\text{cool}}\boxed{\blacktriangledown}\boxed{\text{heat}}$ has been pruned. On its second iteration the algorithm found a total equivalence for the string $\boxed{\text{cool}}\boxed{\blacktriangledown}$ (see right-hand column above). As such, that string was added to an existing equivalence class, and thus *not* to the sets *bases'*. Thus, it was not in the set *bases* on the third iteration, so that iteration did not check (indeed, did not *compute*) any strings beginning with $\boxed{\text{cool}}\boxed{\blacktriangledown}$.

The algorithm prunes strings to reduce redundant computation, and redundant output. What this example shows is that the algorithm's design implicitly makes a decision about how to do this, and (inevitably) there are then theorems that it does not find. It happens that in this case we can deduce this 'missing' theorem from two theorems already found:

$$
\begin{array}{lclcl}
\boxed{\text{cool}}\boxed{\blacktriangledown} \sim \boxed{\blacktriangledown}\boxed{\text{cool}} & \Longrightarrow & \boxed{\text{cool}}\boxed{\blacktriangledown}\boxed{\text{heat}} \sim \boxed{\blacktriangledown}\boxed{\text{cool}}\boxed{\text{heat}} & & \text{(append)} \\
\boxed{\text{cool}}\boxed{\text{heat}} \sim \boxed{\text{heat}} & \Longrightarrow & \boxed{\text{cool}}\boxed{\blacktriangledown}\boxed{\text{heat}} \sim \boxed{\blacktriangledown}\boxed{\text{heat}} & & \text{(reduce)}
\end{array}
$$

However, in general this is not the case. Consider $\boxed{\text{off}}\boxed{\blacktriangle}\boxed{\blacktriangledown} \sim \boxed{\blacktriangle}\boxed{\blacktriangledown}\boxed{\text{off}}$. This is also a true theorem, but there is no way to deduce it from the theorems already discovered. Thus, in general, the deductive system over strings of length n that arises from the theorems discovered over strings of length

$< n$ is *incomplete.*

This raises an important, if somewhat subtle, point about the envisaged context of use of the theorem discovery algorithm. We see the algorithm as primarily useful (and certainly *novel*) as a technique for finding *unforeseen truths* about a system. That is, the theorem discovery algorithm is intended to surprise its users (programmers), that they may act to reduce the amount of surprise remaining in the system for *its* users (end users). The algorithm is *not* primarily intended as a tool for checking the truth of things we expect to be true, i.e. for testing; as such, it is not necessary that it finds every theorem that is true. If the programmer wishes to introduce a regression test that $\boxed{\text{off}}\boxed{\blacktriangle}\boxed{\blacktriangledown} \sim \boxed{\blacktriangle}\boxed{\blacktriangledown}\boxed{\text{off}}$, say, a bad way to go about it would be to run the theorem discovery algorithm and check if that theorem is found: as we have shown, it is *not* found (but it is true).

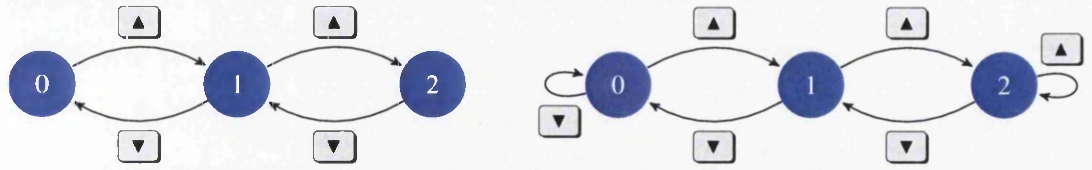
Having said that, the theorem discovery *infrastructure* can be used to check such a theorem's truth, however: the programmer could explicitly write a test that, given the FSM model of the system, computed the actions for the two strings and checked their equality — and indeed we suggest that this is a perfectly sensible thing to do, and a good response to the natural desire to test these kinds of theorems; this suggests that integration of theorem discovery into development environments should be fairly 'open', rather than just a 'black box' implementation of the algorithm.

So far in this case study, theorem discovery *hasn't* shown anything surprising. That is itself not surprising, given the simplicitly and orthogonality of the system being analysed. (Actually there *is* a surprising *absence* of a theorem: we haven't seen that $\boxed{\blacktriangledown}\boxed{\blacktriangle} \sim \boxed{\blacktriangle}\boxed{\blacktriangledown} \sim \lambda$, which we might expect; we return to this point in section 6.3.2 when we consider partial equivalence theorems.)

Going on and looking at theorems of length > 3, we find (as suggested in section 5.4.4) that the algorithm does continue producing more and more total equivalence theorems, but they are (we argue) not particularly interesting or insight-inspiring, and as they get longer they also become much harder to interpret anyway. For example, on its tenth iteration, the algorithm finds 53 theorems, of which one is:

$$\boxed{\blacktriangle}\boxed{\blacktriangle}\boxed{\blacktriangle}\boxed{\blacktriangle}\boxed{\blacktriangledown}\boxed{\blacktriangledown}\boxed{\blacktriangledown}\boxed{\blacktriangledown}\boxed{\blacktriangledown}\boxed{\blacktriangle} \sim \boxed{\blacktriangledown}\boxed{\blacktriangle}\boxed{\blacktriangle}\boxed{\blacktriangle}\boxed{\blacktriangle}\boxed{\blacktriangle}\boxed{\blacktriangledown}\boxed{\blacktriangledown}\boxed{\blacktriangledown}\boxed{\blacktriangledown}$$

It is hard to see what useful insight this theorem can provide, however.



(a) No self-loops on end points. (b) Self-loops on end points.

Figure 6.5: Two versions of a model of a 3-state slider.

6.3.2 Partial equivalence theorems

As noted in section 6.2.2.1, performing theorem discovery with a threshold of 95% yields two partial theorems on strings of length 2:

$$\boxed{\blacktriangledown}\boxed{\blacktriangle} \approx_{96.15} \lambda$$
$$\boxed{\blacktriangle}\boxed{\blacktriangledown} \approx_{96.15} \lambda$$

That is, both ▼▲ and ▲▼ are 96.15% similar in effect to the empty string: they *nearly* have no effect. Put another way, ▼ is nearly the inverse of ▲. As noted in the previous section, we might reasonably expect total equivalences here: why are ▲ and ▼ *not* inverses?

The reason is that the temperature control is a slider, so at the end points, 'nothing happens'. Here it is worth considering the details of such models, and a subtle distinction between model discovery and theorem discovery. Consider figure 6.5, which shows two possible discovered models for a slider with just three values. Figure 6.5(a) shows an 'intuitive' model of such a widget's behaviour: there are three states, with ▲ and ▼ actions moving between them, but when the slider is at its minimum value, no ▼ action is possible, and when it is at its maximum value, no ▲ action is possible. ▲ and ▼ appear, intuitively, to be inverses of each other. Unfortunately, theorem discovery as formalised in this thesis is undefined on this model: in section 5.3.1 finite state machines are defined to have *total* transition functions, and that is not the case here. Thus, the question "what is the effect of the action string ▼▼?" has no answer, because that string's destination set is undefined in states 0 and 1.

One way to deal with situations like this would be to redefine the semantics of theorem discovery so that effects are defined only for the subset of states in which the action string is well-defined (with other states leading to an error — or equivalently, null — value), and to redefine theorem discovery (both total and partial) to compare effects only on the states for which both effects are defined. While this is potentially interesting future work, it is beyond the scope of this thesis, where we explore the semantics laid out in section 5.4, and so the implications of such an approach remain unexplored at this time. Our implementation simply assumes totality, by filling in any 'missing' edges with self-loops — leading to a model as in figure 6.5(b).

To be clear: totality is a requirement of theorem discovery, not model discovery, and the latter can certainly (by design) produce models where different states have different sets of actions. It happens that model discovery as implemented on the air conditioning control panel *does not* discover self-loops on sliders' end points. This is mentioned in section 4.2, and in particular in section 4.2.2.3, in the description of how *getControlActions* is implemented:

> ...given a *GuiControl* value, return a list of available *GuiAction* values for that widget; the possibilities are hard-coded here, with the proviso that we do *not* include actions that we would expect to lead to a self-loop — attempting an *Up* action on a slider in its maximum position, for example.

So the model produced by model discovery happens to be more like 6.5(a) in this respect — but when it is loaded into `fsmKit` for theorem discovery, self-loops are added leading to a model more like 6.5(b).

Now, in such a model, it is clear that $\boxed{\blacktriangle}$ and $\boxed{\blacktriangledown}$ are *not* inverses. For consider the effect of the action string $\boxed{\blacktriangledown}\boxed{\blacktriangle}$ in 6.5(b):

$$[\![\boxed{\blacktriangledown}\boxed{\blacktriangle}]\!] = \langle 1,1,2\rangle \neq \langle 0,1,2\rangle = \mathbb{I} = [\![\lambda]\!]$$

Similarly:

$$[\![\boxed{\blacktriangle}\boxed{\blacktriangledown}]\!] = \langle 0,1,1\rangle \neq \langle 0,1,2\rangle = \mathbb{I} = [\![\lambda]\!]$$

Neither effect is quite the identity — though they are close, and each only differ in one place. And indeed, this is the case in the air conditioning control panel: the temperate slider has 26 possible values (5°–30°), so in $\frac{25}{26} \approx 96.15\%$ of the 312 states of the system, $\boxed{\blacktriangledown}\boxed{\blacktriangle}$ takes you back to that state (i.e. the string has the same effect as the empty string) — but in those 12 remaining states, this is not the case, precisely because in those states, the temperate slider is in its minimum position. A similar analysis holds for the string $\boxed{\blacktriangle}\boxed{\blacktriangledown}$.

To conclude this discussion, we return to the two total equivalence theorems noted on page 140:

$$\boxed{\blacktriangledown}\boxed{\blacktriangle}\boxed{\blacktriangledown} \sim \boxed{\blacktriangledown}$$
$$\boxed{\blacktriangle}\boxed{\blacktriangledown}\boxed{\blacktriangle} \sim \boxed{\blacktriangle}$$

These theorems are both true of the reduced model in figure 6.5(b):

$$\llbracket\boxed{\blacktriangledown}\boxed{\blacktriangle}\boxed{\blacktriangledown}\rrbracket = \langle 0,0,1\rangle = \llbracket\boxed{\blacktriangledown}\rrbracket$$
$$\llbracket\boxed{\blacktriangle}\boxed{\blacktriangledown}\boxed{\blacktriangle}\rrbracket = \langle 1,2,2\rangle = \llbracket\boxed{\blacktriangle}\rrbracket$$

and it is easy to see how this extends onto the full air conditioning control panel's model.

Going on to partial theorems on strings of length 3 with threshold 95%, we find numerous permutations of $\boxed{\blacktriangledown}\boxed{\blacktriangle}\boxed{\text{X}} \approx_{96.15} \boxed{\text{X}}$, and $\boxed{\text{X}}\boxed{\blacktriangledown}\boxed{\blacktriangle} \approx_{96.15} \boxed{\text{X}}$, and others arising from the similarity of $\boxed{\blacktriangledown}$ and $\boxed{\blacktriangle}$; this raises the question of how to prune partial equivalence theorems, which currently remains future work.

6.4 Case study 2: Independent Digit / '5-key' Number Entry

In this section we perform theorem discovery on some of the models, discussed in section 4.3, of a '5-key' indepdendent digit number entry system. This case study has two distinguishing features:

- The model discovery process made extensive use of *conditional exploration* in order to produce models of four different well-contained subspaces of the entire state space of the system, as the entirety would be too large (given our current approaches) for model discovery and theorem discovery. From the theorem discovery point of view, we are thus interested in commonalities and differences of theorems discovered between the various subspaces.
- There are actually three similar but distinct number entry 'routines' being modelled and analysed here, called *Simple Spinner*, *Simple Arithmetic* and *BBraun v686E*; see section 4.3.1 for an overview of their similarities and differences. From the theorem discovery point of view, we are

thus interested in commonalities and differences of theorems discovered between these three routines.

Theorem discovery produces few surprises in this case study; that's quite reasonable: the systems are fairly regular, and section 4.3 notes where they vary from this — our expectations arising from that are, on the whole, met here. In particular the Simple Spinner routine is *highly* regular and has (for example) no significant partial theorems and lots of symmetry; the Simple Arithmetic routine tends to be a bit less regular, with more partial theorems; and the BBraun routine is even less regular, with some odd corners. We do not propose to provide an exhaustive list of theorems in each example, but rather to pick up on particular aspects of interest. In the examples given we run theorem discovery looking for theorems on strings up to length 3 or in some cases 4, with a similarity threshold of 90% or in some cases 95%.

6.4.1 Tenths, hundredths only; digits 0–2 only

These models are as described in section 4.3.3.1: exploring just the tenths and hundredths digits of the display, and only the values 0–2 in those digits.

The Simple Spinner and Simple Arithmetic routines both have the same structure (figure 4.11, page 79), which is quite simple and regular. On this model, theorem discovery finds four theorems of length 2:

$$\boxed{\blacktriangleleft}\boxed{\blacktriangleleft} \sim \boxed{\blacktriangleleft}$$
$$\boxed{\blacktriangleleft}\boxed{\blacktriangleright} \sim \boxed{\blacktriangleright}$$
$$\boxed{\blacktriangleright}\boxed{\blacktriangleleft} \sim \boxed{\blacktriangleleft}$$
$$\boxed{\blacktriangleright}\boxed{\blacktriangleright} \sim \boxed{\blacktriangleright}$$

These theorems simply reflect the restriction of the model to the hundredths and thousandths digits: the space is two digits 'wide', so multiple $\boxed{\blacktriangleleft}$ and $\boxed{\blacktriangleright}$ actions have the same effect as single ones, because there's nowhere else to go. Theorem discovery actually finds these four theorems for *every* one of the models considered in this section, precisely because in each case the space being explored is only two digits 'wide'. As such we will not repeat these theorems in the rest of this section: their presence may be taken as read.

On strings of length 3, theorem discovery finds eleven theorems. Four are also found in the BBraun model (see below), and can be seen as variants of the four length-2 theorems just mentioned, with added $\boxed{\blacktriangle}$ and $\boxed{\blacktriangledown}$ actions:

$$\boxed{\blacktriangleleft}\boxed{\blacktriangledown}\boxed{\blacktriangleleft} \sim \boxed{\blacktriangleleft}\boxed{\blacktriangledown}$$
$$\boxed{\blacktriangleleft}\boxed{\blacktriangle}\boxed{\blacktriangleleft} \sim \boxed{\blacktriangleleft}\boxed{\blacktriangle}$$
$$\boxed{\blacktriangleright}\boxed{\blacktriangledown}\boxed{\blacktriangleright} \sim \boxed{\blacktriangleright}\boxed{\blacktriangledown}$$
$$\boxed{\blacktriangleright}\boxed{\blacktriangle}\boxed{\blacktriangleright} \sim \boxed{\blacktriangleright}\boxed{\blacktriangle}$$

The remaining seven theorems reflect the *vertical* restriction of the model to the digits 0–2:

$$\boxed{\blacktriangledown}\boxed{\blacktriangle}\boxed{\blacktriangledown} \sim \boxed{\blacktriangledown}$$
$$\boxed{\blacktriangle}\boxed{\blacktriangledown}\boxed{\blacktriangle} \sim \boxed{\blacktriangle}$$
$$\boxed{\blacktriangledown}\boxed{\blacktriangledown}\boxed{\blacktriangledown} \sim \boxed{\blacktriangledown}\boxed{\blacktriangledown}$$
$$\boxed{\blacktriangle}\boxed{\blacktriangle}\boxed{\blacktriangle} \sim \boxed{\blacktriangle}\boxed{\blacktriangle}$$
$$\boxed{\blacktriangledown}\boxed{\blacktriangle}\boxed{\blacktriangle} \sim \boxed{\blacktriangle}\boxed{\blacktriangle}$$
$$\boxed{\blacktriangle}\boxed{\blacktriangledown}\boxed{\blacktriangledown} \sim \boxed{\blacktriangledown}\boxed{\blacktriangledown}$$
$$\boxed{\blacktriangledown}\boxed{\blacktriangledown}\boxed{\blacktriangle} \sim \boxed{\blacktriangle}\boxed{\blacktriangle}\boxed{\blacktriangledown}$$

Note that because of this restriction, the Simple Spinner is unable to 'spin' vertically, so we do not have (as we might expect) that $\boxed{\blacktriangle}\boxed{\blacktriangledown} \sim \boxed{\blacktriangledown}\boxed{\blacktriangle} \sim \lambda$ — but see section 6.4.3.

The BBraun model (figure 4.12, page 80) is slightly less regular, as noted in section 4.3.3.1 and indeed throughout section 4.3; this is reflected in the results of theorem discovery. The model exhibits the first four of the length-3 theorems found for the Simple Spinner, mentioned above. It *does not* have the seven other length-3 theorems, but theorem discovery with a threshold of 80% finds *five* of those:

$$\boxed{\blacktriangledown}\boxed{\blacktriangle}\boxed{\blacktriangledown} \approx_{92.86} \boxed{\blacktriangledown}$$
$$\boxed{\blacktriangle}\boxed{\blacktriangledown}\boxed{\blacktriangle} \approx_{92.86} \boxed{\blacktriangle}$$
$$\boxed{\blacktriangledown}\boxed{\blacktriangledown}\boxed{\blacktriangledown} \approx_{85.71} \boxed{\blacktriangledown}\boxed{\blacktriangledown}$$
$$\boxed{\blacktriangle}\boxed{\blacktriangle}\boxed{\blacktriangle} \approx_{92.86} \boxed{\blacktriangle}\boxed{\blacktriangle}$$
$$\boxed{\blacktriangledown}\boxed{\blacktriangle}\boxed{\blacktriangle} \approx_{85.71} \boxed{\blacktriangle}\boxed{\blacktriangle}$$

(The 'missing' two are $\boxed{\blacktriangle}\boxed{\blacktriangledown}\boxed{\blacktriangledown} \sim \boxed{\blacktriangledown}\boxed{\blacktriangledown}$ and $\boxed{\blacktriangledown}\boxed{\blacktriangledown}\boxed{\blacktriangle} \sim \boxed{\blacktriangle}\boxed{\blacktriangle}\boxed{\blacktriangledown}$ — they are true with thresholds $\frac{11}{14} \approx 79\%$ and $\frac{5}{7} \approx 71\%$ respectively, but if we reduce theorem discovery's threshold far enough to find *them*, we find many similar theorems and it becomes difficult to glean any insight.)

The BBraun number entry routine has non-regular behaviour near its minimum values, as described in section 4.3.1.3 — in this example, the minimum value in question is 0.1, and looking at figure 4.12, we see a number of one-way edges leading to states labelled with the value 0.1, and it is these edges that lead to the partiality just described.

6.4.2 Tenths, hundredths only; digits 0–2 and 9 only

These models are as described in section 4.3.3.2: this sub-space is essentially the same as the previous example, but extended to include exploration of the 9 digit. The results of theorem discovery are accordingly similar, with a few differences.

The Simple Spinner routine (figure 4.14, page 82) has the same four length-3 theorems described in the previous section:

$$\begin{gathered}
\boxed{\blacktriangleleft}\boxed{\blacktriangledown}\boxed{\blacktriangleleft} \sim \boxed{\blacktriangleleft}\boxed{\blacktriangledown} \\
\boxed{\blacktriangleleft}\boxed{\blacktriangle}\boxed{\blacktriangleleft} \sim \boxed{\blacktriangleleft}\boxed{\blacktriangle} \\
\boxed{\blacktriangleright}\boxed{\blacktriangledown}\boxed{\blacktriangleright} \sim \boxed{\blacktriangleright}\boxed{\blacktriangledown} \\
\boxed{\blacktriangleright}\boxed{\blacktriangle}\boxed{\blacktriangleright} \sim \boxed{\blacktriangleright}\boxed{\blacktriangle}
\end{gathered}$$

but of the seven length-3 theorems that, in the previous example, reflected the model's vertical restriction to the digits 0–2, only two remain:

$$\begin{gathered}
\boxed{\blacktriangledown}\boxed{\blacktriangle}\boxed{\blacktriangledown} \sim \boxed{\blacktriangledown} \\
\boxed{\blacktriangle}\boxed{\blacktriangledown}\boxed{\blacktriangle} \sim \boxed{\blacktriangle}
\end{gathered}$$

This is understandable, as the vertical restriction has been relaxed by adding another digit, i.e. 9, so the state space is 3 digits 'high'. If we were to consider theorems of length 4 we would find some theorems of similar character (not shown).

The Simple Arithmetic routine (figure 4.15, page 83) was identical to the Simple Spinner in the previous example, but in this case it also has this partial theorem:

$$\boxed{\blacktriangle}\boxed{\blacktriangle}\boxed{\blacktriangle} \approx_{91.67} \boxed{\blacktriangle}\boxed{\blacktriangle}$$

This theorem is also due to the vertical extension of the model by allowing the digit 9, and can be intuitively understood by looking at the graph of the model. In about 8% of the model's states (specifically, those where the current digit is 9) $\boxed{\blacktriangle}\boxed{\blacktriangle}$ changes that digit to 1, and $\boxed{\blacktriangle}\boxed{\blacktriangle}\boxed{\blacktriangle}$ changes it to 2; in all other states this is not the case, and $\boxed{\blacktriangle}\boxed{\blacktriangle}$ takes the system to a state where $\boxed{\blacktriangle}$ is a self-loop.

In the Simple Spinner, this partial theorem is true with similarity exactly $\frac{3}{4}$ — another reflection of the regularity of that model: in exactly $\frac{1}{4}$ of the model's states, the current digit is 9 (and in exactly $\frac{1}{4}$ it is 0, and so on.) Looking at consecutive $\boxed{\blacktriangledown}$ actions, we find that in the Simple Spinner $\boxed{\blacktriangledown}\boxed{\blacktriangledown}\boxed{\blacktriangledown} \approx_{\frac{3}{4}}$ $\boxed{\blacktriangledown}\boxed{\blacktriangledown}$, and for Simple Arithmetic, the similarity is $\frac{19}{24} \approx 79\%$.

Interestingly, in this example, the BBraun routine's model (figure 4.16, page 85) has exactly the same theorems as the Simple Spinner (up to length 3 with threshold 90%, at least).

6.4.3 Hundredths, full

These models are as described in section section 4.3.3.3: this sub-space contains only the tenths and hundredths digits, but includes the full range 0–9 for each of those digits. In some sense this is the most interesting example in this section, and certainly the most realistic: although the state space is still only two digits 'wide', it is now at the full 'depth' of ten digits, and so provides a more genuine and reflective 'slice' of the whole state space of each system.

The first — and main — thing to note is that in the Simple Spinner model (figure 4.18(a), page 87) we finally have that $\boxed{\blacktriangledown}$ and $\boxed{\blacktriangle}$ are inverses:

$$\boxed{\blacktriangledown}\boxed{\blacktriangle} \sim \lambda$$
$$\boxed{\blacktriangle}\boxed{\blacktriangledown} \sim \lambda$$

This makes perfect sense given our understanding of how the Simple Spinner works, and the presence of these theorems is a reassuring confirmation that the implementation is correct. For the Simple Arithmetic routine (figure 4.18(b), page 87) we find these equivalences only hold partially, with similarity 94.5%, and for the BBraun routine (figure 4.18(c), page 87), they hold with similarity 93.96%, providing concrete analytical evidence of these models' non-regularity in this respect, and arising again from behaviour around the minimum value.

From a designer's point of view, this could be an important finding: users might well desire or expect that $\boxed{\blacktriangledown}$ and $\boxed{\blacktriangle}$ are inverses, and in these models that is *almost* but not quite true. Whether this actually leads to surprises and greater risk of error in practice is clearly a matter for empirical study, and beyond the scope of this thesis, but we argue that this is a concrete and realistic example of theorem discovery finding something that *could* be worthy of such study.

Going on to consider theorems of length 3, and raising the similarity threshold to 95% in recognition of the larger state spaces of these models, we find the following for the Simple Spinner, none of which are particularly interesting or surprising:

$$\begin{gathered}
\boxed{\blacktriangleleft}\boxed{\blacktriangledown}\boxed{\blacktriangleleft} \sim \boxed{\blacktriangleleft}\boxed{\blacktriangledown} \\
\boxed{\blacktriangleleft}\boxed{\blacktriangle}\boxed{\blacktriangleleft} \sim \boxed{\blacktriangleleft}\boxed{\blacktriangle} \\
\boxed{\blacktriangleright}\boxed{\blacktriangledown}\boxed{\blacktriangleright} \sim \boxed{\blacktriangleright}\boxed{\blacktriangledown} \\
\boxed{\blacktriangleright}\boxed{\blacktriangle}\boxed{\blacktriangleright} \sim \boxed{\blacktriangleright}\boxed{\blacktriangle}
\end{gathered}$$

whereas for the other two routines we find those four theorems plus the following (Simple Arithmetic on the left, BBraun on the right). Rather than laboriously detailing the implications of each of these, we simply note that there is a great deal of commonality between the two sets of theorems, i.e. the BBraun routine acts very similarly to a Simple Arithmetic implementation — indeed that *is* essentially how it appears to operate to a user — but that there are clearly some differences 'lurking' in corner cases (specifically around minimum values, as previously noted). Once again, we argue that theorem discovery has raised an interesting question to consider for focused empirical study, and we suggest that ordinary user testing would probably not notice this as a potential issue, given the size of the state space to be explored: by exhaustively exploring the whole model, theorem discovery draws our attention onto the parts that deviate from regularity or expectation.

$\boxed{\blacktriangledown}\boxed{\blacktriangle}\boxed{\blacktriangledown} \sim \boxed{\blacktriangledown}$	$\boxed{\blacktriangledown}\boxed{\blacktriangle}\boxed{\blacktriangledown} \sim \boxed{\blacktriangledown}$
$\boxed{\blacktriangleleft}\boxed{\blacktriangledown}\boxed{\blacktriangleleft} \sim \boxed{\blacktriangleleft}\boxed{\blacktriangledown}$	$\boxed{\blacktriangleleft}\boxed{\blacktriangledown}\boxed{\blacktriangleleft} \sim \boxed{\blacktriangleleft}\boxed{\blacktriangledown}$
$\boxed{\blacktriangleleft}\boxed{\blacktriangle}\boxed{\blacktriangleleft} \sim \boxed{\blacktriangleleft}\boxed{\blacktriangle}$	$\boxed{\blacktriangleleft}\boxed{\blacktriangle}\boxed{\blacktriangleleft} \sim \boxed{\blacktriangleleft}\boxed{\blacktriangle}$
$\boxed{\blacktriangleright}\boxed{\blacktriangledown}\boxed{\blacktriangleright} \sim \boxed{\blacktriangleright}\boxed{\blacktriangledown}$	$\boxed{\blacktriangleright}\boxed{\blacktriangledown}\boxed{\blacktriangleright} \sim \boxed{\blacktriangleright}\boxed{\blacktriangledown}$
$\boxed{\blacktriangleright}\boxed{\blacktriangle}\boxed{\blacktriangleright} \sim \boxed{\blacktriangleright}\boxed{\blacktriangle}$	$\boxed{\blacktriangleright}\boxed{\blacktriangle}\boxed{\blacktriangleright} \sim \boxed{\blacktriangleright}\boxed{\blacktriangle}$
$\boxed{\blacktriangle}\boxed{\blacktriangledown}\boxed{\blacktriangle} \sim \boxed{\blacktriangle}$	$\boxed{\blacktriangle}\boxed{\blacktriangledown}\boxed{\blacktriangle} \sim \boxed{\blacktriangle}$
$\boxed{\blacktriangleleft}\boxed{\blacktriangledown}\boxed{\blacktriangle} \approx_{95.00} \boxed{\blacktriangledown}\boxed{\blacktriangle}\boxed{\blacktriangleleft}$	$\boxed{\blacktriangleleft}\boxed{\blacktriangledown}\boxed{\blacktriangle} \approx_{95.05} \boxed{\blacktriangledown}\boxed{\blacktriangle}\boxed{\blacktriangleleft}$
$\boxed{\blacktriangleright}\boxed{\blacktriangledown}\boxed{\blacktriangle} \approx_{95.00} \boxed{\blacktriangledown}\boxed{\blacktriangle}\boxed{\blacktriangleright}$	$\boxed{\blacktriangleright}\boxed{\blacktriangledown}\boxed{\blacktriangle} \approx_{95.05} \boxed{\blacktriangledown}\boxed{\blacktriangle}\boxed{\blacktriangleright}$
$\boxed{\blacktriangle}\boxed{\blacktriangledown}\boxed{\blacktriangleleft} \approx_{95.00} \boxed{\blacktriangleleft}\boxed{\blacktriangle}\boxed{\blacktriangledown}$	
$\boxed{\blacktriangle}\boxed{\blacktriangledown}\boxed{\blacktriangleright} \approx_{95.00} \boxed{\blacktriangleright}\boxed{\blacktriangle}\boxed{\blacktriangledown}$	
$\boxed{\blacktriangleright}\boxed{\blacktriangledown}\boxed{\blacktriangle} \approx_{98.00} \boxed{\blacktriangleright}\boxed{\blacktriangle}\boxed{\blacktriangledown}$	$\boxed{\blacktriangleright}\boxed{\blacktriangledown}\boxed{\blacktriangle} \approx_{97.80} \boxed{\blacktriangleright}\boxed{\blacktriangle}\boxed{\blacktriangledown}$
$\boxed{\blacktriangleright}\boxed{\blacktriangledown}\boxed{\blacktriangle} \approx_{99.00} \boxed{\blacktriangleright}$	$\boxed{\blacktriangleright}\boxed{\blacktriangledown}\boxed{\blacktriangle} \approx_{98.90} \boxed{\blacktriangleright}$
$\boxed{\blacktriangleright}\boxed{\blacktriangle}\boxed{\blacktriangledown} \approx_{99.00} \boxed{\blacktriangleright}$	$\boxed{\blacktriangleright}\boxed{\blacktriangle}\boxed{\blacktriangledown} \approx_{98.90} \boxed{\blacktriangleright}$

6.5 Case study 3: Casio HS-8V

Our final case study is the Casio HS-8V desktop calculator, a device that is sufficiently complex to exhibit interesting properties, and of a class that has been very widely studied in the psychological, HCI and programming literature (e.g. [MB81]); arguably, we should find nothing new of note. Section 4.5 describes the device (pictured in figure 4.22, page 95) and how model discovery was used to produce two models of its number entry behaviour. To review, in each model there are the two actions $\boxed{1}$ and $\boxed{\bullet}$, and the two models are:

- The *device* model (figure 4.23(a), page 96), in which the full internal state of the simulation is projected, including not only the display but also a value indicating whether the $\boxed{\bullet}$ key has been pressed recently.

- The *user* model (figure 4.23(b), page 96), in which only the contents of the device's display are projected, so that this model represents the 'user visible' device behaviour.

(Note that in figure 4.23, $\boxed{1}$ actions are shown as *d*, for 'digit'.)

As noted in section 4.5, the device model is deterministic, but the user model is (unusually) nondeterministic — and much of the discussion in that section is about how that nondeterminstic model was produced.

Here we introduce the notation $\boxed{\times}^y$ for y presses of the key $\boxed{\times}$, so $\boxed{\times}^0 \sim \lambda$.

6.5.1 Theorem discovery on the HS-8V user model

Running theorem discovery on the HS-8V user model, and allowing it to run to completion, i.e. to find all theorems, yields two total equivalences:

$$\boxed{\bullet} \sim \lambda$$
$$\boxed{1}^9 \sim \boxed{1}^8$$

6.5.1.1 The theorem: $\boxed{\bullet} \sim \lambda$

The first of these is extremely surprising, and was initially thought to have been caused by a bug in the algorithm or implementation; it is, however, true. Put simply: in the user model, the decimal point key does nothing. As states in the user model correspond to what is visible to the user, that suggests that in the actual device, pressing the decimal point key *never changes the display.* This is indeed true, both of the HS-8V simulation used to produce the model, and (as far as we can tell) of the actual device itself.

We argue that this is, probably, the most surprising statement in this thesis.

What's going on here? First, we can verify that the theorem is true by visual inspection of the model; looking at figure 4.23(b), it is easy to visually check that in every state, there is exactly one edge labelled with a decimal point, and it is a self-loop: $\boxed{\bullet}$ actions really do never change the state, in this model. Why, then in the device, does the $\boxed{\bullet}$ never change the display?

When the HS-8V is switched on, although the user has keyed nothing it displays 0., as noted in [TO09]. If the user subsequently keys $\boxed{0}$ or $\boxed{\bullet}$, the display remains unchanged. Therefore with a display of 0., the user cannot tell if they (or somebody else) has previously keyed $\boxed{0}$ or $\boxed{\bullet}$; if they then press $\boxed{5}$ (say) the resultant number could be either 0.5 or 5.0.

What our surprising total equivalence shows us is that *whenever* the user keys a decimal point, whatever the input history and whatever the display currently shows, the display remains unchanged. This is true precisely because the decimal point is already visible when the device is switched on: because it's visible, pressing the $\boxed{\bullet}$ key any time in the future doesn't cause it to appear, and any subsequent $\boxed{\bullet}$ keypresses of course have no effect at all.

That the display remains unchanged is not only surprising, but also quite worrying. The feedback usability heuristic [Nie93] is broken, and in general users cannot predict the effect of a keypress on the device's behaviour simply by looking at its display. [TC10] points out that multiple $\boxed{\bullet}$ presses leave the display unchanged (after the first one, that is), and argues the dangers of this design defect. Theorem discovery has found that the defect is more thoroughgoing — a problem that has gone unnoticed for years, although it is easy to fix. (We find that more elaborate calculators, particularly ones with graphing capability and more complex input modalities, do not exhibit this behaviour, but that the vast majority of both basic desktop and scientific models we examined still do.) This lack of feedback is not trivial; in [MB81], the authors note (in regard to a study of how users build mental models of the operation of a desktop calculator):

> *...one student, for example, concluded that the plus (+) and equals (=) keys did nothing since they caused no visible change in the display.*

Our approach has *automatically* found a design defect which had been overlooked until now (and as far as we can tell, most basic desktop calculators produced in the last 30–40 years have behaved like this). As such, it is hard to imagine performing experiments that would have uncovered it. Having discovered it, it is easy to propose solutions and, for example, plan A/B empirical studies: which designs do users prefer, and which lead to fewer errors? The problem may appear small and trivial, but we would disagree; while rare and unrecognised, it could have major consequences (as noted in section 4.5, desktop calculators of this kind are routinely used in hospitals for drug dosage calculations, for example). Although mode confusion (such as decimal point confusion) is a known design problem, there is no empirical data available on whether adverse incidents are caused by confusion with displaying a decimal point regardless of how many have been keyed by the user. It is easy to envisage usability experiments to quantify the actual error rates, and a sufficiently long experiment would obviously establish $p > 0$.

One could then reflect on the cost/benefit ratios to decide whether to implement an improved design.

6.5.1.2 The theorem: $\boxed{1}^9 \sim \boxed{1}^8$

The second total equivalence found by theorem discovery is $\boxed{1}^9 \sim \boxed{1}^8$. This is unsurprising from our point of view as analysts, as it clearly reflects the display size of the calculator: once the user has entered eight digits, further keypresses have no effect, simply because the display is full. Of course, such behaviour might be surprising to an end user, whose actions are silently discarded: following [Thi12] we would recommend explicitly raising an error in such cases.

This theorem manifests in figure 4.23(b) as the d-labelled self-loop on the leaf state at the very bottom-right. This observation raises an obvious question: why isn't there a similar theorem corresponding to each of the other leaf states, all of which have a d-labelled self-loop? For example, why do we not have:

$$\boxed{\bullet}\boxed{1}^9 \sim \boxed{\bullet}\boxed{1}^8$$

(which manifests as the d-labelled self-loop on the leaf state at the very bottom-left), or:

$$\boxed{1}\boxed{\bullet}\boxed{1}^8 \sim \boxed{1}\boxed{\bullet}\boxed{1}^7$$

(which manifests as the d-labelled self-loop on the next leaf state to the right)?

Those theorems are being pruned: when $\boxed{\bullet} \sim \lambda$ is discovered on the first iteration of the theorem discovery algorithm, $\boxed{\bullet}$ is added to the equivalence class of λ and thus not to *bases'*, and thus does not appear in *bases* on the next iteration — so no strings beginning with $\boxed{\bullet}$ get checked by the algorithm beyond that point. This accounts for the absence of $\boxed{\bullet}\boxed{1}^9 \sim \boxed{\bullet}\boxed{1}^8$. Similarly, because $\boxed{\bullet}$ has been added to an equivalence class, on the second iteration of the algorithm, the string $\boxed{1}\boxed{\bullet}$ is not checked because of *suffix*-pruning (i.e. on $\boxed{\bullet}$); thus, $\boxed{1}\boxed{\bullet}$ has no chance to be added to *bases'*, so none of its suffices are checked, including $\boxed{1}\boxed{\bullet}\boxed{1}^8$. Similar arguments hold for $\boxed{1}^2\boxed{\bullet}\boxed{1}^6$, etc.

6.5.2 Theorem discovery on the HS-8V device model

Running theorem discovery on the device model yields 18 theorems, which may be grouped into two families. First, we have nine theorems:

$$\forall\, 0 \leqslant n \leqslant 8 : \boxed{\bullet}\,\boxed{1}^n\,\boxed{\bullet} \sim \boxed{\bullet}\,\boxed{1}^n$$

These theorems can be interpreted together as meaning *after pressing the* $\boxed{\bullet}$ *key once, subsequent presses of it have no effect.* Of course, in the device model, $[\![\boxed{\bullet}]\!]$ is *not* the identity: while it has no user-visible effect, pressing $\boxed{\bullet}$ (for the first time) has an internal effect on the system. (In the simulation whose models we are analysing, it sets a flag indicating that this was the most recent keypress: presumably in the actual device the mechanism is similar.) So, these nine theorems are the device's model's analogue of $\boxed{\bullet} \sim \lambda$ in the user model.

Then, we have again:

$$\boxed{1}^9 \sim \boxed{1}^8$$

and these eight theorems:

$$\forall\, 1 \leqslant n \leqslant 8 : \boxed{1}^n\,\boxed{\bullet}\,\boxed{1}^{(9-n)} \sim \boxed{1}^n\,\boxed{\bullet}\,\boxed{1}^{(8-n)}$$

These nine theorems can be interpreted together as meaning *after eight digits have been entered, subsequent keypresses have no effect.* Thus, they are the device model's analogue of the singular theorem $\boxed{1}^9 \sim \boxed{1}^8$ in the user model, and they represent the limited size of the device's display. That truth manifests here as these nine theorems rather than just one because the pruning discussed in section 6.5.1.2 doesn't take place here, because $[\![\boxed{\bullet}]\!]$ is not the identity.

Now, neither of these insights is particular deep or surprising, but they do accurately reflect two aspects of the device that we would expect to be true. The main thing these theorems illustrate, we propose, is that for some purposes, it is very useful to work on models that only project user-visible state. Theorem discovery on the user model yields two theorems which may be very clearly interpreted, whereas in the device model, interpreting the theorems (to reach essentially the same conclusions) takes more effort.

Chapter 7

Conclusions

Contents

In this chapter we present a summary and discussion of the contributions of the thesis, and note some potential avenues for future work.

7.1 Contributions of the thesis

In this thesis we have described, formalised, and exemplified two techniques for modelling and analysing interactive computerised systems in a manner which is rigorous, automated, and easily integrated into existing software development workflows; the thesis builds in particular on [GT10], [GTC06] and [GT13], extending and deepening that work as described below. These techniques are motivated by the necessity, but difficulty, of bringing formalised approaches to bear on the problem of creating interactive systems, and in particular by the desire to think about such systems in terms of issues related to dependability, rather than more 'traditional' HCI concerns such as ease of use.

The presented techniques are not proposed as a replacement or alternative for traditional HCI methods centred on actual users; rather, we suggest that they are valuable as an adjunct to such methods, in that they operate exhaustively, automatically, and inexpensively, and that they can in fact support such methods by raising unforeseen issues whose likelihood and severity may then be investigated further with actual users and traditional techniques. Similarly, we do not propose to replace existing formal methods as applied to interactive systems, but rather to complement them by providing formalised techniques for analysing implementations rather than specifications — that is, our techniques provide insight into the interaction behaviour of actual running systems. As such, we propose, they can be of particular use as infrastructure for performing regression tests on interaction behaviour (say of properties arising from formal specifications), as well as providing insight into unforeseen properties which ought to be the subject of such tests. Overall, they add new layers of defence against error to those already routinely used by developers, and those provided by formal methods.

The first technique, model discovery, automatically produces a state space model of the interaction behaviour of a running system by means of a dynamic analysis simulating a user performing all possible actions in all possible states of the system — or, in practice, some subset of actions in some subset of states. Such models may be analysed in several ways, including by application of theorem discovery, the other technique described in the thesis. We have explained model discovery in detail, described the main tasks and concerns of a programmer aiming to implement it in some context, and formalised those requirements in a specification of an API for model discovery, and an algorithm based on that API. We have described a number of variants and extensions of that basic API and algorithm, some of which we have implemented. We have demonstrated the use and utility of the model discovery technique by describing in detail two case studies of its application to example systems of real world complexity, and critiquing its application in two other cases. We have described implementation details where pertinent or of particular interest, and have explored a number of the extensions suggested to the basic algorithm, in particular conditional exploration.

The second technique, theorem discovery, analyses a state space model of the interaction behaviour of a system in order to identify sequences of user actions which are equivalent — or nearly so — in their effect on the system. This technique complements model discovery in that it is tailored for exactly the kind of model produced by that technique, but it is also applicable to models produced by other means, and could, we suggest, be modified to operate on other kinds of models, including ones with more structure. We have informally and formally described the technique and the concepts upon which it is based, and presented a formal description of an algorithm implementing it, including a discussion of techniques for pruning its search space in order to reduce unwanted redundancy in its output, and an informal worst-case performance analysis. Building on this, we have described a number of commonly occurring syntactic patterns of theorems and their meaning, and considered the general problem of analysis of families of theorems. We have described the implementation of a library and prototype command-line tool for performing theorem discovery, and we have demonstrated the use and utility of the theorem discovery technique by describing three case studies of the application of this tool to models produced using model discovery and discussed earlier in the thesis.

It is our hope that the work presented in this thesis may form a basis for further investigation and application of these techniques, both in terms of research aimed at answering questions raised herein, and of application to real world examples in an industrial context.

7.2 Future work

We conclude by summarising future work which we have suggested in order to build on that presented in this thesis.

Model discovery

In section 3.4.3 we described a number of variants and extensions to model discovery, some of which we have begun to explore in this thesis, but much future work remains to be done in this area. In particular, while we have demonstrated the use of *conditional exploration* (e.g. in section 4.3), the more general technique of directed/scripted exploration (as described in section 3.4.3.7) seems an obvious area to explore, and one which we believe would be very fruitful. We have, throughout, expressed our belief that model discovery could be usefully integrated into existing development toolchains including IDEs such as Eclipse[1]; providing a well-defined interface to model discovery via a DSL (essentially wrapping the API) would be a useful step in such integration, and could then provide the basis for allowing directed exploration in that context. More generally, the question of how best to expose advanced control

[1]`http://eclipse.org`

features such as directed and conditional exploration to the investigating programmer, is an interesting an unanswered question.

One of the key tasks of the investigating programmer who wishes to apply model discovery in some context is to choose an appropriate projection of SUD state for the model they are interested in; as described in section 3.2.1 this projection needs to be *sound* to allow proper backtracking, and as suggested in section 3.4.2, one possible way to check for unsound projections is to repeat a model discovery task *stochastically* — either as a post-processing check or on-the-fly as the model is discovered. Similarly, as suggested in section 4.5, adding actions to the pool more than once per state could serve as such a soundness check. However, neither of these techniques have been investigated for their utility and implications as yet.

Going on from this, we have noted that due to this soundness requirement, models can sometimes unavoidably include aspects of state which the analyst is actually uninterested in, from the point of view of the analysis task they wish to perform. As such, we propose that there exists an as-yet-uninvestigated space of post-processing tasks which could be useful in order to transform the 'bare product' of model discovery into more focused models. For example, by removing all trace of the temperature control from the model produced of the air conditioning control system, and collapsing similar states, we can produce a model identical to that which would have been produced by performing model discovery ignoring the temperature control slider altogether.

Finally, model discovery as described in this thesis aims to produce models of the user-interaction behaviour of systems; however, as described in section 4.2.1, actual systems often (indeed usually) also interact with their environment (as distinct from the user), and in many cases act to control a physical *plant*. We have suggested that while applying model discovery, tracking environment/plant state is inadvisable, not least because it adds to the model and thus distracts from or obscures the interaction behaviour which we are nominally interested in. However, there is no conceptual reason why the model discovery technique could not be adapted to such a style, and investigation of its utility for building models of the more general behaviour of systems, could be interesting future work.

Theorem discovery

The theorem discovery algorithm and implementation presented here are prototypical. In section 6.2.2.2 we suggested some performance optimisations which could be applied to the algorithm in particular exploiting concurrency where appropriate, and using locality-sensitive hashing to improve the speed of partial equivalence checking. In terms of the actual algorithm structure, as noted in section 6.3.2, the current pruning techniques do not adequately reduce redundancy of reported partial theorems; how to extend the current pruning mechanisms to do so is thus interesting and necessary future work.

In section 5.2.4 we describe the computation of similarity as the basis of partial equivalence theorems, but our similarity metric weights all model states equally; as noted in that section, one possible extension to this would be to weight states in this computation according to their frequency of visitation by users — as informed either by empirical study, Markov methods, or a combination of both.

We have identified several areas where the theorem discovery *infrastructure* may be of use in and of itself, separately from the theorem discovery algorithm. In section 6.3.1 we described issues around testing for the existence of theorems, and described why simply running the algorithm and looking for a theorem in its output is inadequate for this task. In section 5.5 we also identified several cases where the algorithm itself does not quite provide a particular insight, but specialised application of the infrastructure can do so — for example inverse actions (section 5.5.3) and undo (section 5.5.4). More generally, the task of identifying and interpreting (or aiding the semi-automatic interpretation of) meta-theorems and families of theorems remains future work.

In section 6.2.3 we suggested a number of avenues for improving the theorem discovery user experience. In particular: visual improvements (e.g. re-implementation or wrapping in a GUI); filtering theorems on symbols and substrings; ability to save/load an 'ignore' list, in order to support multiple runs of the algorithm; and 'closing the loop' with respect to states, in order to fully support insights provided by partial equivalence theorems.

Finally, as discussed in section 6.3.2, we might consider redefining the semantics of theorem discovery to allow non-total models, which raises some interesting questions. For example, does this have any major effects on the structure of the algorithm, or on the task of interpreting the theorems produced?

Bibliography

[AWM95] Gregory D. Abowd, Hung-Ming Wang and Andrew F. Monk. 'A formal technique for automated dialogue development'. In: *Proceedings of the 1st conference on Designing interactive systems: processes, practices, methods, & techniques*. DIS '95. Ann Arbor, Michigan, USA: ACM, 1995, pp. 219–226 (cit. on p. 18).

[AP11] Darren Abramson and Lee Pike. 'When formal systems kill: computer ethics and formal methods'. In: *APA Newsletter on Philosophy and Computers*. Vol. 11. 1. American Philosophy Association, 2011 (cit. on p. 8).

[ATO10] Chitra Acharya, Harold Thimbleby and Patrick Oladimeji. 'Human computer interaction and medical devices'. In: *Proceedings of the 24th BCS Interaction Specialist Group Conference*. BCS '10. Dundee, United Kingdom: British Computer Society, 2010, pp. 168–176 (cit. on p. 28).

[ABL02] Glenn Ammons, Rastislav Bodík and James R. Larus. 'Mining specifications'. In: *SIGPLAN Not.* 37.1 (2002), pp. 4–16 (cit. on p. 20).

[ADK03] Alexandr Andoni, Dumitru Daniliuc and Sarfraz Khurshid. *Evaluating the "Small Scope Hypothesis"*. Tech. rep. MIT Laboratory for Computer Science, 2003 (cit. on pp. 30, 122).

[BK07] Michael Bächle and Paul Kirchberg. 'Ruby on Rails'. In: *IEEE Software* 24.6 (2007), pp. 105–108 (cit. on p. 2).

[Bar+09] Clark Barrett, Roberto Sebastiani, Sanjit Seshia and Cesare Tinelli. 'Satisfiability Modulo Theories'. In: *Handbook of Satisfiability*. Ed. by Armin Biere, Marijn J. H. Heule, Hans van Maaren and Toby Walsh. Vol. 185. IOS Press, 2009. Chap. 26, pp. 825–885 (cit. on p. 16).

[Bas+11] Ellen J. Bass, Karen M. Feigh, Elsa L. Gunter and John M. Rushby. 'Formal Modeling and Analysis for Interactive Hybrid Systems'. In: *ECEASST* 45 (2011) (cit. on p. 16).

[BNP03] Rémi Bastide, David Navarre and Philippe Palanque. 'A Tool-Supported Design Framework for Safety Critical Interactive Systems'. In: *Interacting with computers* 15.3 (2003), pp. 309–328 (cit. on p. 19).

[Bas+99] Rémi Bastide, Philippe Palanque, Ousmane Sy, Duc-Hoa Le and David Navarre. 'Petri Net Based Behavioural Specification of CORBA Systems'. In: *Application and Theory of Petri Nets 1999*. Ed. by Susanna Donatelli and Jetty Kleijn. Vol. 1639. LNCS. Springer Berlin Heidelberg, 1999, pp. 66–85 (cit. on p. 19).

[Bec03] Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003 (cit. on p. 2).

[Ben38] Frank Benford. 'The Law of Anomalous Numbers'. In: *Proceedings of the American Philosophical Society* 78.4 (1938), pp. 551–572 (cit. on p. 28).

[BM04] Michel Bidoit and Peter D. Mosses. CASL *User Manual*. Vol. 2900. LNCS. Springer, 2004 (cit. on p. 42).

[Bla+08] Ann E. Blandford, Joanne K. Hyde, Thomas R. G. Green and Iain Connell. 'Scoping Analytical Usability Evaluation Methods: A Case Study'. In: *Human-Computer Interaction* 23.3 (2008), pp. 278–327 (cit. on p. 13).

[BB10] Matthew L. Bolton and Ellen J. Bass. 'Formally verifying human-automation interaction as part of a system model: limitations and tradeoffs'. In: *Innov. Syst. Softw. Eng.* 6.3 (2010), pp. 219–231 (cit. on p. 13).

[BH95] J.P. Bowen and M.G. Hinchey. 'Seven more myths of formal methods'. In: *IEEE Software* 12.4 (1995), pp. 34–41 (cit. on p. 9).

[BR10] Judy Bowen and Steve Reeves. 'Developing usability studies via formal models of UIs'. In: *Proceedings of the 2nd ACM SIGCHI symposium on Engineering interactive computing systems*. EICS '10. ACM, 2010, pp. 175–180 (cit. on p. 19).

[BF99] Howard Bowman and Giorgio Faconti. 'Analysing Cognitive Behaviour using LOTOS and Mexitl'. In: *Formal Aspects of Computing* 11.2 (1999), pp. 132–159 (cit. on p. 13).

[Bum+95] Peter Bumbulis, P.S.C. Alencar, D.D. Cowan and C.J.P. Lucena. 'Combining Formal Techniques and Prototyping in User Interface Construction and Verification'. In: *Design, Specification and Verification of Interactive Systems '95*. Ed. by Philippe Palanque and Rémi Bastide. Eurographics. Springer Vienna, 1995, pp. 174–192 (cit. on p. 12).

[CH97] José Creissac Campos and Michael D. Harrison. 'Formally Verifying Interactive Systems: A Review'. In: *Design, Specification and Verification of Interactive Systems '97*. Ed. by Michael D. Harrison and Juan Carlos Torres. Eurographics. Springer Vienna, 1997, pp. 109–124 (cit. on p. 17).

[CH98] José Creissac Campos and Michael D. Harrison. 'The Role of Verification in Interactive Systems Design'. In: *Interactive Systems. Design, Specification, and Verification. Proceedings of the 5th International Workshop, DSV-IS 1998*. Ed. by Panos Markopoulos and Peter Johnson. Springer, 1998, pp. 155–170 (cit. on p. 17).

[CH01] José Creissac Campos and Michael D. Harrison. 'Model Checking Interactor Specifications'. In: *Automated Software Engineering* (2001) (cit. on p. 17).

[CH08] José Creissac Campos and Michael D. Harrison. 'Systematic Analysis of Control Panel Interfaces Using Formal Tools'. In: *Interactive Systems. Design, Specification, and Verification. Proceedings of the 15th International Workshop, DSV-IS 2008*. Ed. by T.C. Nicholas Graham and Philippe Palanque. Vol. 5136. LNCS. Springer, 2008, pp. 72–85 (cit. on p. 17).

[CH09] José Creissac Campos and Michael D. Harrison. 'Interaction engineering using the IVY tool'. In: *Proceedings of the 1st ACM SIGCHI symposium on Engineering interactive computing systems*. EICS '09. ACM, 2009, pp. 35–44 (cit. on p. 17).

[CH11] José Creissac Campos and Michael D. Harrison. 'Modelling and analysing the interactive behaviour of an infusion pump'. In: *Proceedings of the 4th International Workshop on Formal Methods for Interactive Systems*. EASST, 2011 (cit. on p. 17).

[CD07] Gerardo Canfora and Massimiliano Di Penta. 'New Frontiers of Reverse Engineering'. In: *2007 Future of Software Engineering*. FOSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 326–341 (cit. on p. 20).

[CNM83] Stuart K. Card, Allen Newell and Thomas P. Moran. *The Psychology of Human-Computer Interaction*. Hillsdale, NJ, USA: L. Erlbaum Associates Inc., 1983 (cit. on p. 12).

[CLM03] Keith Chan, Zhi Cong Leo Liang and Amir Michail. 'Design recovery of interactive graphical applications'. In: *Proceedings of the 25th International Conference on Software Engineering*. ICSE '03. Portland, Oregon: IEEE Computer Society, 2003, pp. 114–124 (cit. on p. 21).

[Cha+92] E. Moll van Charante, R.I. Cook, D.D. Woods, L. Yue and M.B. Howie. 'Human-Computer Interaction in context: Physician interaction with automated intravenous controllers in the heart room'. In: *Analysis, design, and evaluation of man-machine systems 1992*. Ed. by H.G. Stassen. Pergamon Press, 1992, pp. 263–274 (cit. on p. 26).

[Cho+11] A. Tankeu Choitat, J.-C. Fabre, P. Palanque, D. Navarre and Y. Deleris. 'Self-checking widgets for interactive cockpits'. In: *Proceedings of the 13th European Workshop on Dependable Computing*. EWDC '11. Pisa, Italy: ACM, 2011, pp. 43–48 (cit. on p. 19).

[CH00] Koen Claessen and John Hughes. 'QuickCheck: a lightweight tool for random testing of Haskell programs'. In: *Proceedings of the 5th ACM SIGPLAN international conference on Functional programming*. ICFP '00. New York, NY, USA: ACM, 2000, pp. 268–279 (cit. on p. 23).

[CGP99] E. M. Clark, O. Grumberg and D. Peled. *Model Checking*. MIT Press, 1999 (cit. on p. 14).

[CES86] E. M. Clarke, E. A. Emerson and A. P. Sistla. 'Automatic verification of finite-state concurrent systems using temporal logic specifications'. In: *ACM Trans. Program. Lang. Syst.* 8.2 (1986), pp. 244–263 (cit. on pp. 14, 15).

[CW96] Edmund M. Clarke and Jeannette M. Wing. 'Formal methods: state of the art and future directions'. In: *ACM Comput. Surv.* 28.4 (1996), pp. 626–643 (cit. on pp. 1, 8).

[CP09] Sébastien Combéfis and Charles Pecheur. 'A bisimulation-based approach to the analysis of human-computer interaction'. In: *Proceedings of the 1st ACM SIGCHI symposium on Engineering interactive computing systems*. EICS '09. ACM, 2009, pp. 101–110 (cit. on p. 13).

[CC77] Patrick Cousot and Radhia Cousot. 'Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints'. In: *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages.* POPL '77. Los Angeles, California: ACM, 1977, pp. 238–252 (cit. on p. 30).

[Cou87a] Joëlle Coutaz. 'PAC, an Object Oriented Model for Dialog Design'. In: *Proceedings of the 2nd IFIP International Conference on Human-Computer Interaction (INTERACT 87).* Ed. by Hans-Jörg Bullinger and Brian Shackel. 1987, pp. 431–436 (cit. on p. 31).

[Cou87b] Joëlle Coutaz. 'PAC: An Object Oriented Model For Implementing User Interfaces'. In: *SIGCHI Bulletin* 19.2 (1987) (cit. on pp. 12, 31).

[Cro06] Douglas Crockford. *RFC 4627 - The application/json media type for JavaScript Object Notation.* Tech. rep. 2006 (cit. on p. 40).

[CH07] Daniela Da Cruz and Pedro Rangel Henriques. 'Slicing wxHaskell modules to derive the User Interface Abstract Model'. In: *Proceedings of the International Multiconference on Computer Science and Information Technology.* Ed. by M. Ganzha, M. Paprzycki and T. Pełech-Pilichowski. 2007, pp. 1021–1024 (cit. on p. 23).

[CB02] Paul Curzon and Ann Blandford. 'From a Formal User Model to Design Rules'. In: *Interactive Systems. Design, Specification, and Verification. Proceedings of the 9th International Workshop, DSV-IS 2002.* Ed. by Bodo Urban, Jean Vanderdonckt and Quentin Limbourg. Vol. 2545. LNCS. Springer, 2002, pp. 1–15 (cit. on p. 13).

[CRB07] Paul Curzon, Rimvydas Rukšėnas and Ann Blandford. 'An approach to formal verification of human-computer interaction'. In: *Form. Asp. Comput.* 19.4 (2007), pp. 513–550 (cit. on p. 13).

[Dea+02] B Dean, M Schachter, C Vincent and N Barber. 'Prescribing errors in hospital inpatients: their incidence and clinical significance'. In: *Quality safety in health care* 11.4 (2002), pp. 340–344 (cit. on p. 26).

[DH97] Andrew M. Dearden and Michael D. Harrison. 'Abstract models for HCI'. In: *Int J Hum-Comput St* 46.1 (1997), pp. 153–178 (cit. on p. 12).

[Deg04] Asaf Degani. *Taming HAL: Designing Interfaces Beyond 2001*. Palgrave Macmillan, 2004 (cit. on p. 18).

[DH02] Asaf Degani and Michael Heymann. 'Formal Verification of Human-Automation Interaction'. In: *Human factors* 44.1 (2002), pp. 28–43 (cit. on p. 18).

[DH03] Asaf Degani and Michael Heymann. 'Analysis and Verification of Human-Automation Interfaces'. In: *Proceedings of the 10th International Conference on Human-Computer Interaction*. 2003, pp. 1–6 (cit. on p. 18).

[Dil96] David L. Dill. 'The Murphi Verification System'. In: *Proceedings of the 8th International Conference on Computer Aided Verification*. CAV '96. London, UK, UK: Springer-Verlag, 1996, pp. 390–393 (cit. on p. 16).

[DF09] Anke Dittmar and Peter Forbrig. 'Task-based design revisited'. In: *Proceedings of the 1st ACM SIGCHI symposium on Engineering interactive computing systems*. EICS '09. ACM, 2009, pp. 111–116 (cit. on p. 13).

[DHF08] Anke Dittmar, Toralf Hübner and Peter Forbrig. 'Interactive Systems. Design, Specification, and Verification'. In: ed. by T. C. Graham and Philippe Palanque. Berlin, Heidelberg: Springer-Verlag, 2008. Chap. HOPS: A Prototypical Specification Tool for Interactive Systems, pp. 58–71 (cit. on p. 13).

[Dix91] AJ Dix. *Formal methods for interactive systems*. Academic Press, 1991 (cit. on pp. 11, 38, 51, 125).

[Dix95] Alan J Dix. 'Formal methods: an introduction to and overview of the use of formal methods within HCI'. In: *Perspectives on HCI: Diverse Approaches* (1995), pp. 9–43 (cit. on p. 125).

[Dix13] Alan J. Dix. 'Formal Methods'. In: *The Encyclopedia of Human-Computer Interaction, 2nd Ed.* Ed. by Mads Soegaard and Rikke Friis Dam. Aarhus, Denmark: The Interaction Design Foundation, 2013 (cit. on p. 125).

[Doh98] Gavin John Doherty. 'A Pragmatic Approach to the Formal Specification of Interactive Systems'. Doctoral dissertation. University of York, 1998 (cit. on p. 12).

[DH93] David J. Duke and Michael D. Harrison. 'Abstract Interaction Objects'. In: *Computer Graphics Forum* 12.3 (1993), pp. 25–36 (cit. on p. 17).

[DAC98] Matthew B. Dwyer, George S. Avrunin and James C. Corbett. 'Property specification patterns for finite-state verification'. In: *Proceedings of the second workshop on Formal methods in software practice.* FMSP '98. Clearwater Beach, Florida, United States: ACM, 1998, pp. 7–15 (cit. on p. 17).

[Dwy+04] Matthew B. Dwyer, Robby Robby, Oksana Tkachuk and Willem Visser. 'Analyzing interaction orderings with model checking'. In: *ASE '04: Proceedings of the 19th IEEE international conference on Automated software engineering.* IEEE Computer Society, 2004, pp. 154–163 (cit. on p. 2).

[DCH97] Matthew Dwyer, Vicki Carr and Laura Hines. 'Model checking graphical user interfaces using abstractions'. In: *ESEC '97/FSE-5: Proceedings of the 6th European conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering* (1997) (cit. on p. 15).

[Ell+02] John Ellson, Emden Gansner, Lefteris Koutsofios, StephenC. North and Gordon Woodhull. 'Graphviz — Open Source Graph Drawing Tools'. In: *Graph Drawing.* Ed. by Petra Mutzel, Michael Jünger and Sebastian Leipert. Vol. 2265. LNCS. Springer Berlin Heidelberg, 2002, pp. 483–484 (cit. on p. 36).

[Gam+95] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. *Design patterns: elements of reusable object-oriented software.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995 (cit. on p. 59).

[GT10] Andy Gimblett and Harold Thimbleby. 'User interface model discovery: towards a generic approach'. In: *Proceedings of the 2nd ACM SIGCHI symposium on Engineering interactive computing systems.* EICS '10. ACM, 2010, pp. 145–154 (cit. on pp. 6, 25, 98, 157).

[GT13] Andy Gimblett and Harold Thimbleby. 'Applying theorem discovery to automatically find and check usability heuristics'. In: *Proceedings of the 5th ACM SIGCHI symposium on Engineering interactive computing systems.* EICS '13. ACM, 2013, pp. 101–106 (cit. on pp. 6, 157).

[GIM99] Aristides Gionis, Piotr Indyk and Rajeev Motwani. 'Similarity Search in High Dimensions via Hashing'. In: *Proceedings of the 25th International Conference on Very Large Data*

Bases. VLDB '99. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999, pp. 518–529 (cit. on p. 136).

[Giv+13] Paul Givens, Aleksandar Chakarov, Sriram Sankaranarayanan and Tom Yeh. 'Exploring the internal state of user interfaces by combining computer vision techniques with grammatical inference'. In: *Proceedings of the 2013 International Conference on Software Engineering*. ICSE '13. San Francisco, CA, USA: IEEE Press, 2013, pp. 1165–1168 (cit. on p. 22).

[GM93] M. J. C. Gordon and T. F. Melham, eds. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993 (cit. on p. 12).

[GTC06] Jeremy Gow, Harold Thimbleby and Paul Cairns. 'Automatic critiques of interface modes'. In: *Interactive Systems. Design, Specification, and Verification. Proceedings of the 12th International Workshop, DSV-IS 2005*. Ed. by Michael D. Harrison and Stephen W. Gilroy. Vol. 3941. LNCS. Newcastle upon Tyne, UK: Springer, 2006, pp. 201–212 (cit. on pp. 100, 101, 117, 157).

[HVV91] G. de Haan, G.C. van der Veer and J.C. van Vliet. 'Formal modelling techniques in human-computer interaction'. In: *Acta Psychologica* 78.1-3 (1991), pp. 27–67 (cit. on p. 11).

[Hal90] Anthony Hall. 'Seven Myths of Formal Methods'. In: *IEEE Softw.* 7.5 (1990), pp. 11–19 (cit. on p. 9).

[Har87] David Harel. 'Statecharts: a visual formalism for complex systems'. In: *Science of Computer Programming* 8.3 (1987), pp. 231–274 (cit. on p. 17).

[HN96] David Harel and Amnon Naamad. 'The STATEMATE semantics of statecharts'. In: *ACM Trans. Softw. Eng. Methodol.* 5.4 (1996), pp. 293–333 (cit. on p. 17).

[Har+08] Michael D. Harrison, José Creissac Campos, Gavin Doherty and Karsten Loer. 'Connecting Rigorous System Analysis to Experience-Centered Design'. In: *Maturing Usability*. Ed. by EffieLai-Chong Law, EbbaThora Hvannberg and Gilbert Cockton. Human-Computer Interaction Series. Springer London, 2008, pp. 56–74 (cit. on p. 17).

[HD02] Michael Heymann and Asaf Degani. *On Abstractions and Simplifications in the Design of Human-Automation Interfaces*. Technical Report. NASA, 2002 (cit. on p. 18).

[HK06] Antawan Holmes and Marc Kellogg. 'Automating Functional Tests Using Selenium'. In: *Proceedings of the conference on AGILE 2006.* AGILE '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 270–275 (cit. on p. 5).

[Hol91] Gerard J. Holzmann. *Design and validation of computer protocols.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1991 (cit. on p. 16).

[Hud+07] Paul Hudak, John Hughes, Simon Peyton Jones and Philip Wadler. 'A history of Haskell: being lazy with class'. In: *Proceedings of the third ACM SIGPLAN conference on History of programming languages.* HOPL III. San Diego, California: ACM, 2007, pp. 12/1–12/55 (cit. on p. 42).

[HC96] Andrew Hussey and David Carrington. 'Using Object-Z to compare the MVC and PAC architectures'. In: *Proceedings of the 1996 BCS-FACS conference on Formal Aspects of the Human Computer Interface.* FAC-FA'96. Sheffield, UK: British Computer Society, 1996, pp. 6–6 (cit. on p. 12).

[ISO89] ISO/IEC. *Information Processing Systems – Open Systems Interconnection: LOTOS, A Formal Description Technique Based on the Temporal Ordering of Observational Behavior.* ISO 8807. 1989 (cit. on p. 13).

[Jac06] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis.* The MIT Press, 2006 (cit. on p. 30).

[Jac83] Robert J. K. Jacob. 'Using formal specifications in the design of a human-computer interface'. In: *Commun. ACM* 26.4 (1983), pp. 259–264 (cit. on p. 10).

[JJ07] Raoul Praful Jetley and Paul L Jones. 'Safety Requirements based Analysis of Infusion Pump Software'. In: *Proceedings of the IEEE Real Time Systems Symposium* (2007), pp. 1–4 (cit. on p. 28).

[JH92] Chris W. Johnson and Michael D. Harrison. 'Using temporal logic to support the specification and prototyping of interactive control systems'. In: *International journal of man-machine studies* (1992) (cit. on p. 15).

[KSH08] Nadjet Kamel, Sid Ahmed Selouani and Habib Hamam. 'A decomposed model-checking approach for the verification of CARE usability properties for multimodal user interfaces'. In: *AVOCS '08: Proceedings of the 8th International Workshop on Automated Verification of Critical Systems* (2008), pp. 99–112 (cit. on pp. 16, 27).

[KNP03] R Kaye, R North and M Peterson. 'UPCARE: An analysis, description, and educational tool for medical device use problems'. In: *Proceedings of the Eighth Annual International Conference of Industrial Engineering Theory, Applications and Practice* (2003) (cit. on p. 27).

[Knu74] Donald E. Knuth. 'Structured Programming with go to Statements'. In: *ACM Comput. Surv.* 6.4 (December 1974), pp. 261–301 (cit. on p. 135).

[LW08] Peng Li and Eric Wohlstadter. 'View-based maintenance of graphical user interfaces'. In: *AOSD '08: Proceedings of the 7th international conference on Aspect-oriented software development.* Brussels, Belgium: ACM, 2008, pp. 156–167 (cit. on p. 23).

[Lim+05] Quentin Limbourg, Jean Vanderdonckt, Benjamin Michotte, Laurent Bouillon and Víctor López-Jaquero. 'USIXML: A Language Supporting Multi-path Development of User Interfaces'. In: *Engineering Human Computer Interaction and Interactive Systems.* Ed. by Rémi Bastide, Philippe Palanque and Jörg Roth. Vol. 3425. LNCS. Springer Berlin Heidelberg, 2005, pp. 200–220 (cit. on pp. 12, 19).

[Lin+98] Laura Lin, Racquel Isla, Karine Doniz, Heather Harkness, KimJ. Vicente and D.John Doyle. 'Applying Human Factors to the Design of Medical Equipment: Patient-Controlled Analgesia'. In: *Journal of Clinical Monitoring and Computing* 14.4 (1998), pp. 253–263 (cit. on p. 26).

[LVD01] Laura Lin, Kim J Vicente and D.John Doyle. 'Patient Safety, Potential Adverse Drug Events, and Medical Device Design: A Human Factors Engineering Approach'. In: *Journal of Biomedical Informatics* 34.4 (2001), pp. 274–284 (cit. on p. 26).

[Loe03] Karsten Loer. 'Model-based Automated Analysis for Dependable Interactive Systems'. Doctoral dissertation. University of York, 2003 (cit. on p. 17).

[LH01] Karsten Loer and Michael D. Harrison. 'Formal interactive systems analysis and usability inspection methods: two incompatible worlds?' In: *Interactive Systems. Design, Specification, and Verification. Proceedings of the 7th International Workshop, DSV-IS 2000.* Ed. by Philippe Palanque and Fabio Paternò. Vol. 1946. LNCS. Springer, 2001, pp. 169–190 (cit. on p. 18).

[LH04] Karsten Loer and Michael D. Harrison. 'Integrating model checking with the industrial design of interactive systems'. In: *26th International Conference on Software Engineering - W1L Workshop* Bridging the Gaps II: Bridging the Gaps Between Software Engineering and Human-Computer Interaction. 2004, 9–16(7) (cit. on p. 17).

[LH06] Karsten Loer and Michael D. Harrison. 'An integrated framework for the analysis of dependable interactive systems (IFADIS): Its tool support and evaluation'. In: *Automated Software Engg.* 13.4 (2006), pp. 469–496 (cit. on p. 17).

[LC99] Gerald Lüttgen and Victor Carreño. 'Analyzing Mode Confusion via Model Checking'. In: *Theoretical and Practical Aspects of SPIN Model Checking.* Ed. by Dennis Dams, Rob Gerth, Stefan Leue and Mieke Massink. Vol. 1680. LNCS. Springer Berlin Heidelberg, 1999, pp. 120–135 (cit. on p. 16).

[Mar+02] Gary Marsden, Harold Thimbleby, Matt Jones and Paul Gillary. 'Data Structures in the Design of Interfaces'. In: *Personal Ubiquitous Comput.* 6.2 (2002), pp. 132–140 (cit. on p. 24).

[Mas+11] Paolo Masci, Rimvydas Rukšėnas, Patrick Oladimeji, Abigail Cauchi, Andy Gimblett, Yunqiu Li, Paul Curzon and Harold Thimbleby. 'On formalising interactive number entry on infusion pumps.' In: *ECEASST* 45 (2011) (cit. on p. 14).

[Mas+13] Paolo Masci, Rimvydas Rukšėnas, Patrick Oladimeji, Abigail Cauchi, Andy Gimblett, Yunqiu Li, Paul Curzon and Harold Thimbleby. 'The benefits of formalising design guidelines: a case study on the predictability of drug infusion pumps'. In: *Innovations in Systems and Software Engineering* (2013), pp. 1–21 (cit. on p. 14).

[MB81] Richard E. Mayer and Piraye Bayman. 'Psychology of calculator languages: a framework for describing differences in users' knowledge'. In: *Commun. ACM* 24 (8 1981), pp. 511–520 (cit. on pp. 151, 153).

[McM92] Kenneth Lauchlin McMillan. 'Symbolic model checking: an approach to the state explosion problem'. UMI Order No. GAX92-24209. Doctoral dissertation. Pittsburgh, PA, USA, 1992 (cit. on p. 16).

[MBN03] Atif Memon, Ishan Banerjee and Adithya Nagarajan. 'GUI ripping: reverse engineering of graphical user interfaces for testing'. In: *WCRE '03: Proceedings of the 10th Working Conference on Reverse Engineering*. IEEE Computer Society, 2003, p. 260 (cit. on pp. 21, 37).

[MS05] M Mernik and AM Sloane. 'When and how to develop domain-specific languages'. In: *ACM Computing Surveys (CSUR)* 37.4 (2005), pp. 316–344 (cit. on p. 54).

[MBD08] Ali Mesbah, Engin Bozdag and Arie van Deursen. 'Crawling AJAX by inferring user interface state changes'. In: *ICWE '08: Proceedings of the 2008 Eighth International Conference on Web Engineering*. IEEE Computer Society, 2008, pp. 122–134 (cit. on p. 21).

[Mil78] Robin Milner. 'A theory of type polymorphism in programming'. In: *Journal of Computer and System Sciences* 17 (1978), pp. 348–375 (cit. on p. 2).

[Nav+01] David Navarre, Philippe Palanque, Rémi Bastide and Ousmane Sy. 'Structuring Interactive Systems Specifications for Executability and Prototypability'. In: *Interactive Systems. Design, Specification, and Verification. Proceedings of the 7th International Workshop, DSV-IS 2000.* Ed. by Philippe Palanque and Fabio Paternò. Vol. 1946. LNCS. Springer, 2001, pp. 97–119 (cit. on p. 19).

[Nav+09] David Navarre, Philippe Palanque, Jean-Francois Ladry and Eric Barboni. 'ICOs: A model-based user interface description technique dedicated to interactive systems addressing usability, reliability and scalability'. In: *ACM Trans. Comput.-Hum. Interact.* 16.4 (2009), 18:1–18:56 (cit. on p. 19).

[Nie93] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann Publishers Inc., 1993 (cit. on pp. 18, 27, 153).

[Ola12] Patrick Oladimeji. 'Towards safer number entry in interactive medical systems'. In: *Proceedings of the 4th ACM SIGCHI symposium on Engineering interactive computing systems*. EICS '12. ACM, 2012, pp. 329–332 (cit. on pp. 25, 67).

[OTC11] Patrick Oladimeji, Harold Thimbleby and Anna Cox. 'Number Entry Interfaces and their Effects on Errors and Number Perception'. In: *Proceedings IFIP Conference on Human-Computer Interaction — Interact 2011*. Vol. IV. Lisbon, Portugal: Springer-Verlag, 2011, pp. 178–185 (cit. on p. 25).

[ORS92] Sam Owre, John M. Rushby and Natarajan Shankar. 'PVS: A Prototype Verification System'. In: *Proceedings of the 11th International Conference on Automated Deduction: Automated Deduction*. CADE-11. London, UK, UK: Springer-Verlag, 1992, pp. 748–752 (cit. on p. 12).

[PFM08] Ana C. R. Paiva, João C. P. Faria and Pedro M. C. Mendes. 'Reverse engineered formal models for GUI testing'. In: *FMICS '07: International Workshop on Formal Methods for Industrial Critical Systems*. Vol. 4916. LNCS. Springer, 2008, pp. 218–233 (cit. on pp. 21, 37).

[Pai+05] Ana C. R. Paiva, João C. P. Faria, Nikolai Tillmann and Raul A. M. Vidal. 'A model-to-implementation mapping tool for automated model-based GUI testing'. In: *ICFEM '05: 7th International Conference on Formal Engineering Methods*. Vol. 3785. LNCS. Springer, 2005, pp. 450–464 (cit. on p. 21).

[Pal+11] Philippe Palanque, Eric Barboni, Célia Martinie, David Navarre and Marco Winckler. 'A model-based approach for supporting engineering usability evaluation of interaction techniques'. In: *Proceedings of the 3rd ACM SIGCHI symposium on Engineering interactive computing systems*. EICS '11. ACM, 2011, pp. 21–30 (cit. on p. 20).

[Par69] David L. Parnas. 'On the use of transition diagrams in the design of a user interface for an interactive computer system'. In: *Proceedings of the 1969 24th national conference of the ACM*. ACM '69. New York, NY, USA: ACM, 1969, pp. 379–385 (cit. on p. 10).

[Par10] David Lorge Parnas. 'Really Rethinking 'Formal Methods''. In: *Computer* 43.1 (2010), pp. 28–34 (cit. on p. 1).

[Pat99] Fabio Paternò. *Model-Based Design and Evaluation of Interactive Applications.* 1st. London, UK, UK: Springer-Verlag, 1999 (cit. on p. 13).

[PS01] Fabio Paternò and Carmen Santoro. 'Integrating model checking and HCI tools to help designers verify user interface properties'. In: *Interactive Systems. Design, Specification, and Verification. Proceedings of the 7th International Workshop, DSV-IS 2000.* Ed. by Philippe Palanque and Fabio Paternò. Vol. 1946. LNCS. Springer, 2001, pp. 135–150 (cit. on p. 16).

[Pet62] Carl Adam Petri. 'Kommunikation mit Automaten'. Doctoral dissertation. Universität Hamburg, 1962 (cit. on p. 19).

[Pey03] Simon Peyton Jones, ed. *Haskell 98 Language and Libraries – The Revised Report.* Cambridge University Press, 2003 (cit. on pp. 42, 57).

[Pnu77] Amir Pnueli. 'The temporal logic of programs'. In: *Proceedings of the 18th Annual Symposium on Foundations of Computer Science.* SFCS '77. Washington, DC, USA: IEEE Computer Society, 1977, pp. 46–57 (cit. on p. 15).

[RJJ09] Arnab Ray, Raoul Jetley and Paul Jones. 'Engineering high confidence medical device software'. In: *ACM SIGBED Review* 6.2 (2009), pp. 1–7 (cit. on p. 27).

[Rea00] James Reason. 'Human error: models and management'. In: *BMJ* 320.7237 (18th2000), pp. 768–770 (cit. on pp. 3, 26).

[Ree79] Trygve M. H. Reenskaug. *Models - Views - Controllers.* Technical Note. Xerox PARC, 1979 (cit. on pp. 12, 31).

[Ree05] Greg A. Reeve. 'A Refinement Theory for μ-Charts'. Doctoral dissertation. University of Waikato, 2005 (cit. on p. 19).

[RW06] A. W. Roscoe and Zhenzhong Wu. 'Verifying Statemate Statecharts Using CSP and FDR'. In: *Proceedings of ICFEM 2006.* 2006 (cit. on p. 18).

[Ros98] A.W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall, 1998 (cit. on p. 15).

[RJB99] James Rumbaugh, Ivar Jacobson and Grady Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley Professional, 2nd1999 (cit. on p. 2).

[Rus02] John Rushby. 'Using model checking to help discover mode confusions and other automation surprises'. In: *Reliability Engineering and System Safety* 75.2 (2002), pp. 167–177 (cit. on p. 16).

[Rus07] John Rushby. 'Automated Formal Methods Enter the Mainstream'. In: *Journal of Universal Computer Science* 13.5 (2007), pp. 650–660 (cit. on pp. 1, 9, 14).

[Sca98] Bryan Scattergood. 'The Semantics and Implementation of Machine-Readable CSP'. DPhil thesis. The University of Oxford, 1998 (cit. on p. 18).

[Sch96] David A. Schmidt. 'On the need for a popular formal semantics'. In: *ACM Comput. Surv.* 28.4es (1996) (cit. on p. 1).

[Shn82] B. Shneiderman. 'Multiparty Grammars and Related Features for Defining Interactive Systems'. In: *IEEE Transactions on Systems, Man and Cybernetics* 12.2 (1982), pp. 148–154 (cit. on p. 10).

[SP04] Ben Shneiderman and Catherine Plaisant. *Designing the User Interface: Strategies for Effective Human-Computer Interaction (4th Edition)*. Pearson Addison Wesley, 2004 (cit. on p. 27).

[Sil10] João Carlos Silva. 'GUISURFER: A Generic Framework for Reverse Engineering of Graphical User Interfaces'. Doctoral dissertation. Universidade do Minho, 2010 (cit. on p. 23).

[SCS06] João Carlos Silva, José Creissac Campos and João Saraiva. 'Models for the Reverse Engineering of Java/Swing Applications'. In: *3rd International Workshop on Metamodels, Schemas, Grammars, and Ontologies (ateM 2006) for Reverse Engineering*. Ed. by J. M. Favre, D. Gasevic, R. Lämmel and A. Winter. Informatik-Bericht series 1/2006. Johannes Gutenberg-Universität Mainz, Institut für Informatik – FB 8, 2006 (cit. on p. 22).

[SCS07] João Carlos Silva, José Creissac Campos and João Saraiva. 'Combining Formal Methods and Functional Strategies Regarding the Reverse Engineering of Interactive Applications'. In: *Interactive Systems: Design, Specification and Verification*. Vol. 4323. LNCS. Springer-Verlag, 2007, pp. 137–150 (cit. on p. 22).

[SCS10] João Carlos Silva, José Creissac Campos and João Saraiva. 'GUI Inspection from Source Code Analysis'. In: *Electronic Communications of the EASST* (2010). to appear (cit. on p. 23).

[SSC09] João Carlos Silva, João Saraiva and José Creissac Campos. 'A generic library for GUI reasoning and testing'. In: *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*. Honolulu, Hawaii: ACM, 2009, pp. 121–128 (cit. on p. 22).

[Sil+10] João Carlos Silva, Carlos Silva, Rui D. Gonçalo, João Saraiva and José Creissac Campos. 'The GUISurfer tool: towards a language independent approach to reverse engineering GUI code'. In: *Proceedings of the 2nd ACM SIGCHI symposium on Engineering interactive computing systems*. EICS '10. ACM, 2010, pp. 181–186 (cit. on p. 23).

[Sir10] Alexander Sirotkin. 'Web application testing with selenium'. In: *Linux J.* 2010.192 (2010) (cit. on p. 5).

[SHC05] Julian Smart, Kevin Hock and Stefan Csomor. *Cross-Platform GUI Programming with wxWidgets*. Prentice Hall, 2005 (cit. on p. 57).

[Sta07a] Stefan Staiger. 'Reverse Engineering of Graphical User Interfaces Using Static Analyses'. In: *WCRE '07: Proceedings of the 14th Working Conference on Reverse Engineering*. IEEE Computer Society, 2007, pp. 189–198 (cit. on p. 23).

[Sta07b] Stefan Staiger. 'Static Analysis of Programs with Graphical User Interface'. In: *CSMR '07: Proceedings of the 11th European Conference on Software Maintenance and Reengineering*. IEEE Computer Society, 2007, pp. 252–264 (cit. on p. 23).

[SB82] William Swartout and Robert Balzer. 'On the inevitable intertwining of specification and implementation'. In: *Commun. ACM* 25 (7 1982), pp. 438–440 (cit. on pp. 3, 10).

[Tau90] Michael J. Tauber. 'ETAG: Extended task action grammar. A language for the description of the user's task language'. In: *Proceedings of the IFIP TC13 3rd Interational Conference on Human-Computer Interaction*. INTERACT '90. Amsterdam, The Netherlands, The Netherlands: North-Holland Publishing Co., 1990, pp. 163–168 (cit. on p. 11).

[Thi97] Harold Thimbleby. 'A True Calculator'. In: *Engineering Science and Education Journal* 6.3 (1997), pp. 128–136 (cit. on p. 23).

[Thi00] Harold Thimbleby. 'Calculators are Needlessly Bad'. In: *International Journal of Human-Computer Studies* 52.6 (2000), pp. 1031–1069 (cit. on p. 24).

[Thi04a] Harold Thimbleby. 'User Interface Design with Matrix Algebra'. In: *ACM Transactions on Computer-Human Interaction* 11.2 (2004), pp. 181–236 (cit. on p. 103).

[Thi07a] Harold Thimbleby. *Press On. Principles of interaction programming.* MIT Press, Boston, USA., 2007 (cit. on p. 30).

[Thi07b] Harold Thimbleby. 'User-centered Methods are Insufficient for Safety Critical Systems'. In: *USAB'07 — Usability & HCI for Medicine and Health Care.* Ed. by Andreas Holzinger. Vol. 4799. LNCS. Graz, Austria: Springer Verlag, 2007, pp. 1–20 (cit. on pp. 3, 24).

[Thi09] Harold Thimbleby. 'Contributing to safety and due diligence in safety-critical interactive systems development by generating and analyzing finite state models'. In: *Proceedings of the 1st ACM SIGCHI symposium on Engineering interactive computing systems.* EICS '09. ACM, 2009, pp. 221–230 (cit. on pp. 24, 30, 39, 41, 42).

[Thi10] Harold Thimbleby. 'Avoiding latent design conditions using UI discovery tools'. In: *International Journal of Human-Computer Interaction* 26.2 (2010), pp. 1–12 (cit. on p. 25).

[Thi12] Harold Thimbleby. 'Heedless Programming: Ignoring Detectable Error is a Widespread Hazard'. In: *Software — Practice & Experience* 42.11 (2012), pp. 1393–1407 (cit. on pp. 26, 154).

[TC10] Harold Thimbleby and Paul Cairns. 'Reducing Number Entry Errors: Solving a Widespread, Serious Problem'. In: *Journal Royal Society Interface* 7.51 (2010), pp. 1429–1439 (cit. on pp. 3, 25, 153).

[TCJ01] Harold Thimbleby, Paul Cairns and Matt Jones. 'Usability Analysis with Markov Models'. In: *ACM Transactions on Computer-Human Interaction* 8.2 (2001), pp. 99–132 (cit. on p. 108).

[Thi+12] Harold Thimbleby, Abigail Cauchi, Andy Gimblett, Paul Curzon and Paolo Masci. 'Safer "5-key" Number Entry User Interfaces using Differential Formal Analysis'. In: *Proceedings BCS Conference on HCI.* Vol. XXVI. Birmingham, UK, 2012, pp. 29–38 (cit. on pp. 25, 68, 69).

[TG08] Harold Thimbleby and Jeremy Gow. 'Engineering Interactive Systems'. In: ed. by Jan Gulliksen, Morton Borup Harning, Philippe Palanque, Gerrit C. Veer and Janet Wesson. Berlin, Heidelberg: Springer-Verlag, 2008. Chap. Applying Graph Theory to Interaction Design, pp. 501–519 (cit. on pp. 17, 23, 24).

[TO09] Harold Thimbleby and Patrick Oladimeji. 'Social network analysis and interactive device design analysis'. In: *Proceedings of the 1st ACM SIGCHI symposium on Engineering interactive computing systems*. EICS '09. ACM, 2009, pp. 91–100 (cit. on pp. 25, 30, 42, 88, 92–94, 152).

[Thi04b] William Thimbleby. 'A novel pen-based calculator and its evaluation'. In: *Proceedings of the 3rd Nordic conference on Human-computer interaction*. NordiCHI '04. Tampere, Finland: ACM, 2004, pp. 445–448 (cit. on p. 24).

[Tip95] F. Tip. 'A Survey of Program Slicing Techniques'. In: *Journal of Programming Languages* 3 (1995), pp. 121–189 (cit. on p. 22).

[UIM92] UIMS Workshop. 'A metamodel for the runtime architecture of an interactive system: the UIMS tool developers workshop'. In: *SIGCHI Bull.* 24.1 (1992), pp. 32–37 (cit. on pp. 9, 12).

[Vic+03] Kim J. Vicente, Karima Kada-Bekhaled, Gillian Hillel, Andrea Cassano and Beverley A. Orser. 'Programming errors contribute to death from patient-controlled analgesia: case report and estimate of probability'. In: *Canadian Journal Anethesia* 50.4 (2003), pp. 328–332 (cit. on p. 26).

[Was85] Anthony I. Wasserman. 'Extending State Transition Diagrams for the Specification of Human-Computer Interaction'. In: *IEEE Trans. Softw. Eng.* 11.8 (1985), pp. 699–713 (cit. on p. 10).

[Win92] Patrick Henty Winston. *Artificial Intelligence*. Addison-Wesley, 1992 (cit. on p. 30).

[WCB13] Sarah Wiseman, Anna L. Cox and Duncan P. Brumby. 'Designing Devices With the Task in Mind: Which Numbers Are Really Used in Hospitals?' In: *Human Factors: The Journal of the Human Factors and Ergonomics Society* 55.1 (2013), pp. 61–74 (cit. on p. 28).

[Woo+09] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui and John Fitzgerald. 'Formal methods: Practice and experience'. In: *ACM Comput. Surv.* 41.4 (2009), 19:1–19:36 (cit. on pp. 1, 8).

[Zha+03] Jiajie Zhang, Todd R Johnson, Vimla L Patel, Danielle L Paige and Tate Kubose. 'Using usability heuristics to evaluate patient safety of medical devices'. In: *Journal of Biomedical Informatics* 36.1-2 (2003), pp. 23–30 (cit. on p. 27).