Supporting Interactive System Testing with Interaction Sequences

A thesis

submitted in fulfilment

of the requirements for the degree

of

Doctor of Philosophy in Computer Science

at

The University of Waikato

by

JESSICA DAWN TURNER

THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

2019

Abstract

Despite extensive research into the modelling and testing of interactive systems, existing strategies do not adequately cover all parts of an interactive system. These existing strategies model and test either the functional or interactive components of an interactive system separately, however, issues may arise where these components intersect. Therefore, further investigation into the modelling and testing of this intersection is required.

Interaction sequences are a series of steps a user can take to complete a specific task or to arbitrarily explore an interactive system. In this research interaction sequences are used as an abstraction of the interactive system to inform a model-based testing approach using lightweight formal methods. Interaction sequences provide an abstract view of the point at which the functional and interactive components intersect, and as a result also provide a good starting point for investigation into the modelling and testing of this area. Interaction sequences are applicable to all types of interactive systems irrespective of the type of interaction, therefore modelling and testing approaches using this abstraction are also applicable to all types of interactive systems.

In this thesis the findings of our investigation into modelling and testing using interaction sequences are presented. We describe formalisation of interaction sequences and modelling of these sequences using Finite State Automata (FSA). We introduce the *self-containment property* and show how this is used to control the size and state space of FSA. We demonstrate simulating interaction sequences and discuss how these models can be applied within both model checking and testing techniques. Lastly, we present a new approach for generating tests from interaction sequences and their associated models.

Acknowledgements

I would like to express my sincere gratitude to my supervisors Professor Steve Reeves and Doctor Judy Bowen for their continued support, patience, and enthusiasm. Your guidance throughout this time of research and writing of this thesis has been invaluable. I am exceedingly grateful for your care, understanding and mentorship during my degree while I managed difficult personal circumstances.

In addition to my supervisors, I would like to thank my friends and colleagues at the University of Waikato. I have appreciated your friendship and learning from your expertise.

I am also grateful to the University of Waikato, the Waikato Graduate Women Educational Trust, the University of Waikato Computer Science department, the New Horizons for Women Trust in partnership with the Association for Women in the Sciences for providing essential financial support, without which I would not have been able to undertake or continue doctoral study.

A thank you to Trevor and Dawn Harrison, Glenda Harrison and Mark Hollands, and Owen and Leanne Lucas who have supported me throughout this degree. A special thank you to Chris McNeil whose lifelong mentorship and friendship has been integral to my academic success.

Lastly, I would like to take this opportunity to thank my family for enduring the high and low points of this research with me. To Dad, thank you for always being ready with a “solution” for problems you did not always fully comprehend. To Callan, thank you for always being so understanding, patient and kind. To Mum, thank you for late night calls, afternoon drives, and for your perseverance, despite the circumstances we did it.

Soli Deo Gloria.

Publications

Some of the research presented in this thesis is published in the following:

- J. Turner, J. Bowen, and S. Reeves. Supporting Interactive System Testing with Interaction Sequences. In *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '17, pages 129–132, New York, NY, USA, 2017. ACM.

- J. Turner, J. Bowen, and S. Reeves. Simulating Interaction Sequences. In *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '18, pages 8:1–8:7, New York, NY, USA, 2018. ACM. Discussed in chapters 2 and 6.

- J. Turner, J. Bowen, and S. Reeves. Interactive System Testing using Interaction Sequences. In *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '18, pages 16:1–16:5, New York, NY, USA, 2018. ACM.

- J. Turner, J. Bowen, and S. Reeves. Using Abstraction with Interaction Sequences for Interactive System Modelling. To appear in *Software Technologies: Applications and Foundations 2018 - Workshop*, STAFW '18. Discussed in chapters 4 and 5.

As primary author for these publications the technical contributions come from my research, the other authors on these papers are my supervisors.

Contents

List of Figures

List of Tables

Glossary

Alaris GP Pump	Alaris General Purpose Volumetric Infusion Pump.
CIS	Complete Interaction Sequences.
CPModel	Component Presentation Model.
CTT	Concur Task Trees.
CTTE	Concur Task Tree Environment.
DFA	Deterministic Finite Automata.
EFG	Event Flow Graph.
EFSM	Extended Finite State Machine.
FDA	United States Food and Drug Administration.
FSA	Finite State Automata.
FSM	Finite State Machine.
GIP	The Generic Infusion Pump.
GUI	Graphical User Interface.
GUITAR	GUI Testing frAmewoRk.

HAMSTERS	Human-centered Assessment and Modelling to Support Task Engineering for Resilient Systems.
ISTQB	International Software Testing Qualifications Board.
MVC	Model View Controller.
NFA	Non-deterministic Finite Automata.
PCA	Patient-controlled Analgesia.
PIM	Presentation Interaction Model.
PModel	Presentation Model.
PMR	Presentation Model Relation.
SF	Start Final.
SUT	System Under Test.
UI	User Interface.
VDM	Vienna Development Method.
VTBI	Volume to be Infused.
WIMP	Windows, Icons, Menus and Pointers.

Chapter 1

Introduction

Interactive systems are used to support humans in their everyday tasks, often to make us more productive, to increase safety and to reduce time costs [48]. Interactive systems come in several different forms and the nature of interaction continues to evolve (speech, touch, etc.). In this research, an interactive system is defined as software or a hardware device with a user interface that requires human input.

Conceptually, interactive systems consist of two main components, the interactive and the functional. The interactive component is the user interface of the system, this acts as a "gateway" to the functional component. The user interface is made up of interactive elements, for example a visual button on a screen or a voice input mechanism, which we refer to as widgets. Each widget has associated behaviours which manipulate the interactive and functional components of the system. The functional component performs appropriate calculations and instructions based on the information it receives and gives to the interactive component.

Interactive systems are used in safety-critical domains to assist in complex domain-specific tasks. For example, infusion pumps are used in hospitals to dispense medicine to patients. These types of pumps are commonly used in place of gravity drips in order to provide better safety to patients. These types of pumps solved common faults associated with gravity drips, such as monitoring of medication flow to identify air bubbles, and administering medication in more accurate volumes, therefore increasing patient safety. Systems which are used in a safety-critical context are referred to as safety-critical interactive systems.

A safety-critical interactive system is an interactive system in which error could lead to the injury or death of end user(s), where an end user is a human or humans interacting with the user interface. For example, an end user of an infusion pump could be the patient receiving an infusion or a nurse inputting an infusion to be delivered. If the nurse inputs an infusion incorrectly he or she could endanger the patient connected to the pump. Safety-critical interactive systems were created for a number of reasons, such as to improve task safety, reduce the time costs associated with a task or to solve some issue that could not be solved with a manual process. For example, an infusion pump system may allow a human end user (in this instance a nurse) to make calculations related to an infusion more reliably than "by hand". As a result of the safety-critical nature of these systems it is necessary to ensure they work as intended.

Interactive system testing is the process of following some testing strategy to demonstrate systems will behave as expected. We use interactive system testing to identify errors, either before the system is in use or regularly as the system is updated. This allows us to fix errors to ensure that the system continues to behave in an expected way.

As stated by Dijkstra, "Program testing can be used to show the presence of bugs, but never their absence" [21]. This means that despite extensive research into the testing of safety-critical interactive systems and interactive systems in general, no testing strategy can show that a system will be "perfect".

Perfect meaning a system which is guaranteed to be completely free of error. Therefore, we extensively test interactive systems in order to remove as many errors as possible, to ensure that systems are as reliable and safe as possible.

This is particularly important in interactive systems as the addition of a human user can introduce new errors (either accidentally or intentionally), even when we are satisfied all identified errors have been removed from the system. This is because we cannot be sure that the user will interact with the system as intended. Therefore, in addition to removing as many errors as possible, we must also test for and remove interactions that a human could perform which have the potential for error.

As human users can be unpredictable, instead of focussing on the user and their interactions we instead focus on the possible interactions available in the interactive system. This is so we can explore the many possibilities of interaction from the perspective of the system. By doing this we can capture unpredictable human behaviours without having to formalise or model the human end user.

In interactive system testing, the interactive and functional components of a system are often tested separately. This is evidenced by the many modelling and testing strategies which focus on one of these two components (we present some of these strategies in chapter 2). That is, there are individual strategies tailored to test the functionality and interactivity individually, resulting in two different sets of tests. We refer to these sets as a testing suite, which consists of a group of tests for a particular system. This approach is useful in development as different skill sets are required to build the functional and interactive components, and as a result these are often designed and created by different teams of people. Testing each of these components separately allows us to investigate that each part of the system will behave in a way that is

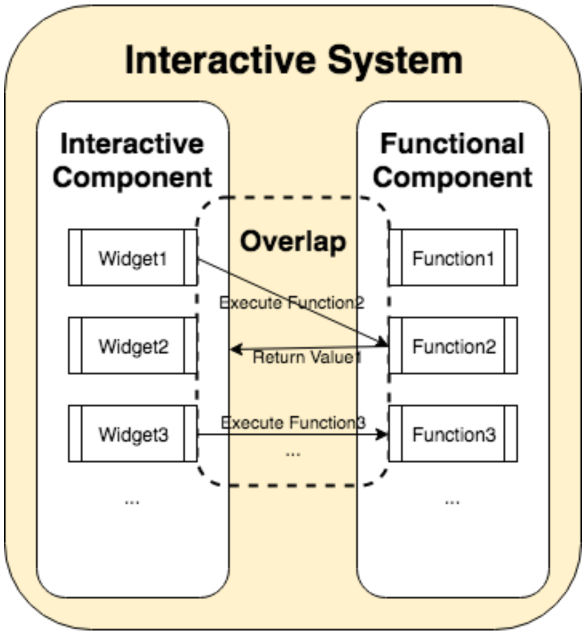


Figure 1.1: Abstract view of the Components of an Interactive System

expected.

Furthermore, when testing the different components, the types of errors we search for can be different. For example, in the functional component we can test for logical correctness, code coverage, back-end connectivity and so on. In the interactive component we may consider task satisfaction, usability, learnability in addition to code aspects. Therefore, it is essential to test each area of an interactive system to find errors and demonstrate the system works as expected.

However, while each component of an interactive system can be tested individually, errors can still arise when each of these components intersect, we refer to this intersection as the overlap component. That is, when the interactive component sends instructions to the functional component, these instructions may not be sent correctly, and vice versa (see figure 1.1). Therefore, to remove as many errors as possible, not only do we need ways to test the interactive and functional components of an interactive system, but to test this overlap component. Furthermore, we must make this intersection resilient to arbitrary

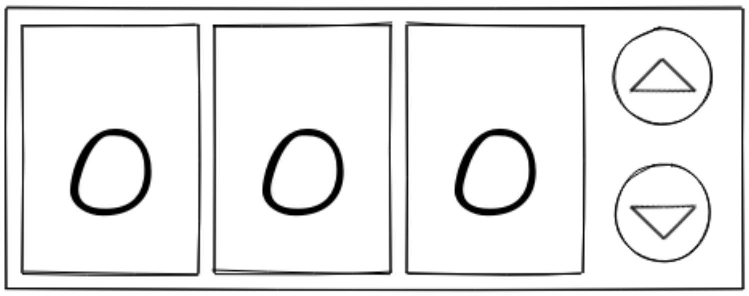


Figure 1.2: 3 Digit Display Interactive System

interactions from the user, as due to their unpredictably they may cause errors to occur in the system by interacting in ways which are not expected. Adding resiliency to the system in this way will increase system reliability and safety.

Consider the Model View Controller (MVC) framework which consists similarly of three different components (see [36, 60]). In the abstract view described above the model component is the functional component of the system, the view component the interactive component, and the controller component the overlap component. It is perhaps obvious that if the controller component of this framework does not behave as expected neither the model or the view can behave as expected, as the controller ensures that these components work together correctly. Similarly for the overlap component, if instructions are lost or communicated incorrectly between the functional and the interactive components, the consequences could have a serious effect on the end users, particularly in safety-critical contexts.

For example, consider the three digit display interactive system shown in figure 1.2. In this system a user can increase or decrease the value of the display by interacting with either the up or down buttons respectively. We can test separately to show that the interactive and functional components work as expected, however, we must also demonstrate that the relationship between the display and the stored value remains intact (part of the overlap component). For instance, if the display shows 999 and the user selects the increase option there are several possibilities: the display may remain at 999, the display may roll-over to 000, or an error may be displayed and so on. In contrast, in the functional component the increase function may allow a value of 1000 to be stored, resulting in a mismatch between the display and value being stored internally. This error may seem trivial, however safety-critical medical devices often have similarly limited segmented numeric displays, and errors with such systems may have serious, life-threatening consequences (see [72] for examples).

While this is a simple example, the context in which this system is used could lead to unexpected consequences. For example, if the three digit display was used to input an infusion rate, the user could end up inputting a dose of a significantly higher amount than what they believe was specified (due to the mismatch between the internal value and the value displayed), this could result in serious injury or even death to the patient connected to the machine (such errors are commonly seen, see [26, 66, 15, 59, 18, 50] for some examples). While we cannot control what context the system is used in, we can test the overlap component to determine that it works as expected.

Design artefacts are created to define the different aspects of an interactive system. For example, a model may be used to specify the way in which a system transitions between different modes or windows. Formal specifications are one type of design artefact which are used to specify the behaviour of the interactive system. For example, the expected behaviour of a function such as addition, subtraction and so on. Throughout the interactive system development process we ensure that the system adheres to the design artefacts, to ensure we have created the intended system.

In general, formal methods are a collection of techniques which allow us to reason about systems, using formal logic and mathematics. They are used to inform modelling and testing processes of systems to confirm desirable properties, such as safety or reliability. We can use formal methods to determine that a system works as expected.

In this research we take a "light-weight" approach to formal methods, in that we specify parts of an interactive system instead of the system in its entirety (see [81]). This is in order to reduce the complexity and scope of the models we create (models being one form of a system specification) and to increase likelihood of industry adoption. For example, formal methods are used in the Generic Infusion Pump (GIP) project[1] in order to improve safety, security and usability of medical systems. In this work we will use specifications of the functional and interactive components of an interactive system (which we assume will exist as part of a robust software engineering approach). In addition to this, we will specify the overlap component using interaction sequences. The specifications we use will be in various forms, including formal models.

Model-based testing is a testing strategy which incorporates models of the system under test (SUT) to inform the testing approach. These models are used as the basis for generating tests. Models are created from abstractions of the SUT and often have a defined focus. For example, task models are used to model the tasks of an interactive system, categorised by differing levels of user interaction. Abstraction is a useful technique used in the creation of models, as it allows us to focus on specific parts of the SUT while providing a way to hide unnecessary details.

Interaction sequences are the steps that a user can take to complete a predefined task or to arbitrarily explore an interactive system. They encapsulate both the interactive and functional component behaviours of the system and as

[1]See `https://rtg.cis.upenn.edu/gip/`

a result provides us with a view of the overlap component. Therefore, they are a useful abstraction for informing a new testing approach for this component.

In this research, we define a technique for using interaction sequences as an abstraction of interactive systems to support interactive system testing. This abstraction will be used to generate models of the SUT, which can be used to inform the testing process. This testing strategy can be used in addition to functional and interactive component testing as another way of potentially identifying errors for removal, in order to improve system safety and reliability.

1.1 Problem Statement

Testing interactive systems is a necessary part of the development process. Myers *et al.* state, "Whereas low-level impacts of bugs may only inconvenience the end user, the worst impacts can result in large financial losses, or even cause harm to people" [48]. In interactive system testing, the separate components of the system, the interactive and functional, are commonly tested individually ignoring the overlap component. However, errors can arise in the overlap component. Furthermore, we cannot guarantee that end users will interact with systems as expected, or that the context in which the system is used is appropriate, both of which can lead to new errors. Therefore, we must investigate ways in which to improve the interactive system testing process with a specific focus on the overlap component, to identify as many errors as possible for removal in order to increase system reliability and safety, and to help make systems more resilient to differing interactions and environments.

1.2 Research Questions

To address the problem statement we propose the use of interaction sequences.

This leads to the following research questions:

1. How can we generate and simulate interaction sequences automatically to ensure reproducibility?

2. Can the state space of interaction sequences be controlled while preserving the properties of the interaction sequence, so that we do not lose information?

3. How can we use interaction sequences as an abstraction so that they may be used to inform a testing suite to enhance interactive system testing?

One of the main issues with using testing strategies is the considerable time cost of the various existing testing approaches. As a result, many approaches include ways to add automation and structure to reduce the human effort required to follow the approach and to ensure reproducibility (we will discuss this in more detail later). Therefore, we must find ways to generate and simulate interaction sequences automatically to ensure reproducibility, so that we may reduce the human effort required to create sequences.

In modelling, the state space explosion problem occurs when a model has a state space which is too large to remain tractable. The state space of a model consists of all possible combinations of the states and transitions within the model. Conceptually, interaction sequences have the possibility to be never ending, with never ending combinations of these never ending sequences, this leads to the state space explosion problem. Therefore to address research question two, we must find ways in which to constrain interaction sequence length and consequently the state space, to ensure that models remain tractable.

In addition, existing techniques which reduce the state space are not sufficient, as important information is lost when the technique is applied (we will discuss this in more detail later). Therefore, we must find ways to control the state space by hiding information, so that it is retrievable when required.

Interaction sequences as an abstraction of interactive systems provide us with a view of the intersection between the interactive and functional component, however it is not clear how we will utilise this abstraction to inform the testing process. Therefore to address research question three, we must investigate ways in which we can use this abstraction for modelling and testing purposes.

1.3 Contributions

In answering the research questions, this thesis makes the following contributions:

1. A technique for formalising and generating interaction sequences. This contribution addresses research question one. We will demonstrate how to formalise interaction sequences to ensure reproducibility based on given assumptions, and how we can generate these formalised sequences from formal models of the interactive system.

2. A technique for modelling interaction sequences as Finite State Automata (FSA). This contribution addresses research question one. We will demonstrate how we can convert a formalised interaction sequence to a finite state automaton to allow for the generation of varying forms of that interaction sequence. We will describe the benefits of this conversion.

3. A technique to control the state space of interaction sequences using the *self-containment property.* This contribution addresses research question two. We will discuss why the existing formal theory of FSA is not enough to address the state space explosion problem. We will introduce the self-containment property and show how we can use this to reduce and expand the state space as required. We will demonstrate how this property gives control over the state space to ensure that our models remain tractable for testing purposes.

4. Simulation of interaction sequences as an aid to testing. This contribution addresses research question one and three. We will demonstrate a "proof-of-concept" tool to illustrate the use of interaction sequences for testing. In addition we will demonstrate the use of the self-containment property to enhance this process. Finally we will discuss the generation of abstract tests and ways in which these can be converted to concrete tests for implementations of the SUT.

1.4 Structure of Thesis

This thesis is structured as follows: in chapter two we discuss background material that relates to our work, with a particular focus on relevant literature which seeks to address similar research questions to our own. We will summarise existing approaches to explain how our work relates to these and how it differs from these.

This will be followed in chapter three by a discussion on formalising the interaction sequences. This will include defining the interaction sequences as an abstraction, generation of the interaction sequences based on formal models of the SUT, and a discussion of the different types of interaction sequences.

In chapter four we demonstrate how formalised interaction sequences can be used to build FSA, to enable the exploration of sequences of varying lengths for specific tasks. A discussion of the existing techniques in FSA theory will demonstrate the need for abstraction and as a result we introduce the self-containment property. In addition we will demonstrate the use of tasks and task ordering to constrain interaction sequences in order to build tractable models of the SUT.

In chapter five constraining interaction sequences will be discussed in depth, with an emphasis on controlling the interaction sequence model state space. We will formally define the self-containment property and the useful aspects of using this property to control the state space. Finally, we will finish with an example of this technique on a specific safety-critical interactive system, the Alaris General Purpose Volumetric Infusion Pump (Alaris GP Pump).

In chapter six we introduce a "proof-of-concept" tool to demonstrate the simulation of interaction sequences using interaction sequence models as well as a formal specification of the functional component and models of the interactive component. We will also demonstrate the self-containment property functions included in this tool.

In chapter seven we describe how we can generate abstract tests from interaction sequences and how these can be used to explore different testing types for interactive systems. We explore the concept of abstract tests and demonstrate how these can be converted to concrete tests for implementations of the SUT. We will follow with some examples to illustrate the overall technique. In chapter eight we finish with concluding remarks and a discussion for future work.

1.5 Summary

In this chapter we introduced interactive systems and the subset of safety-critical interactive systems. We discussed interactive system testing and its importance. We discussed how interactive system testing cannot prove systems are error free, but that we can use it to identify as many errors as possible for removal, to improve safety and reliability. We introduced interaction sequences and briefly discussed how they can be used as an abstraction of interactive systems. This was followed by a discussion on formal methods and model-based testing. Lastly, we introduced the research questions and the contributions of this thesis.

Chapter 2

Background and Related Work

2.1 Introduction

In this chapter we introduce the relevant literature related to this work. We start with an introduction to the research area, followed by a discussion of the different types of interactive systems. This is followed by an overview of the necessary theory to understand this work. Next, we cover testing concepts and strategies that are applicable to interaction sequences. Finally, we conclude with a discussion of model-based testing methods for interactive systems.

2.2 Research Area

The focus of this research spans two significant areas of software engineering: formal methods and human computer interaction. In particular we look at how these areas are used together, with a specific focus on interactive system testing.

Formal methods are used in human computer interaction in many different ways, one example being to help support and improve testing techniques [78]. Specifically, model-based testing techniques have been developed to ensure that systems behave as we expect, based on specifications in the form of models [76] (as stated previously in chapter 1, a specification formally defines the behaviour of a system). The use of formal methods allows us to specify several different aspects of interactive systems, some examples include models of interaction, tasks, or even relations between the functional and interactive components of a system [67, 57, 7]. We will discuss relevant techniques in detail later.

Traditional formal methods are not widely used in industry, as they are considered too complex, time consuming and not cost effective [17, 82, 33]. Therefore, to encourage formal methods use in industry the area of lightweight formal methods was created [33]. In this research we use a lightweight approach to formal methods to make our methods more accessible.

In human computer interaction, the study of how humans interact with computer systems, our objective is to enhance the interaction in some way. For example, we may wish to improve accessibility, usability or design (see [8, 52, 2] for examples). In this research techniques which are used to observe and/or test the interface are relevant.

As defined by the International Software Testing Qualifications Board (ISTQB) testing is "the process of all lifecycle activities both static and dynamic, concerned with planning, preparation and evaluation of software products and related work products to determine that they satisfy specified requirements, to demonstrate that they are fit for purpose and to detect defects" [31]. The purpose of interactive system testing is not only to identify defects but to ensure that the system does what it is expected to. Defects are often referred to as "issues", "problems", "faults" or "errors" within the system. In particular, errors can refer to human or machine errors while faults are specific to the system and its design. The resolution of these problems will help to "demonstrate that they [interactive systems] are fit for purpose" [31] and help to avoid future incidents.

In this research we draw on examples based on safety-critical medical devices, such as infusion pumps. These devices are used in hospitals to dispense medicine to patients, an example of this type of system is the Alaris GP Pump. Infusion pumps are used as an alternative to administering infusions with a gravity drip. These systems improve infusion safety in that they allow calculated medication delivery rates for a user automatically (instead of having to do this manually), in addition to providing alarms if there is an issue with the feed line, such as the presence of an air bubble. While these systems helped to improve the safety of infusions in regard to these two issues, there are a growing number of injuries and deaths associated with these types of medical devices [26, 47]. This highlights the need for better interactive system testing practices.

2.3 Types of Interactive Systems

WIMP-based systems, systems which include windows, icons, menus and pointers, are still one of the most common types of interactive systems. However, these are only one example of an interactive system. Interactive systems can include a wide variety of interactions which go beyond WIMP-based systems (often referred to as post-WIMP), for example touch-based systems on smartphones, voice-based recognition systems such as voice-identification used in banking, or even airplane cockpit display systems [19] which allow end users to interact with a system using physical buttons and other widgets as opposed to a digital display. That is, the widgets that the user interface is made up of is not limited to WIMP interaction.

Different types of interactive systems have differing types of interaction methods. Many interactive systems support multiple interaction techniques, such as clicking on widgets or observing the information in a display. However, some systems only allow for a single interaction, that is, there is a singular interaction technique in use. An example of this was given in [22] which describes an adaptive traffic control system. This system consists of a traffic light which changes the light signals depending on the traffic approaching it, in order to avoid long wait times. The users only interact with this system via sight, and multiple users can interact with this system at the same time. However, the user also unintentionally interacts with this system via their presence, this is called a low-intention interaction (see [78, p. 185-190]) and we do not consider these types of interactions further here.

Whilst the adaptive traffic light is a simple version of a safety-critical example in terms of its singular interaction technique, there are more complex safety-critical systems which exist that a user also interacts with via single interaction. An example of this is one use of the Apple Watch, where blind-deaf people navigate from the haptic feedback that the watch provides [35]. This has allowed target users more freedom to explore with the knowledge that they can find their way easily. Obviously, extensive testing of this system is required in order to prevent people from being lost or injured and is another good example of why interactive system testing is so important.

While systems can have differing types of interaction, the way in which the system responds to those interactions can also change, depending on the state the system is in. These systems are defined as modal interactive systems, that is the mode of the system determines the system's behaviour. The Alaris GP Pump is one such example of a modal device, in which key presses trigger behaviours based on the state the system is in. As a result of their modality, these types of devices often have less widgets, as the functionality of those widgets may change with the mode.

The converse of a modal system is a non-modal or rather multi-modal interactive system (WIMP-based systems are often multi-modal). In a multi-modal interactive system the mode of the system is not closely linked to the system behaviour. For example, a simple standard calculator performs the functions of addition, multiply, division, and subtraction when the appropriate keys are pressed. These behaviours do not change depending on the mode the calculator is in. In contrast, many calculators now have additional functions which do rely on modes, highlighting how modes can be used to increase the number of functions available in a specific system.

Safety-critical interactive systems, as defined previously, are systems in which error can lead to injury or fatalities. Many of these kinds of system can be found in medical settings, for example the Alaris GP Pump. The serious harm or death that the failure of safety-critical systems can cause (and continues to cause see [66, 59, 18, 15, 47]) to end users highlights the importance of interactive system testing and good software engineering practices.

2.4 Models

In this research we use FSA as representations of the interaction sequences, therefore the theory of FSA is relevant here. We refer to Hopcroft *et al.*'s *Introduction to Automata Theory, Languages, and Computation* as the basis for this theory. Concepts which are of interest include FSA definitions [29, p. 13-22], removal of non-determinism in FSA [29, p. 19-28], equivalence of FSA and regular expressions [29, p. 28-35], FSA equivalence [29, p. 64-65], and minimisation of FSA [29, p. 68-71]. We describe briefly how we use each of these techniques.

The formal definitions of FSA are given in [29] and we use these as the basis for our FSA which we will discuss in detail later in chapters 4 and 5.

Furthermore, we will discuss the removal of non-determinism and minimisation of FSA and demonstrate that these techniques were not enough to address the state space explosion problem, as defined in chapter 1 (FSA equivalence is a required concept to understand how these techniques work). In addition to this, we demonstrate how the equivalence of FSA and regular expressions (a regular expression defines a sequence of characters) can be used to take advantage of task ordering in order to build more complete interaction sequence models (complete here meaning a model with higher task coverage, that is the number of tasks specified in the model is increased).

FSA are commonly visualised as a directed graph. Throughout this research we use directed graphs to more easily demonstrate changes made to FSA. For each graph, each node is a circle containing a widget name from the interactive system and every transition is labelled with an interaction corresponding to the next state. We use green states to represent start states and red states for final or accepting states. We will see how this is used to visualise FSA in chapter 4.

For interaction sequence simulation we take advantage of existing models for the interactive and functional components of an interactive system. Presentation Models (PModel), Presentation Model Relations (PMRs), and Presentation Interaction Models (PIMs) as described in [7] allow us to model the interaction component of a system. The PModels model the system behaviour via each state, describing the name, type, and behaviour of each widget. There are two types of behaviours, an interaction behaviour (*I_Behaviour*) and a system behaviour (*S_Behaviour*). An *I_behaviour* is a behaviour of the interface while an *S_behaviour* is a behaviour of the underlying functionality. The PMR allows us to relate the names of the *S_behaviours* to operations in a functional specification (in this case Z specification). The PIM shows the navigation between windows or modes of a system with each state representing a PModel. We use these models to allow us to simulate the interactive component of an interactive system by triggering the *I_Behaviours* and *S_Behaviours*.

The functionality of the SUT is specified using the Z language [34]. We use this in conjunction with the ProZ plugin of the ProB tool[1], which enables animation of the model for the simulation of interaction sequences. The Z specification describes all possible operations of the SUT in terms of changes to observations of the state space. The values of these observations provide the basis for the assumptions we need for simulation. We will discuss this further in chapter 6.

Note that the models we have selected for simulation are chosen as they provide flexibility and have libraries which are easily integrated, however, the simulation and consequent testing technique we present is not limited to these models. For example, the PVSio-web toolkit[2] could be used to model the interactive component, while a Vienna Development Method (VDM) specification (see [10]) could be used to model the functional component. Therefore, the interaction sequence models presented in this thesis are intended to be integrable with several other pre-existing formal models.

2.5 Modelling and Testing Interactive Systems

In this section we will discuss the relevant techniques for modelling and testing interactive systems. This includes discussion of where particular types of models lend themselves to tractability and therefore help avoid state space explosion. This is not intended as an exhaustive list of these techniques, as we only discuss those relevant to this work (for a comprehensive overview of

[1]See `https://www3.hhu.de/stups/prob/index.php/Main_Page`.
[2]See `http://www.pvsioweb.org/`

formal methods in human computer interaction see [78]).

2.5.1 Existing Modelling Techniques

Task modelling is used to model user tasks and has been adapted to model tasks for an interactive system, for example CTT [58] and HAMSTERS [3] (see [42] for an interesting comparison in terms of system coverage and scalability between these two methods). The CTT and HAMSTERS task models focus on the set of steps a user will take to complete a certain task, hierarchically decomposing tasks into smaller and smaller steps. These approaches rely heavily on following a specific framework in order to use them for modelling and testing, for example The Concur Tree Task Environment (CTTE) has been developed to help make CTT more accessible [57]. A heavy reliance on frameworks in this way provides one way to address the state space explosion problem, as we can expect models to be in a certain form (however, intractable models are still possible for large state spaces).

Most frameworks require a specific programming or testing language used within the framework. For example, Dwyer *et al.* present an approach for Java Swing applications in [23], Campos *et al.* present the GUISurfer tool in [13], Paiva *et al.* describe an approach using Spec# in [56], and lastly Masci *et al.* describe an approach for Patient-controlled Analgesia (PCA) infusion pumps and United States Food and Drug Administration (FDA) regulations in [44]. Using techniques such as these ensures the model is created efficiently, however there is the potential that the use of these techniques could be impossible due to an application model with an intractable state space, or lack of access to the software implemented. As the frameworks and tools used force these techniques to follow a strict process, often these approaches are adapted for only one type of interactive system (e.g. WIMP, web-based, etc.). The technique we demonstrate is intended to be applicable to all types of interactive systems.

There is considerable research into interaction sequence mining, in which actual end user sequences are recorded and then analysed depending on certain criteria, see [37, 39] for two such examples. This largely avoids the state space explosion problem as sequence data is gathered, from actual use, which consequently limits their length, and thus the models. Sequences can be recorded in "real world" or from artificial environments. In real world environments, particularly for safety-critical interactive systems, it is perhaps not appropriate to use this post-implementation approach for testing, as if errors do exist they could lead to serious harm or injury to the user [66, 59]. On the other hand, in artificial environments the sequences gathered may be simpler as it is impossible to re-create the real world context perfectly. This led us to explore the idea of modelling and generating the sequences by focussing on the system itself and the interaction it allows a user to perform, as opposed to focusing on context.

Petri nets are another abstraction of interactive systems used in formal methods. They are described in [20] as, "an alternative to automata that would put the *local interaction of components* at the center of modelling." Some work on the use of Petri nets to describe interaction sequences has been given in [14] but with a focus on HCI evaluation rather than on the development of a behavioural model.

Thimbleby suggests abstracting interactive systems using matrix algebra [67]. In this algebra a matrix represents a MxN rectangular array of number data, where M is the number of rows and N is the number of columns in that array. In his work Thimbleby suggests representing each widget and action as a matrix. This allows him to draw on matrix algebra to manipulate the widgets and actions by using matrix multiplication. A combination of matrices by matrix multiplication can represent an interaction sequence, and the final matrix represents the context change in the system from that sequence. Due to the types of examples (small fixed UIs such as mobile phones) it is not clear that this approach is easily adaptable for more complex systems. In addition, the approach has not been developed beyond this since its original proposal.

In [73], buffer automata are used as an abstraction of the interactive system, as an alternative to other formalisms. The benefit of this approach is that it can represent large numbers of states as automata with fewer buffers than states, thus reducing the time complexity of processing. This is used for human-computer interaction purposes to, "define a layer between the physical user interface and the application *etc.*"[73]. These focus singularly on states or modes of the interactive system. Note that this differs from the overlap component as they attempt to model the intersection between the end user and interactive component, rather than the interactive component and functional component.

The symmetry property is introduced in [54, 32] by Ip and Dill and can be applied to directed graphs in order to simplify them. They argue that if a series of states results in the same output, it does not matter which path is taken, as the result will be the same. The author's claim the use of symmetry could, therefore, help to reduce even "infinitely" long graphs, and as a consequence reduce the overall sequence length. Complete Interaction Sequences (CIS) are a way to model the responsibilities (what the system should allow the user to perform) of an interface rather than the user actions [79, 80]. In order to reduce the number of states, strongly connected components, or symmetric components, are identified and abstracted into a 'super' state. This gives a significant reduction in the number of sequences, as well as their length. While these interaction sequences differ from those we present in this thesis (they consider sequences at a higher level of abstraction) we found the identification of specific components as the basis for abstraction was relevant for our work and adapted this in our own approach.

Harrison *et al.* carried out an investigation into the similarities of the models of two different safety-critical interactive systems in [28]. The purpose of this investigation is to see how much a model of a system can be re-used for a similar system. This concept could be adapted to address the state space explosion problem, as we could re-use models for sequences which are similar for different systems. However, as tasks can be completed in a large variety of ways, devices which perform the same tasks may have very different interaction sequences.

2.5.2 Existing Testing Techniques

Campos *et al.* present an approach for test case manipulation using task models based on the HAMSTERS notation [11]. They discuss ways of generating test cases to ensure good coverage with a smaller number of tests. The tests developed from the work in [11] ensure the system allows users to be able to complete tasks. This differs from the work we describe in later sections as we wish to use interaction sequences to demonstrate the system responds as expected on a given sequence during the completion of a task, in order to check ability and correctness.

Some testing approaches which utilise interaction sequences simply use well-known traversal algorithms or variations of these to explore their models. This kind of approach focuses on restricting the sequence length to those generated by specific traversal algorithms. For example, Salem presents an approach using Finite State Machines (FSM) where complex traversal algorithms are used for testing purposes [62]. In [30] Huang *et al.* use weight-based methods to calculate paths of certain length to traverse through the models. Essentially these approaches, and others like them, allow the traversal algorithm to "trim" the model. For example, a weighted strategy only traverses sequences which are more likely to occur based on probability metrics. This type of strategy only works under certain conditions for specific types of software (such as graphical user interface (GUI) based applications as in [30]) and further abstraction is often used to reduce the model's complexity.

Many existing interactive system models are represented visually via the use of transition graphs. We can explore these systematically to test each different area of system interaction. We can also generate the complement of this type of graph (as proposed by [5]) which describes all the transitions that a user should not be able to perform. The combination of these provides a good overall coverage of the system.

There are different ways in which state space explosion in directed graphs can be managed. One approach is to limit by sequence length, which is utilised by Nguyen *et al.* in the creation of their testing tool GUITAR [51]. They utilise interaction sequences to describe systems using Event-flow graphs (EFGs). All sequences of a given length (such as two) are then generated and they systematically explore these sequences in a breadth-first search approach. Constraining sequences to a defined length gives control over the state space size, however, it does also potentially hide behaviours that could be exposed by longer sequences, or combinations of longer sequences. It is important to explore sequences of varying lengths and combinations in order to expose such behaviours and find alternatives to constrain them rather than by pre-defined lengths.

FSA, or more specifically Mealy machines, can also be used to model systems for testing purposes by making certain assumptions about the SUT, and then modelling the system based on input/output pairs [75]. To address the state space explosion problem an extended finite state machine (EFSM) is used which has variables to store important information. For example, a timeout counter variable can be used instead of three duplicated timeout states. This reduces the number of states required to model the SUT and thus restricts the length of the sequences. However, it is possible to have lengthy sequences with no duplication and thus using an EFSM does not guarantee constraining models to a tractable size.

Thimbleby *et al.* focus on infusion pumps and their number entry systems [72, 71, 68, 55]. They discuss the many different number entry systems and show how even systems which look the same can perform differently. This draws attention to the multiple possibilities of both the implementation of, and interactions with, these types of devices. Hence the need to find new ways of constraining interaction sequences and consequently addressing the state space explosion problem.

In this section we have covered several different testing techniques. In the work we have covered, interaction sequences are used in different ways as an abstraction (as in the task models). Several of these approaches are post-implementation, which leaves room for the possibility of causing harm or fatalities in safety-critical settings. It is important to ensure testing is carried out at different points of the software development approach to try and remove as many errors as possible, to improve system reliability and safety. In addition to this, a common problem is to constrain the state space of the models, to ensure models are tractable. Different approaches are taken to this depending on the type of approach, such as constraining sequence length or removing duplication in the model. These all have the potential to hide or ignore sequences that may lead to error.

2.6 Testing Interactive Systems

In this section we introduce some related testing strategies and concepts. We refer to different types of testing as testing strategies, for example unit testing, model-based testing and so on. We will only address testing concepts relevant to this work.

2.6.1 Concepts

In coverage testing, unit testing etc. every test we define must have an "oracle". The oracle is specified as an input output pair, where given a certain input we expect a certain output. If the test shows the oracle pair matches the system has passed the test and if it shows it does not match the system has failed the test (if we ignore the possibility of false positives and negatives). It is essential that oracles are included in tests, as using oracles in tests keeps us accountable, because we have a defined point of pass or failure. Oracles allow us to reason about the specifics of the behaviours as opposed to crash testing. This allows for a different kind of testing, in that it adds detail to the tests we can create and consequently increases the comprehensiveness of the test suite.

Error is something that occurs within the system that was not expected or should not have happened, thus it is incorrect (we use oracles to identify error). As stated previously, "Program testing can be used to show the presence of bugs, but never to show their absence" [21]. Thus, although we can do our best to discover as many bugs as possible, it is impossible to learn we have discovered them all. Therefore, we need to design better systems which, when faults occur, do not result in catastrophic failure of the entire system. Testing of interactive systems to find areas where errors occur is necessary to achieve this goal.

Safety and liveness are two properties of interactive systems that testers would like to ensure. Safety means that “nothing bad will happen”, while liveness means that “eventually something good will happen” [41]. Note that we do not consider time issues or time-critical systems here where this definition of liveness is not adequate because we may have a system which requires a safety-critical operation to occur every five seconds. If we can prove both of these properties for a system it is considered to be safe, as we can guarantee that it guards against hazardous events, and proving liveness allows us to guarantee that despite potential delays we will eventually get a positive result. In terms of safety-critical interactive systems these properties are very important, because if proved it is highly unlikely for that system to cause death or injury to the end users. This is not impossible however, because as Dijkstra said errors may still exist that we have not found. Proving that these properties hold in an interactive system is highly complex, and thus a deterrent for testers to complete due to the significant financial and time cost.

In addition to this, specifying exactly what these properties mean can quickly become complex. For example, given the definitions above what do we define to be “bad” or “good” and how do we ascertain this. This is also highlighted by the fact that the liveness definition provided here is not adequate for time-critical systems. While we have a simple starting point in terms of these definitions this emphasises the complexity of testing, in that we cannot easily define tests for these properties with the definitions as described. Further decisions are required around what the terms bad and good actually mean in order to prove that a system has these properties, and this can be dependent on the type of SUT.

A ‘Hazard’ is defined in interaction system testing as a behaviour in the system that has the potential to cause harm to end users. If a widget or sequence is described as hazardous it means that there is the high potential for a fault to occur when interacting with that widget or completing that sequence. Reliable systems minimise these hazards.

A common concept in interactive system testing is the idea of end user or human error. This is when an error occurs that has been caused by the end user of the system, that is they have performed an incorrect action based on their desired goal. The opposite of this is machine error, when an error occurs that has been caused by the hardware the system is running on. However, neither of these classifications is quite correct, as they are often used to mask poor system design.

This view has changed [38], as in many cases we now consider the error as the result of poor system design or the "programmer's fault" that the user or machine error has exposed. Therefore, to prevent either human error or machine error we need to engineer better systems which are more fault tolerant (prevent errors or respond safely when errors occur) and provide better training to end users to prevent mistakes in interaction because we can never control what the user will do or fully predict their behaviour.

In coverage analysis we analyse a test suite for the coverage of some metric on the SUT. For example, we can analyse how many lines of code are called and executed by a test suite. This allows us to identify gaps in the test suite, with the overall goal to reach complete coverage. However, this is not always achievable as the state space of a SUT could be large.

2.6.2 Testing Strategies

In our work we are interested in formal modelling and model-based testing (see [63]). In formal modelling the system is abstracted in the form of a model, either from the requirements, prototypes or the system itself. This allows us to hide any unnecessary detail of the interface that is not relevant to testing, such as font types or button positions. While these can have an impact on usability we consider this best identified by usability evaluation techniques as a complimentary measure. From these models we can then learn about the system as well as generate tests in order to verify that the system meets its specifications. Modelling is a powerful technique as it allows us to focus directly on a certain aspect of the system via abstraction, whether that be the functional or interactive components of the SUT. We will use formal modelling in this research using FSA as a representation of interactive sequences.

There are several different types of software testing techniques (see [49] for a complete list) that have been used with the different formal modelling techniques. The most common of these is white-box testing, which is defined as testing with a knowledge of the internal structure of the system. In the case of interactive systems this means testing the functionality of the interactive system. Functionality testing is a mature field and as a result there is no need to explore this further with interaction sequences. It is expected that an appropriate existing technique will suit to test the functional component. However, this is not necessarily the case, and it is essential that all areas of the system are tested to ensure adequate coverage of the state space.

Black-box testing is defined as treating the system as a black-box, that is the tester has no knowledge of the internal structure of the system. By using the exterior of the system, in this instance the user interface, and defining tests based on this interface, a tester can use the implementation to check that the system meets its specifications.

An approach which recognises the need to test both the interactive and functional parts of an interactive system is defined in [1], in which the authors adapt a grey-box testing approach. This is where the user has knowledge of the interface and some knowledge of the functionality. The approach presented in

[1] is inappropriate for generating interaction sequences, however, it is useful to learn more about grey-box testing specific to interactive systems in general.

GUI testing is used to test interactive systems by interacting with the GUI. This is a black-box testing strategy and can be used in a variety of ways to test the SUT. Capture-replay techniques are one example of this type of testing. However, as the GUI can change as the system is improved and consequently updated, these types of tests can be fragile (fragile is defined as easy to crash as the system updates). Therefore, testing only via the GUI is not always the best choice.

The most commonly applied testing strategy for interactive systems is robustness testing. This is testing the robustness of a system, meaning that we inspect the way the system handles unknown faults during execution (unknown faults referring to crash points in the system). While this type of testing allows us to find these fault points easily, these tests lack comprehensiveness (which can only be included in the form of oracles). The reason this type of testing lacks this comprehensiveness is because we systematically explore the system in some pre-defined way looking for crash points, oracles are not used to inform this process. Therefore, it is not a complete approach to testing interactive systems because it lacks this comprehensiveness.

Fault prevention is exactly that, preventing faults before they occur. There are five different types of fault prevention. These are: fault avoidance; fault removal; fault mitigation; fault forecasting; and fault tolerance. Each focuses on a different technique to help prevent faults from appearing in the system. It is important for the dependability of software for fault prevention to occur, particularly in safety-critical devices. Some examples of fault prevention in use are given in [24] and [53], which also describe the importance of making the system resilient so that when humans inevitably make a mistake during interaction it does not result in catastrophe. By exploring possible user paths using interaction sequences, some of which could be erroneous, we can use sequences to increase resiliency in a similar way.

Safety testing is the process of creating a test suite to determine the system's level of safety. This involves proving that a system has true safety and liveness properties. We could use interaction sequences to perform safety testing by searching for, and identifying, hazardous sequences, providing statistics on hazardous combinations as sequence exploration continues. However, we do not explore this technique further here.

Mutation testing involves taking an existing test case and "mutating" it using some pre-defined technique to generate a new test case. These mutations allow us to explore variations of tests that we would not otherwise consider or define. We could mutate interaction sequences to explore several different variations. However, given that modal devices change the system behaviour based on modes, we would get unexpected outputs on these mutations. Furthermore, we cannot define an oracle for these types of tests, as we cannot know what to expect based on the mutation, therefore, mutation testing has the same issues as robustness testing. However, this type of testing would allow us to see what interaction sequences are possible and what effects they have, such as leading to an error or hazard.

Random testing is another black-box testing strategy which allows us to interact with elements of a system based on a pseudo-random algorithm. In terms of interaction sequence testing, we could use this to execute several different types of random sequences. Random sequences are generated based on random selection of steps in tasks. We could then use these sequences to generate abstract tests which ensure nothing hazardous happens. This would allow us to show the reliability of the interactive system.

Top-down and bottom-up testing are two different testing strategies with an incremental approach to integration testing. Integration testing involves testing parts of a computer system in integration to see where faults lie. Top-down testing involves testing components from the highest level of some predefined hierarchy and moving down through the components, while bottom-up testing starts at the bottom level of the hierarchy and moves up through the components. We could envision using these within different hierarchies of interaction sequences.

Acceptance testing is a formal testing method where a system is tested to ensure it meets some set of requirements, that is whether a system is acceptable or not based on those requirements. For example, the FDA has a set of safety requirements for infusion pumps, we could use acceptance testing to show that a given pump meets these requirements. Acceptance testing could be used with interaction sequences.

Hazard analysis is the process of analysing a system identifying particularly hazardous areas. In order to perform hazard analysis the analyser needs access to significant data of actual users interacting with the system [46, 43]. Furthermore, ignoring the potential hazards is defined as "heedless programming", meaning that the programmer has not taken care to avoid detectable hazards [69]. In order to avoid heedless programming and reduce the hazards in a system it is important to detect these during the testing phase of development. However, without the data required for the hazard analysis this can be quite difficult to do, this means that hazards have the potential to occur before being detected. By using interaction sequences and being able to generate different interaction sequences we could be able to build a history of the system without the need for gathering user data which will allow us to perform hazard analysis. However, as our focus is on creating comprehensive tests we do not explore this idea further here.

Path testing is a test strategy where paths of execution (whether for the functionality or interaction) for a system are specified and used to design test cases. Obviously this is applicable to interaction sequences, as a sequence is a path through the interactive system. We can use these sequence paths to help us define test cases for the interactive system.

Invalid testing is a test strategy where incorrect input values are used to ensure that systems fail as expected by ignoring bad input or failing safely. We could use this technique in combination with interaction sequences to input known hazardous sequences, with the expectation that the system will prevent this sequence from occurring or handle this appropriately. If the system is designed well it should prevent error from occurring.

2.7 Summary

In this chapter we discussed formal methods and human computer interaction and described how our work relates to this field. In particular we discussed the different types of interactive systems, models, model-based testing, and testing of interactive systems.

In this research it is intended that the different types of interactive systems will have no effect on the strategy presented here. We assume as interaction sequences are an abstraction of the system, they will be able to be used to abstract any form of interactive system. That is, we are not simply limited to GUI-based systems or WIMP interfaces.

We discussed FSA, PModels, PIMs, PMRs, and Z specifications and stated how we use these to model interactive systems in addition to the relevant concepts required to understand our techniques. This was followed by a discussion on the existing modelling and testing interactive system techniques relevant to this work.

In particular, our models differ from task models, as the point of a task model as used in software design is to specifically ensure that the system allows the end user to carry out some task. We simply use tasks as a grouping mechanism for the interactions, our tests will take a different approach in that we ensure that tasks performed using specific interaction sequences can be completed as expected. Frameworks are often used to support the techniques we discussed, it is our intention that testing with interaction sequences will be able to be used to support existing testing processes instead of introducing a completely new framework. This is done to ensure that the testing strategy is adaptable to different types of interactive systems. Furthermore, we want our testing approach to be more comprehensive than in robustness testing and traversal algorithm approaches. Therefore, we require access to knowledge of the SUT, which we assume is available as part of a good software engineering approach. This is important as we will be defining a test suite which is more comprehensive than one which follows (sometimes arbitrary) pre-defined paths (as in robustness testing).

We highlighted common issues with sequence mining, in particular with safety-critical devices. It is not always safe to gather sequences from a device in use, in addition to the ethical considerations behind gathering this data. Therefore, we must find ways to generate interaction sequences, to avoid these issues, in particular to avoid unnecessary harm to end users.

Interaction sequences are an abstraction of the SUT, however, some techniques discussed also use abstraction within models to simplify them. This is an interesting concept and we will explore this idea further in later chapters to show how we can use the self-containment property to control the state space of our models.

Despite extensive testing of interactive systems, errors can still occur. It is our intention that interaction sequences will allow us to inspect the interactive system behaviour with a focus on the overlap component, providing better coverage of that behaviour and consequently an improved testing strategy.

To ensure we create a more comprehensive testing approach than those discussed above we must incorporate oracles. This will keep us accountable when looking for errors. We will search for several different types of error, with a focus on fault prevention. We will utilise model-based testing and path testing to create and design our tests. This will be discussed later in chapter 7.

Interaction sequences are applicable to several of the different testing strategies discussed above, in addition to being used in some of the existing techniques. This demonstrates interaction sequences suitability as an abstraction of an interactive system to support interactive system testing. This further supports their use as a suitable model of the interactive system.

Chapter 3

Formalising Interaction Sequences

3.1 Introduction

In this chapter we discuss interaction sequences and how they are used as an abstraction of an interactive system for modelling (with a focus on creating a model-based testing approach). We explore the benefits of using interaction sequences and discuss reasons for formalisation. This is followed by a description of how we formalise the interaction sequences with appropriate examples. We introduce the Alaris GP Pump to use as an example of an "in use" system. This pump will be used as the basis for our examples throughout this work.

We discuss generating interaction sequences, more specifically how we can vary the length of sequences using assumptions based on observations from a functional specification. We discuss where these assumptions originate and provide examples using the Alaris GP Pump. We also discuss the Z specification and how this allows us to specify the values of these assumptions.

Lastly, we discuss the different "types" of interaction sequences, that is the different contexts in which we can view the SUT. We discuss why task-widget based sequences were chosen to be explored throughout the remainder of this work. We demonstrate how these allow us to constrain sequences and again give examples using the Alaris GP Pump. In the final section we conclude with a summary of this chapter.

3.2 Interaction Sequences as an Abstraction

In this work we use interaction sequences as an abstraction of the interactive system. In this section we discuss why we chose interaction sequences, and why and how we formalise these sequences, we conclude with a discussion on this formalisation.

3.2.1 Why Interaction Sequences?

Interaction sequences provide a view of the overlap component in terms of the sequence being executed or simulated on the interactive component, in combination with the responses received from the functional component. The sequence itself is the series of steps to be simulated or executed on the user interface, while we make use of assumptions to describe the expected behaviour of the functional component. We explore this in more detail in the following section.

As interaction sequences provide us with this view of the overlap component they are applicable as an abstraction of the interactive system to specifically inspect the overlap behaviour. We use these abstractions of interaction sequences in a lightweight formal methods approach to create and build models of the interactive system which we will then use to inform a model-based testing strategy. Therefore, interaction sequences allow us to investigate ways to test this overlap using a simple abstraction. In addition to this, while testing the functional and interactive components separately is an important and necessary part of the testing process, issues may still arise in the overlap component. Therefore, to ensure system reliability and safety we must investigate ways to also test this overlap.

As defined previously, an interaction sequence is the steps a user can take to complete a task or arbitrarily explore a system. In general, a sequence is something which specifies an order of items. Within the interaction sequence, we are specifying steps of user interaction one after another (that we determine from the system itself, system requirements, prototypes or so on). A step is some interaction that a user may have with the interactive system, specifically with the interactive component or rather user interface of the system. For example, a user may press a button or observe a display. The interaction sequence consists of a series of these steps, that is we can build a sequence from these steps. There are two ways to do this, either by creating a sequence for a specific purpose, such as a task, or to arbitrarily explore the system.

When we create a sequence for a specific purpose this means there is some goal for the sequence, that is the end user is trying to complete some task. In an interactive system there are several different types of tasks a user can complete, each task has some pre-defined goal or state that the user wants the system to be in. For example, in the task of switching off a device the user has finished interacting with the device and as a result wants to put it in the off state. The goal of the task is to put the system into the off state, while the task itself is switching off the system. We can specify an interaction sequence which allows us to achieve this. Tasks and goals are often more complex than this example, as the user may wish to specify certain values, or the system may not be in the correct state for the user to complete the task and so on.

Therefore, there are several considerations required in creating an interaction sequence for a specific task.

The second way we can create interaction sequences is to arbitrarily explore a system. Instead of having a specific task for the end user to complete we can simply select arbitrary steps to build a sequence. This is defined as a "random sequence" in that an end user is "randomly" selecting steps to create an interaction sequence. This type of sequence also has several considerations, for example an arbitrary sequence may not have any effect on the system or in a safety-critical setting the end user may inadvertently cause harm or death to themselves or to others. To avoid these issues we can generate both arbitrary and/or specific sequences during a testing phase.

Therefore, while interaction sequences on the surface appear to be a simple abstraction to inspect the overlap component behaviour and consequently test that behaviour, there are several considerations for these sequences which must be explored before they can be defined and used. These include specifying values for input, the state of the system and effect of state changes, possible harm to the end user, and lastly length of a sequence and constraining that length. One of the main contributions of this work is to define ways to formalise and model interaction sequences and explore the different ways in which this abstraction can be used to support interactive system modelling and testing. We begin by formalising the interaction sequences to simplify and "solve" the considerations as listed above.

3.2.2 Formalisation of Interaction Sequences

In order to simplify the interaction sequences and their associated considerations as discussed in the previous section we need to specify an approach for defining the sequences in a structured way. Several different interaction sequences were inspected from previous research and experimentation and a common form in which to specify the steps was identified. Arbitrary and specific sequences can be built using these structured steps, however further formalisation is required for task-specific sequences.

For a task-specific sequence, or more generally a sequence that has a specific goal, we need to formally define a way in which to specify this goal. Certain assumptions for each interaction sequence were specified. It became clear from investigating several different sequences that certain internal values (the observations from the functional component) were needed to ensure the same result occurred after every simulation or execution of a given sequence. This included the state the sequence started and ended in, and the internal values specified for the starting and ending state.

Therefore, to formalise an interaction sequence it must be built of steps of a given form, we must know the starting and ending states of the system and the values or rather observations for a sequence in those starting and ending states. We demonstrate this technique with a small example.

Interaction sequence steps are of the form: "(Interaction) (Widget) (Number of Interactions)". Note that we use the parentheses simply to group each part of the step, they do not appear in the actual step. For example, following on from the previous example of switching off a device we can assume the final step of the sequence would involve interacting with some 'off' button widget. A button is normally something we click or press to interact with, therefore the step might be: "Press OffButton 1". This indicates that the user should press the off button once to complete the task of turning off the device.

The reason widgets and interactions are used is because this allows us to easily divide the sequence steps into small manageable parts. This allows us to easily build up lengthy sequences using a simple three part format as given above. Furthermore, this building of sequences can be automated using PModels and widget interaction knowledge.

In addition, the interaction part of the interaction sequence step is an abstraction of the interaction itself. For example, for a user to “Press” a physical button they must first locate the button, ensure it is the correct selection, then physically press the button. In a WIMP-based system this would further require the user to navigate to the button position on the screen using the mouse and pointer. In a touch-based system, such as a smartphone application, the user would have to locate the correct area of the screen to tap with either their finger or a stylus. By abstracting the interaction, we are able to use interaction sequences to model interactive systems with a wide variety of interactions.

PModels provide us with an abstract view of the interactive component of an interactive system with widgets described as triples of the form: “$((WidgetName, WidgetCategory, (Behaviour(s))$”. To build sequences we begin by modelling the PModels of the SUT, taking into account the widgets and their related actions, for example “Button1” has the action “Press”. In order to be able to build these models and their respective sequences, we must have a thorough understanding of the system. It is expected that in a good engineering design process this knowledge is readily available from task models, user-centred design artefacts, specifications and so on. We make assumptions about the sequence based on internal values of the system (for example, we may want to generate a sequence where a counter variable is 10) and generate steps in the appropriate form.

It is typical in interaction sequences to focus mainly on either direct (see [4, 70]) or response (see [42, 64]) actions. Direct actions are the literal actions performed by the user, for example “Press Ok 1”. Response actions are the actions that the user will perform in response to a change in the system, for example "Observe Display 1". In this work we use both direct and response actions to create a complete set of actions for sequences.

Lastly, we must specify assumptions for the internal values or observations. In a good software engineering approach we expect that there will exist some formal specification or model of the functional component, from this we can gather the internal values of a system. We can also gather the interaction state from some formal specification of the interaction component states (in this case we have used PIMs). For example, in the sequence of setting a device to the "off state", we have an internal value which specifies whether "power" is true or false. The start and end assumptions would be as follows:

Start Assumptions:	**End Assumptions:**
state: On	state: Off
power: true	power: false

Using this technique allows us to define an interaction sequence using a structured format. We use this format to build sequences for different types of interaction sequence abstractions. Using this structure also ensures reproducibility in the sequences we create.

While these assumptions ensure reproducibility of the sequences, these sequences can be of varying lengths and still have the same assumptions. Therefore, we introduce direct and indirect sequences. A direct interaction sequence's length is restricted to the smallest number of steps required to satisfy the assumptions. In contrast, an indirect sequence may still satisfy the assumptions, but in a larger number of steps, allowing for more variation. The importance of this concept will become clearer as we explore FSA in chapter 4.

3.3 Generating Interaction Sequences

In this section we introduce the Alaris GP Pump which is used to demonstrate the generation of interaction sequences. We follow this with a short example on generating an interaction sequence for a specific task.

3.3.1 Alaris GP Pump

The Alaris GP Pump (see figure 3.1) is used in hospitals to dispense medicine to patients. It is one example of a safety-critical interactive system. Throughout this work we will use this example to illustrate our ideas on a device which is already "in use". In this section we will describe the parts of the device which are relevant to our work.

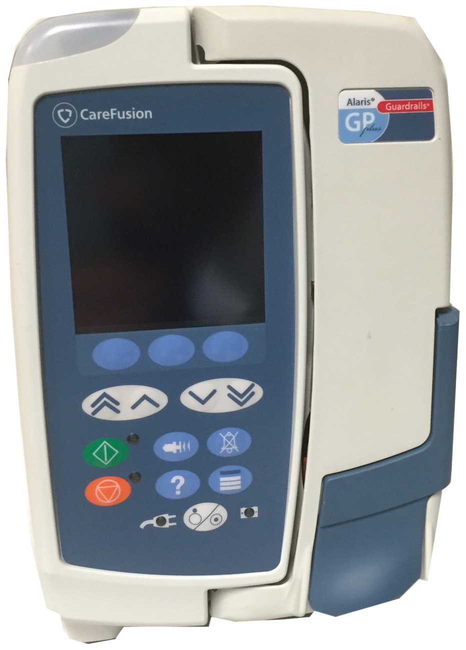


Figure 3.1: Alaris GP Pump with Guardrails

The Alaris GP Pump is a modal interactive system, that is, it has several different modes which change the functionality of the interactive system. There are 30 different modes which specifically relate to the examples we discuss in this work, these are: off, clear setup, power down, confirm profile, rate on hold, rate infusing, profile, select drug category, select drug, confirm rate, volume, volume to be infused (VTBI), set not fitted, prime, options, pressure level, VTBI Bags, attention, dosing, VTBI/TIME, adjust alarm volume, event log, pump details, profile filter, standby, set VTBI, bolus, titrate, titrate dose not permitted, and door open.

The off mode represents the behaviour of the device when the system is turned off. That is, the majority of the widgets have no behaviour in this mode as the device is "off". However, some widgets are still accessible, the most important of these is the on/off button which allows us to turn the device on, the system usually begins in the clear setup mode, or it begins by selecting a profile for an infusion.

The clear setup mode displays to the end user the settings from the last infusion. The end user can observe these settings and make a decision whether or not to clear the settings or continue with the previous infusion values. If the user clears the settings they begin the process of setting up a new infusion, otherwise if they use the previous settings they go to the rate on hold mode.

In the power down mode the system begins to switch off, the end user must hold down the on/off button until the bar on the screen is full before the system will switch off. If the end user lets go of the on/off button before the bar is full the system will return to the previous mode that it was in.

The profile mode allows the user to select which profile they would like to use. The confirm profile mode allows the end user to confirm which profile they would like to use the system in. The profile is based on pre-defined settings for different areas of the hospital. We have used the system in the default ANZ demo configuration, as a result the set of profiles available are: critical care, medical ward, pediatrics, surgical ward, training adult, and training pediatrics.

The end user can select a profile and then certain drugs are available with default values based on that profile.

Default drug categories are stored in the Alaris GP Pump, they are contained under the alphabetical categories for the different drugs available. These are: ml/h, ABCDE, FGHIJ, KLMNO, PQRST, UVWXYZ. The first category, ml/h, allows an end user to specify a drug in millilitres per hour, while the other categories simply filter the drugs available alphabetically.

Once the end user has selected a drug category they may select a drug based on the options available. The default drugs available are: adrenaline, amiodarone bolus, amiodarone inf, dobutamine, fluids and bloods, gentamicin, morhpine, noradrenaline, and propofol. Once they have selected a drug the user can then confirm the rate in the confirm rate mode. Different drugs have different default settings based on the profile and drug selected.

After the user has confirmed the rate they are then in the rate on hold mode. In this mode they can check the settings before starting the medication to infuse. However, before they can begin an infusion they must set up the VTBI in the VTBI mode, this is triggered by an alarm which sets the system to the set VTBI mode.

The VTBI allows an end user to specify the size of the bag of medication that is going to be dispensed. They can pick from a range of pre-defined values in the VTBI Bags mode. Once this selection is made the user can select from a menu of options what action to perform when a bag is empty, they can either stop the infusion, keep the vein of the patient open, or continue an infusion after replacing the bag.

The rate on hold and rate infusing modes are similar except that the rate on hold mode provides the user with all the details of the infusion while the pump is not infusing. In the rate infusing mode the same details are provided except that the pump is now administering an infusion to a patient.

The volume mode allows an end user to see what volume has currently been administered to a patient. It is simply an informative mode to provide the end user with information. There are several modes like this in the device. The options mode provides the user with a list of options so that they can make modifications to the system set up. The event log displays a list of history of the different actions performed on the device including timestamps. The pump details displays the configuration of the device. The dosing mode allows the end user to inspect the dosage. The standby mode allows the device to standby to reduce energy usage.

There is also a set of modes directly related to the alarm and prompting the end user to perform some action. The set not fitted mode (the set is connected to the patient to administer the fluids) alerts the end user that the set is not fitted correctly or perhaps not fitted at all. The attention mode informs the end user that they were in the middle of setting up some infusion but that this is not complete, or that the infusion was set up correctly but not infusing or device is left unattended. The titrate dose not permitted mode informs the end user that they have hit the maximum level titrate for the selected drug. Lastly, the door open mode triggers the alarm and alerts the end user that the door to the set has been opened.

There are also modes which allow the user to modify the pump's setup. The prime mode allows us to prime the infusion before it begins. The pressure level mode allows the end user to make changes to the pressure and alarm level. VTBI/TIME mode allows the end user to set up the VTBI with respect to time instead of the bag size. The adjust alarm volume mode allows the user to adjust the alarm volume. The profile filter mode allows the user to set active and inactive profiles which an end user can select in the profile mode.

The bolus mode allows an end user to administer a drug via a quicker rate. The titrate allows the user to adjust the current infusion rate.

In each of these device modes the widgets of the system execute different behaviours. For example, in the bolus mode interacting with button 1 triggers an alarm beep while in the rate infusing mode the same widget changes the system to the volume mode. The Alaris GP Pump has 23 different widgets, these are: button 1, button 2, button 3, double up, up, down, double down, run, hold, prime, mute, options, level, on/off, alarm light, alarm, timer, door, battery light, on hold light, run light, display and plug light widgets (see figure 3.2). Note that these are the labels we give the widgets for convenience. An end user can both press and press and hold buttons, while they can observe displays, alarms and lights. The timer usually triggers a state change and thus the user observes the display to see these changes.

We have focussed on a subset of the tasks an end user can complete using this device. These 15 tasks are: set up an infusion, starting an infusion, stopping an infusion, pausing an infusion, prime infusion, view summary, set VTBI over time, view pressure volume, adjust alarm volume, view event log, view pump details, modify profile, standby device, set bolus, and modify infusion setup.

The task of setting up an infusion involves specifying all the values for some given infusion. We can then confirm the values that we have input to ensure the rate is calculated correctly. If an end user has completed this task correctly all the correct values will be entered and the system will be in the rate on hold mode.

To pause an infusion the user must transition the system from the rate infusing state to the rate on hold state. This stops the pump from dispensing medicine to a patient. Once a user has paused an infusion they may stop the

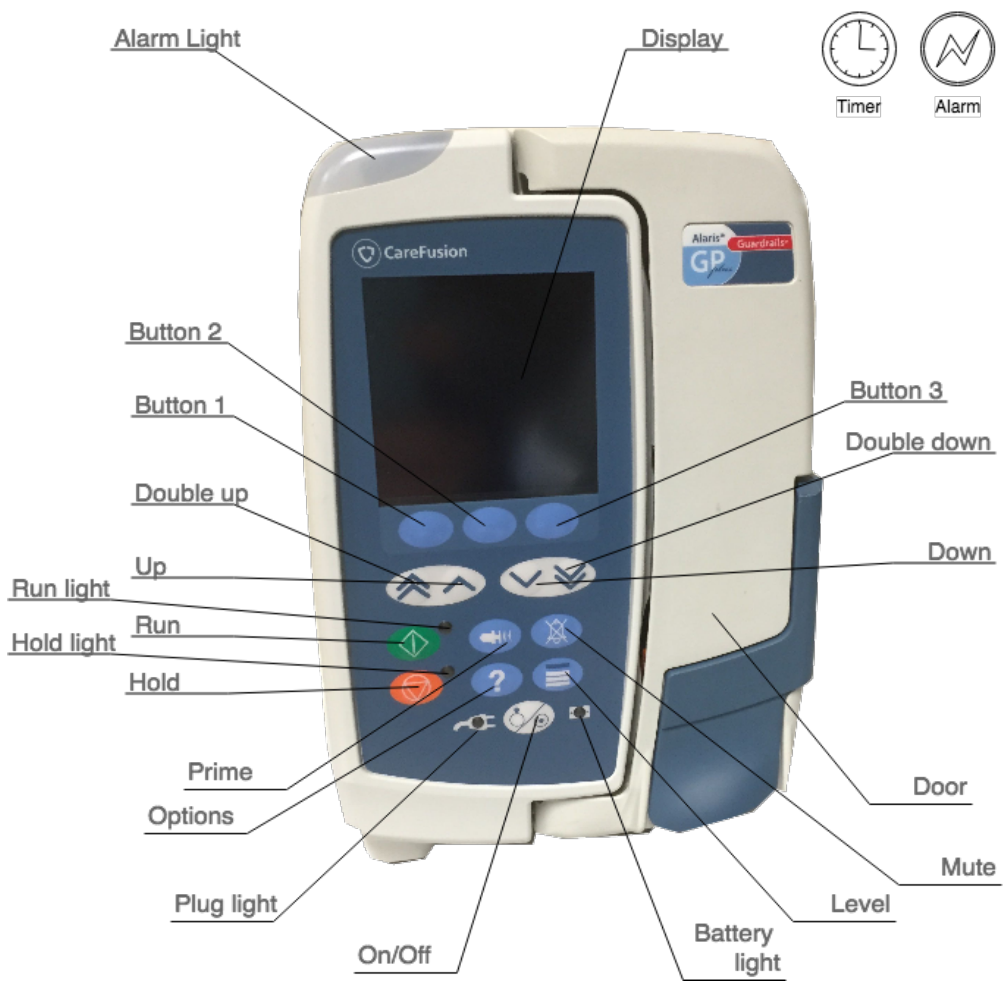


Figure 3.2: Alaris GP Pump's Widgets

infusion altogether if necessary.

Stopping an infusion begins in the rate infusing state, that is there is an infusion which has been set up and started. The user must first pause the infusion and then proceed to stop the infusion. The end user can modify the infusion if required.

To prime the infusion the end user must already have an infusion set up. They then navigate to the prime mode of the system. In this mode they can use the appropriate controls to prime the infusion. The system then returns to the rate on hold mode.

Several of these tasks are related to the options menu of the interactive system. The view summary allows the user to see a summary of the device and infusion currently set up. The set VTBI over time allows the user to set the VTBI rate over some given time and replaces the current VTBI value. The view pressure volume allows the user to check the pressure volume and modify it accordingly. The adjust alarm volume task allows a user to increase or decrease the volume of the alarms as required. The view event log allows a user to interact with the history of actions logged on the device, which is useful for looking up issues or incorrectly input infusions.

In addition to these tasks viewing the pump details shows the set up of the device and software configurations. The modify profile task allows users to modify the active profiles for the system. Standby device allows the user to specify whether the device should go into standby mode or not, after a certain time period. The set bolus task allows the user to modify the bolus value so that drugs can be administered at this new rate. The modify infusion setup task allows the user to make changes to the current infusion. This will pause and stop the current infusion from running.

We can create interaction sequences for each of these tasks separately or together according to task ordering information. This allows us to create a more complete model, in terms of the tasks covered, of the overlap component using interaction sequences, as we can explore sequences which we expect the end users to execute on the actual interactive system.

Furthermore, we can explore this concept of the common mistakes users may make around different tasks using this task knowledge. This means that in addition to exploring the sequences we expect, we can explore varying sequences for each task, ensuring the system guards against commonly known mistakes, based on the sequences we explore and task knowledge.

By defining and constraining ourselves to the tasks we have a domain or group of things we can model and consequently test. This means conversely we have a group of tests not in our domain, that is the opposite of these tasks could also be included in a testing approach. This would allow us to also test the things we do not expect, in comparison to the tasks.

3.3.2 An Interaction Sequence for the Alaris GP Pump

Now that we have introduced the Alaris GP Pump we give a short example of an interaction sequence for this device, in particular a sequence which allows an end user to set up and start an infusion. In this sequence we begin with the device switched off, we select pediatrics as our infusion profile, the drug dobutamine and set a rate of 60. By the end of the sequence the system should be in the rate infusing state and the device should be infusing. The assumptions are as follows:

Start Assumptions:
State: Off
Profile: Pediatrics
Drug category: ABCDE
Drug: Dobutamine
Rate: 60
VTBI: 1000
VTBI Bag Size: 1000
End Rate: Stop
Battery status: Charging
Bolus: Disabled
Dose Rate Soft Min: 1
Dose Rate Soft Max: 61
Dose Rate Hard Max: 100
Infusing: No

End Assumptions:
State: RateInfusing
Profile: Pediatrics
Drug category: ABCDE
Drug: Dobutamine
Rate: 60
VTBI: 1000
VTBI Bag Size: 1000
End Rate: Stop
Battery status: Charging

Bolus: Disabled

Dose Rate Soft Min: 1

Dose Rate Soft Max: 61

Dose Rate Hard Max: 100

Infusing: Yes

The starting assumptions here may not be as expected since the system remembers the values from the previous infusion, hence why values are stored in the off mode. From the PModel of the interactive system we can gather a list of all the widgets of the system and derive the associated interactions. For this example we simply describe the widgets and interactions relevant to this sequence in table 3.3.2.

Note in this table and in the following sections we have changed the widget names to simplify, such as "On/Off" to "OnOff". With these assumptions and interaction knowledge we could simply generate an arbitrary sequence. For example:

1. Press OnOff 1.
2. Observe Alarm 2.
3. Press Button1 3.
4. PressHold Up 10.
5. Press Down 2.
6. Observe RunLight 1.
7. PressHold OnOff 2.

However, this sequence essentially makes no useful changes to the interactive system. A useful change means something that a user might wish to achieve. In this instance the system will not set up and start an infusion, thus our ending assumptions will not be correct. Therefore, task knowledge is required to ensure a meaningful sequence is generated. That is, a sequence which we would expect an end user to perform. It is assumed that in a good software engineering approach task knowledge such as this would be readily available from design artefacts.

Widget Name	**Interaction**
OnOff	Press, PressHold
Alarm	Observe
AlarmLight	Observe
Display	Observe
Button1	Press, PressHold
Button2	Press, PressHold
Button3	Press, PressHold
Down	Press, PressHold
Up	Press, PressHold
Run	Press, PressHold
RunLight	Observe

Table 3.1: Widgets and Available Interactions

However, this does not mean that arbitrary sequences are not a useful abstraction of the interactive system. They could be used in a robustness testing approach to search for crash or failure points within the system to help ensure good system reliability. This is a commonly used approach as discussed in chapter 2. In this work we wish to create sequences which allow us to explore testing strategies beyond robustness testing, this is one reason why we require task knowledge.

Therefore, using task knowledge in conjunction with the PModel we can generate the interaction sequence as follows:

1. Press OnOff 1.
2. Observe Alarm 1.
3. Observe AlarmLight 2.
4. Observe Display 1.
5. Press Button1 1.
6. Press Button3 1.
7. Press Down 2.
8. Press Button1 2.
9. Press Down 1.
10. Press Button1 1.
11. Press Down 2.
12. Press Button1 1.
13. Press Button2 3.
14. Press Up 1.
15. Press Button1 2.
16. Press Up 1.
17. Press Button1 1.
18. Press Run 1.
19. Observe Display 1.
20. Observe RunLight 1.

This type of sequence is what we categorise as a task-widget based sequence. The reason for this is it is generated based on task knowledge using widgets and their interactions. In the next section we will discuss different types of interaction sequences.

3.4 Types of Interaction Sequences

In this section we discuss the different views of the interactive system and how this relates to the different types of interaction sequences. We discuss each different type of sequence and describe why task-widget based sequences are used. We discuss how this choice allows us to constrain interaction sequences. Specifically, we discuss the different tasks for the Alaris GP Pump and how these tasks allow us to constrain the sequences we generate for this interactive system.

3.4.1 Requirements

The larger goal for formalising and modelling interaction sequences is to adapt them for interactive system testing purposes. Taking into account limitations of existing techniques in addition to our research questions this leads to the following requirements for sequences:

1. We must be able to automatically generate sequences of varying lengths so that the testing process is adaptable and usable.

2. We must be able to constrain the sequence length in order to avoid the state space explosion problem.

3. The sequences must allow us to clearly identify why the system did not behave as expected, for example by producing counter-examples.

These requirements allow us to define interaction sequences which are more meaningful, that is interaction sequences which inform more comprehensive testing strategies (than robustness testing). The point of formalising interaction sequences is to, in part, address requirements one and two.

By addressing requirement one we will be able to generate sequences automatically. This is important as it allows us to generate sequences easily in addition to reducing the human effort required to follow the testing process. In general, testing is often neglected or incomplete due to the time and financial costs associated with this process. Therefore, some automation is essential when creating a new testing strategy to ensure adaptability and usability.

For automation to occur we must follow a defined process, as a result we must have some formalised structure to follow, hence the need to formalise interaction sequences. Furthermore, by formalising interaction sequences we can generate sequences of varying lengths. This is important as the longer an end user interacts with a system there is a higher potential for an error to occur. Therefore, to capture this behaviour we must be able to generate sequences of varying lengths.

In addition to this, an end user does not always follow a pre-defined process and can make “mistakes” along the way. In order to ensure we capture this type of behaviour we need to be able to generate sequences which allow for differing lengths to capture these “mistakes”, hence the need for sequences of varying lengths.

Our second requirement is to constrain the sequence length in order to avoid the state space explosion problem as described in chapter 1. We do this in part by using task knowledge, assumptions, and using widget interaction knowledge. We will discuss this further in the following sections and highlight how these are the best option to constrain sequences.

The last requirement focuses on testing, and thus is not relevant in this chapter. We will come back to this requirement when we discuss testing strategies in chapter 7. However, this requirement underpins our work and helps to drive some of the decisions behind automating and constraining the interaction sequences.

3.4.2 Building and Constraining Interaction Sequences Based on Type

We typically consider three different ways to define interaction sequences: state-based; task-based; widget-based. Each of these describes the system from a different perspective. They can be used individually, or in combination with each other to model sequences.

State-based sequences are created by looking at the different states available in the system. For example, the Alaris GP Pump can be in the ‘infusing’ or ‘not infusing’ state. In these sequences we model the different states and actions the user could select to move between the different states of the system. A state change in the Alaris GP Pump would be caused by pressing the run button to begin an infusion (assuming a correctly set up infusion) or pressing the hold button to pause or stop an infusion. One issue with this type of sequence is that they have the potential to unintentionally hide widgets of the system which do not have an observable effect on state. This can result in poor coverage of the system behaviour as it limits all sequences to a restricted set of widgets.

For example, in the Alaris GP Pump the pump is either infusing or not infusing. However, there are several other widgets in the system which we could interact with that change the status of the infusion pump, such as quickly administering the drug via bolus. We cannot capture this kind of behaviour in states, nor determine how much was administered. This issue can be resolved using task-widget based sequences.

Task-based sequences are created by taking a goal the user wishes to achieve and then listing the steps it takes to achieve that goal. In terms of the infusion example, the Alaris GP Pump can be infusing or not, we can specify the necessary steps to complete this task. They are very specific, as they model only the tasks we expect the user to want to achieve. In order to have a high coverage of the system behaviour we should also be able to investigate beyond expected user behaviours, particularly if we want to eventually find hazards which are typically hidden in these unexpected behaviours.

The third type of interaction sequence is the widget-based sequence. We create these by looking at the different widgets that are available and the properties of those widgets. Rather than have a state for ‘infusing’ or ‘not infusing' as we saw in the state-based sequence for the Alaris GP Pump, there are two widgets which have the actions 'startInfusion' and 'pauseInfusion' or 'stopInfusion'. We can then use these to build sequences of actions based on the knowledge we have of each widget irrespective of state or task.

Task-based sequences alone are informal as they are arbitrarily described by informal steps. In order to introduce formalisation we have used the widgets to create steps of a pre-defined form which allows for automation. Widget-based sequences on their own are too free, as they have no pre-defined end point. This allows us to generate sequences of all different lengths without restriction. Therefore, the combination of both the tasks and widgets resolves the informality of the task-based sequences and freedom of the widget-based sequences.

State-based sequences have not been considered in this research as they have the potential to hide widgets of the system, resulting in poor system coverage. Therefore, they do not allow us to easily explore all the state space of an interactive system. If used in combination with the other types of sequences this problem would still occur, hence they are also not suitable for combination (based on our requirements).

For example, using the Alaris GP Pump a state-widget based sequence for setting up an infusion would allow us to specify which widgets to interact with in order to transition through different states (e.g. ConfirmRate to RateOnHold etc.). This would hide the necessary interactions required to input the correct values for the infusion into the system. This could result in an infusion which is either not valid or could cause harm or death to a user. Therefore, we cannot use state-based sequences even in combination with other types.

Keeping in mind our requirements, we chose to use task-widget based sequences. This combination is effective because tasks allow us to constrain the sequence length as they have a defined “end point”, while widgets allow us to easily generate sequences of varying lengths for those tasks. This is because they allow us to create steps of the interaction sequence for easily describable components of the system.

3.5 Summary

In this chapter we discussed formalising interaction sequences. We introduced interaction sequences and discussed why they are useful as an abstraction of interactive systems. This was followed by a discussion on the syntax of these sequences. We introduced the Alaris GP Pump which will be used throughout this research to create an illustrative example of our techniques. We gave an example of setting up and starting an infusion as an interaction sequence based on a task-widget based sequence approach. Finally, we finished with a discussion of our requirements for creating a testing strategy using interaction sequences and considered possibilities for constraining interaction sequences via tasks and widgets. This allows us to address requirements one and two in part. We described the relevant tasks for the Alaris GP Pump which we will explore in this work.

Chapter 4

Modelling Interaction Sequences as FSA

4.1 Introduction

In this chapter we demonstrate the process of using a formalised interaction sequence to create a finite state automaton for a given interaction sequence. This allows us to explore sequences of varying lengths and to take advantage of existing theory (see [29], we discuss this in more detail in the following sections).

We begin with a formal definition of an automaton. We also demonstrate how we display these automata visually as a directed graph. We follow this with a description of the process of modelling a formalised interaction sequence as an automaton.

In the next section we demonstrate how this use of FSA allows for manipulation and constraining the length of interaction sequences. We discuss the relevant existing properties of FSA which can be used to manipulate the sequences and discuss how this allowed for simplification, but was not sufficient for our purposes. We discuss the benefits of non-determinism and the simplicity of FSA, in addition to ways in which we can use task ordering to our advantage and how it is used to create a more complete model of the system behaviour in terms of task coverage. Lastly, we demonstrate this task ordering process with a short example based on the Alaris GP Pump.

4.2 Modelling Interaction Sequences as FSA

In this section we discuss the use of FSA to model interaction sequences. We first give the formal definition of a finite state automaton, followed by the process of converting a formalised sequence to an automaton. We finish with an example using the Alaris GP Pump for the tasks of setting up and starting an infusion.

4.2.1 Formal Definition of a Finite State Automaton

We start by introducing the formal definition of a finite state automaton, followed by the appropriate supporting definitions. We use these definitions to define interaction sequences as FSA and refer back to these throughout this work. We begin with the definition of an automaton (definition 1) based on existing automata theory (see [29, p. 13-22]). Then we give definitions for a path through the automaton (definition 2), as well as for of a connected automaton (definition 3).

Definition 1 *A* finite state automaton *is of the form* $M \overset{def}{=} (Q, \Sigma, \delta, S, F)$ *where:*

1. *Q is a finite set of* states*,*

2. *Σ is a finite set of symbols, the* alphabet accepted *by M,*

3. *δ is a finite set of triples which defines the* transitions *of automaton M, i.e. given states $q, q' \in Q$, input $x \in \Sigma$, we can denote each transition as (q, x, q'),*

4. *S is the set of* start states *and $S \subseteq Q$,*

5. *F is the set of* final (accepting) states *and $F \subseteq Q$.*

Definition 2 *Given a finite state automaton $M = (Q, \Sigma, \delta, S, F)$, a* path *$\rho$ from $q \in Q$ to $q' \in Q$ is a sequence of transitions from δ such that ρ is the empty sequence $< >$, or ρ has first element $(q, x, q'') \in \delta$ and the remainder of ρ is a path from q'' to q'.*

If a path exists between two states $q, q' \in Q$ we say that q' is reachable *from q.*

Definition 3 *A finite state automaton is* connected *iff every state is reachable from a start state as per definition 2.*

For example, using definition 1 we can define an automaton $A = (Q_A, \Sigma_A, \delta_A, S_A, F_A)$ as follows:

$Q_A = \{State_0, State_1, State_2\}$

$\Sigma_A = \{0\}$

$\delta_A = \{(State_0, 0, State_1), (State_0, 0, State_2), (State_1, 0, State_0), (State_2, 0, State_1)\}$

$S_A = \{State_0\}$

$F_A = \{State_2\}$

In finite state automaton A a path exists between states "$State_0$" and "$State_2$", as there is a path from "$State_0$" to "$State_2$" with alphabet symbol "0". Therefore, "$State_2$" is reachable from "$State_0$". Note that we could extend this path to state "$State_0$" is reachable from "$State_2$" if we process an additional two "0" symbols via "$State_1$" (this also implies "$State_1$" is reachable from

"*State*$_2$" as it is part of the path to "*State*$_0$"). As every state in automaton A is reachable from the start state it is also a connected finite state automaton.

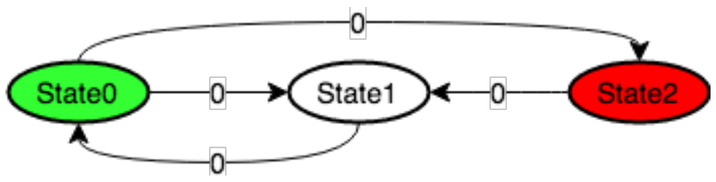


Figure 4.1: Directed Graph of Automaton A

Every finite state automaton can be displayed visually as a directed graph. States are represented by labelled ovals and we denote transitions between states using arrows with the alphabet symbol on the line of the arrow. Start states are denoted by a green coloured state, while final states are denoted by a red coloured state. Figure 4.1 shows automaton A as a directed graph.

4.2.2 Formalised Sequences as FSA

In this section we describe how we can use interaction sequence steps to build a finite state automaton for a given interaction sequence. As previously described in chapter 3, an interaction sequence is built of a series of steps in the form: "(Interaction) (Widget) (Number of Interactions)". Therefore, we can "generalise" a sequence to the following form:

1.$(Interaction)_1$ $(Widget)_1$ $(Number\ of\ Interactions)_1$

2.$(Interaction)_2$ $(Widget)_2$ $(Number\ of\ Interactions)_2$

...

$N.(Interaction)_N$ $(Widget)_N$ $(Number\ of\ Interactions)_N$

Where N is the number of the last step in the sequence, and the ellipsis is used to denote several more possible steps between step 2 and N. Note that we have numbered the steps here for convenience, and that such numbering is not required. In addition, we have included subscripts on each different part of each step to illustrate that they may not necessarily be identical.

Method 1 *We build a finite state automaton* $B = (Q_B, \Sigma_B, \delta_B, S_B, F_B)$ *for any given interaction sequence where:*

- Q_B *is the set of all widgets in the sequence,*
- Σ_B *is the set of all interactions in the sequence,*
- δ_B *is a set of triples* (q, x, q') *where* $q, q' \in Q_B$ *and* $x \in \Sigma_B$ *such that* q *is the widget of the previous step,* x *is the interaction of the current step, and* q' *is the widget of the current step. Note that in* δ_B *the first step of the interaction sequence is a special case as it has no previous step, we solve this by simply using a* place-holder state *called "Initialise",*
- S_B *is a singleton set containing the place-holder state "Initialise". This ensures that any sequence we generate from the automaton begins with the first step of the given sequence,*
- F_B *is a singleton set containing the widget of the last step in the sequence.*

We can use the process in method 1 to automatically convert interaction sequences to FSA via a computer program. This is particularly useful for lengthy sequences and assists in producing sequences automatically (which is one of our requirements as in chapter 3).

Note that as the states in the FSA are the set of all widgets in the interaction sequence this creates a one-to-one relationship between states and widgets. That is, a widget from the interaction sequence maps directly to a single state of the FSA, we will see in later sections why this relationship is significant. Next we demonstrate this process using the example of the Alaris GP Pump for the tasks set up an infusion and start an infusion.

4.2.3 Setting up and Starting an Infusion using the Alaris GP Pump

In section 3.3.2 we gave an example of an interaction sequence for the tasks of setting up and starting an infusion as follows:

1. Press OnOff 1.
2. Observe Alarm 1.
3. Observe AlarmLight 2.
4. Observe Display 1.
5. Press Button1 1.
6. Press Button3 1.
7. Press Down 2.
8. Press Button1 2.
9. Press Down 1.
10. Press Button1 1.
11. Press Down 2.
12. Press Button1 1.
13. Press Button2 3.
14. Press Up 1.
15. Press Button1 2.
16. Press Up 1.
17. Press Button1 1.
18. Press Run 1.
19. Observe Display 1.
20. Observe RunLight 1.

We demonstrate the process of converting this sequence to a finite state automaton $C = (Q_C, \Sigma_C, \delta_C, S_C, F_C)$ using method 1:

- Q_C is the set of widgets in the interaction sequence, therefore $Q_C = \{\mathit{OnOff}, \mathit{Alarm}, \mathit{AlarmLight}, \mathit{Display}, \mathit{Button}1, \mathit{Button}3, \mathit{Down}, \mathit{Button}2, \mathit{Up}, \mathit{Run}, \mathit{RunLight}\}$.

- Σ_C is the set of interactions in the interaction sequence, therefore $\Sigma_C = \{\mathit{Press}, \mathit{Observe}\}$.

- δ_C is a set of transitions as defined above.
  - For the first step, q is the "Initialise" place holder widget, x is the "Press" interaction, and q' is the "OnOff" widget, therefore, the transition is (Initialise, Press, OnOff).
  - For the second step, q is the "OnOff" widget, x is the "Observe" interaction, and q' is the "Alarm" widget, therefore, the transition is (OnOff, Observe, Alarm).
  - For the third step, this action is performed twice, therefore we must include a "loop" to ensure that this is possible, this results in the following transitions: (Alarm,Observe,AlarmLight) and (AlarmLight,Observe,AlarmLight).
  - Continuing this pattern we create the set:
    $\delta_C = \{(\mathit{Initialise}, \mathit{Press}, \mathit{OnOff}), (\mathit{OnOff}, \mathit{Observe}, \mathit{Alarm}), (\mathit{Alarm}, \mathit{Observe}, \mathit{AlarmLight}), (\mathit{AlarmLight}, \mathit{Observe}, \mathit{AlarmLight}), (\mathit{AlarmLight}, \mathit{Observe}, \mathit{Display}), (\mathit{Display}, \mathit{Press}, \mathit{Button}1), (\mathit{Button}1, \mathit{Press}, \mathit{Button}3), (\mathit{Button}3, \mathit{Press}, \mathit{Down}), (\mathit{Down}, \mathit{Press}, \mathit{Down}), (\mathit{Down}, \mathit{Press}, \mathit{Button}1), (\mathit{Button}1, \mathit{Press}, \mathit{Button}1), (\mathit{Button}1, \mathit{Press}, \mathit{Down}), (\mathit{Button}1, \mathit{Press}, \mathit{Button}2), (\mathit{Button}2, \mathit{Press}, \mathit{Button}2), (\mathit{Button}2, \mathit{Press}, \mathit{UpButton}), (\mathit{UpButton}, \mathit{Press}, \mathit{Button}1), (\mathit{Button}1, \mathit{Press}, \mathit{UpButton}), (\mathit{Button}1, \mathit{Press}, \mathit{Run}), (\mathit{Run}, \mathit{Observe}, \mathit{Display}), (\mathit{Display}, \mathit{Observe}, \mathit{RunLight})\}$.
- S_C is a set containing the place holder widget "Initialise", therefore $S_C = \{\mathit{Initialise}\}$.
- F_C is a set containing the widget on the final step, therefore $F_C = \{\mathit{RunLight}\}$.

Note that as δ_C is a set, duplicates are removed, therefore we end up with the minimal number of transitions required for this interaction sequence. Furthermore, actions which are performed more than once produce a "loop". A loop is simply a transition in which $q = q'$. The addition of this loop allows us to capture the ability to perform this particular interaction several times. It also allows for generation of a larger set of sequences of varying lengths for the same task, we will discuss the benefits of this later in chapters 6 and 7.

Using definition 1 and automaton C, we automatically generate a directed graph for this automaton as a visualisation (see figure 4.2) via a computer program. This allows us to visually inspect the automaton, providing an alternate way to reason about and understand the FSA.

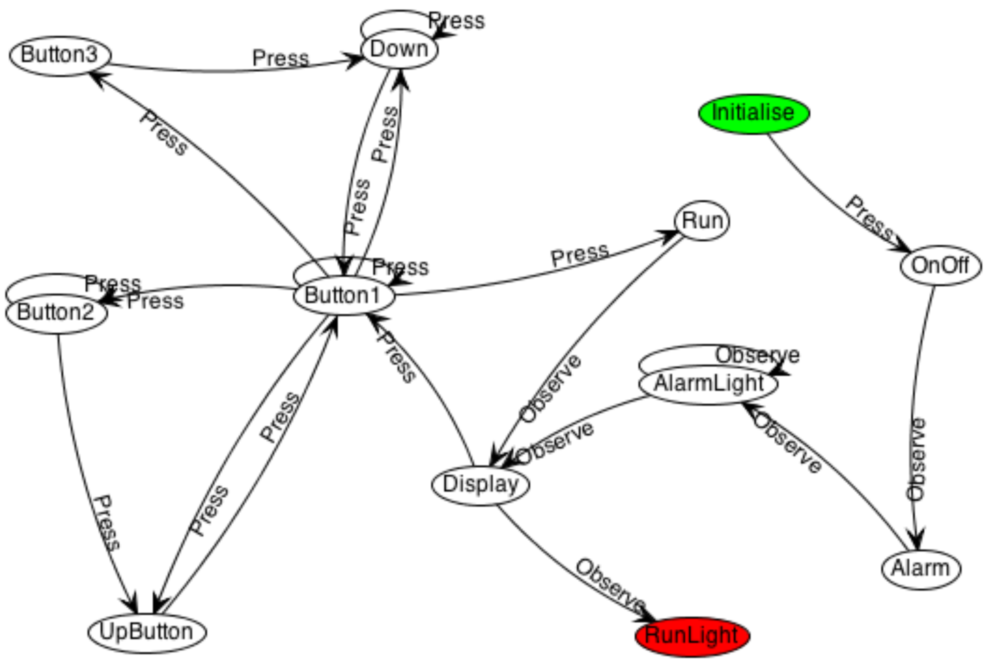


Figure 4.2: Directed Graph of Automaton C

4.3 Using FSA Theory to Constrain Sequences

In this section we explore existing FSA techniques as mentioned in chapter 2. In particular, we demonstrate and discuss the removal of non-determinism and minimisation. We also demonstrate how we take advantage of task ordering to build a more complete model of the system behaviour (in terms of task coverage).

4.3.1 Removing Non-determinism and Minimisation

We first explored ways to constrain the sequences using pre-existing techniques such as the removal of non-determinism and minimisation. Our primary concern was the issue of sequence length and the potential for sequences to be never-ending, with never ending combinations. Therefore, our first investigations were into techniques which would allow us to reduce and control the state space of FSA.

The first technique we investigated was the removal of non-determinism as defined by Hopcroft *et al.* in [29, p. 19-28]. Of particular interest was the section on "The equivalence of DFA's and NFA's" (where DFA stands for Deterministic Finite Automata and NFA stands for Non-deterministic Finite Automata) [29, p. 22-24], and "Finite Automata with ϵ-moves" [29, p. 24-28]. Here we discuss how we use these techniques for interaction sequences and discuss why they did not constrain sequences sufficiently.

Hopcroft *et al.* prove in "The equivalence of DFA's and NFA's" (see [29, p. 22-24]) that we can create a DFA which accepts the same language as some given NFA, that is, they are equivalent. Using this technique we can convert a non-deterministic automaton for any given interaction sequence to a deterministic automaton. While this increases the state space of the automaton constructed, it allows us to deterministically make decisions about the steps

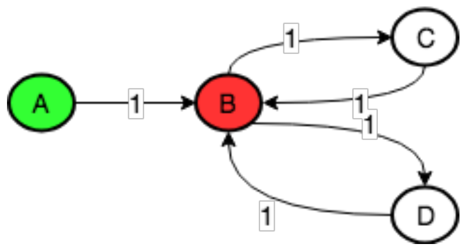


Figure 4.3: Automaton E

of a sequence we are generating.

However, to achieve determinism states are collected together in sets to create new states for the deterministic automaton. This meant that "widgets" which had similar actions were collected together into one "super-widget". This, of course, had implications when generating sequences, such as which widget was actually selected on a given step, and did this matter? Could we select both widgets at the same step, and what effect would this have on the overall sequence?

Consider automaton E in figure 4.3 and the following associated interaction sequence (we ignore assumptions for simplicity):

1. One B 1.

2. One C 1.

3. One B 1.

4. One D 1.

5. One B 1.

In this interaction sequence "One" is the interaction while "B", "C", and "D" are the widgets and "A" is used as the place-holder state. Note for simplicity in figure 4.3 we have used the associated number for the interaction "One". This sequence is reproducible in the non-deterministic version of this

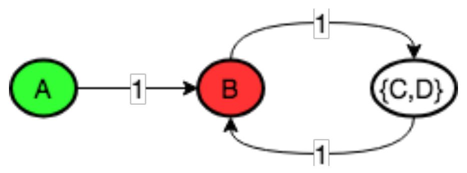


Figure 4.4: Deterministic Automaton E'

automaton, and we can see a clear association between the sequence and the automaton (using method 1). However, if we consider the deterministic version of the automaton as shown in figure 4.4 this is not as obvious.

Taking into consideration the same sequence, we follow this sequence through the automaton (or rather process the word which represents this sequence). Step one is the same as in automaton E, however step 2 has changed. The only way to complete step two is to transition to the state labelled "{C,D}". There is a C in this state, but if we follow the singular mapping of one state to one widget, we have instead processed a "One" action to widget "{C,D}" (which does not exist). Instead we could select either widget "C" or widget "D".

While we have ignored assumptions in this example, assumptions are required for interaction sequences. When using assumptions with a model such as deterministic automaton E', we cannot guarantee which widget would be the correct selection in order to ensure the ending assumptions remain correct. Therefore, removing non-determinism adds a layer of complexity to the model which did not exist in the non-deterministic version.

This occurs because in the existing theory equivalence is defined simply as two automata accepting the same language. Therefore, the non-deterministic E and deterministic E' automata are equivalent, in that they accept the same language. However, because we are using automata in this specific way, mapping of a state per widget name, we are unable to use this technique to help control the state space of the model and consequently the interaction sequence.

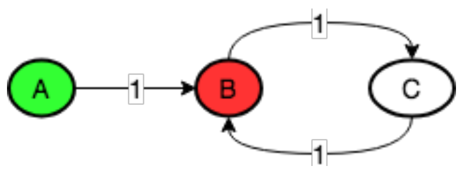


Figure 4.5: Deterministic Automaton E''

B	X	
C		X
	A	**B**

Table 4.1: Calculation of Equivalent States for Automaton E''

As a direct result of this mapping the level of equivalence required for interaction sequence FSA goes beyond the simple definition of equivalent languages, we also require equivalent states (even if widget names vary slightly e.g. "button1" and "buttonOne").

Using minimisation as described by Hopcroft *et al.* in the section "A minimization algorithm" (see [29, p. 68-71]) we can further reduce the state space of automata. In deterministic automaton E' we first rename the states for simplicity of as shown in figure 4.5, we call this automaton E''. This is because the two automaton still accept the same language and are therefore equivalent.

Minimisation depends upon whether there are states which are non-distinct, that is, there are identical states which we can combine to reduce the number of states in the automaton. Two states are identical if they have the same transitions (see [29, p. 68-71] for more detail). To achieve this we construct a table to determine which states are distinct and which are not, as shown in Table 4.3.1, where an "X" between a pair of states indicates they are distinct and " " indicates they are identical.

For example, to determine if state B and state A are distinct we must inspect their transitions. On input symbol "1" state A transitions to state B while state B transitions to state C. Therefore, we must determine if state

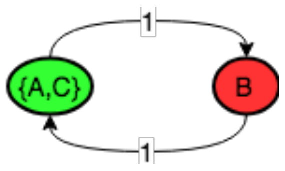


Figure 4.6: Minimal Deterministic Automaton E_{min}

A and state C are distinct. Both state A and state C transition to state B on input symbol "1", therefore these states are non-distinct. However, as per the algorithm, a non-accepting and accepting state is distinct (regardless of transitions), thus states B and A are distinct as are states B and C. These calculations result in the minimal automaton E_{min} as depicted in figure 4.6.

Note that in the renaming process shown in figure 4.5 state "C" represents state "{C,D}", this again is done to simplify the state names. Therefore, in our minimal automaton E_{min} we have a state which groups more than one of the possible widgets for this particular sequence into a single state. Similar to the problem with the removal of non-determinism, we now cannot be sure which widget to select in this state. Furthermore, if we select all widgets at once we invalidate the ending assumptions on the sequence execution.

For example, let widget "C" increase a value by one and widget "D" decrease a value by one. If we interact with both of these widgets at the exactly the same time there will be no observable effect on the interactive system. In particular, if our original intention was to increase the value by 1 we have effectively "un-done" the desired interaction. In addition, in minimal automaton E_{min} the place-holder state A has been combined with state C (which represents both widgets C and D from the original automaton). This is problematic as state A is not a widget and as such we cannot process an interaction "1" to a widget A. As a direct result of the application of this method we have introduced a widget which does not exist. Therefore, for these reasons we are unable to use the removal of non-determinism or minimisation techniques to control and reduce the state space of the models.

While minimisation and removal of non-determinism were not applicable to the FSA used for interaction sequences we decided to investigate further. We found we could build upon other parts of the theory to help control the state space, in addition we were able to use equivalence techniques to ensure that two automata of the same sequence were equivalent. This allows us to show that our new properties are sound. We discuss these new techniques in chapter 5.

4.3.2 Task Ordering

The task-widget based interaction sequences only describe a small part of the interaction, as this interaction is tied to the specific task and the widgets used for that task. However, this use of abstraction of the system behaviour by tasks gives another added benefit, in that the different tasks may be abstracted into individual states. We can use these states to take advantage of task ordering and build more lengthy sequences by combining tasks in specific ways.

Hopcroft *et al.* describe how regular expressions are equivalent to finite automata [29, p .29-26]. In particular they define constructions for union, concatenation and closure. We can use these constructions to combine the FSA for a particular interactive system in meaningful ways taking advantage of task ordering. We demonstrate this with an example using the Alaris GP Pump.

Consider the tasks ‘set up an infusion’, ‘start an infusion’, ‘pause an infusion’ and ‘turn off device’ for the Alaris GP Pump. The Alaris GP Pump only allows an end-user to set up an infusion and start that infusion before they may pause that infusion (this is perhaps obvious), that is there is a pre-defined ordering to the tasks that the user may perform. However, there is nothing

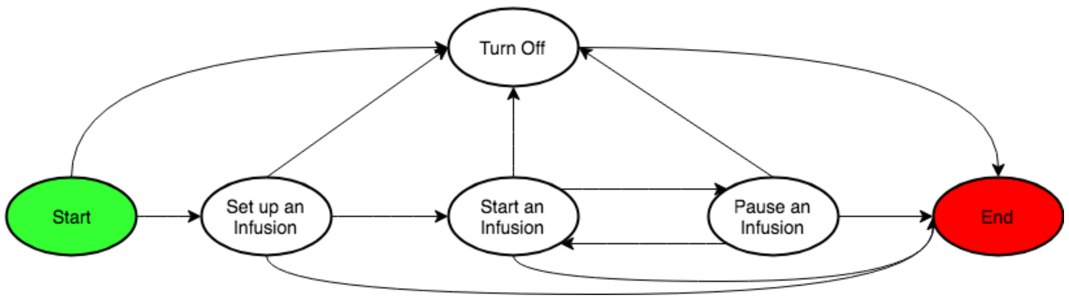


Figure 4.7: An Example of Tasking Ordering for the Alaris GP Pump

in place to prevent the end user from turning off the device, which could have unexpected consequences depending on the state the system is in. This is simplified into a directed graph as shown in figure 4.7, where each state represents a task of the interactive system.

In figure 4.7 it is clear that the tasks of ‘set up an infusion’, ‘start an infusion’, and ‘pause an infusion’ are ordered and therefore can be concatenated. We also have a concatenation between ‘pause an infusion’ and ‘start an infusion’ which creates a loop between these two tasks. Furthermore, after we have completed any task we may switch off the device. This is a simplification of the actual system as a result of the abstraction to tasks, in reality an end user can switch the device off at any time, whether during or after a task is completed. The task of turning off the device, is also in union with the other tasks, as an end user may begin with switching the device off or setting up an infusion. Therefore, using this information we can build a more complete model of this subset of tasks by using these constructions to build an automaton which allows us to generate the interaction sequences in these specific orderings (see figure 4.8), essentially expanding each task state with the corresponding interaction sequence automaton.

Figure 4.8 is the complete automaton F with task ordering observed (visualised as a directed graph). We use individual examples of the concatenation and union to demonstrate how automaton F was constructed. Firstly,

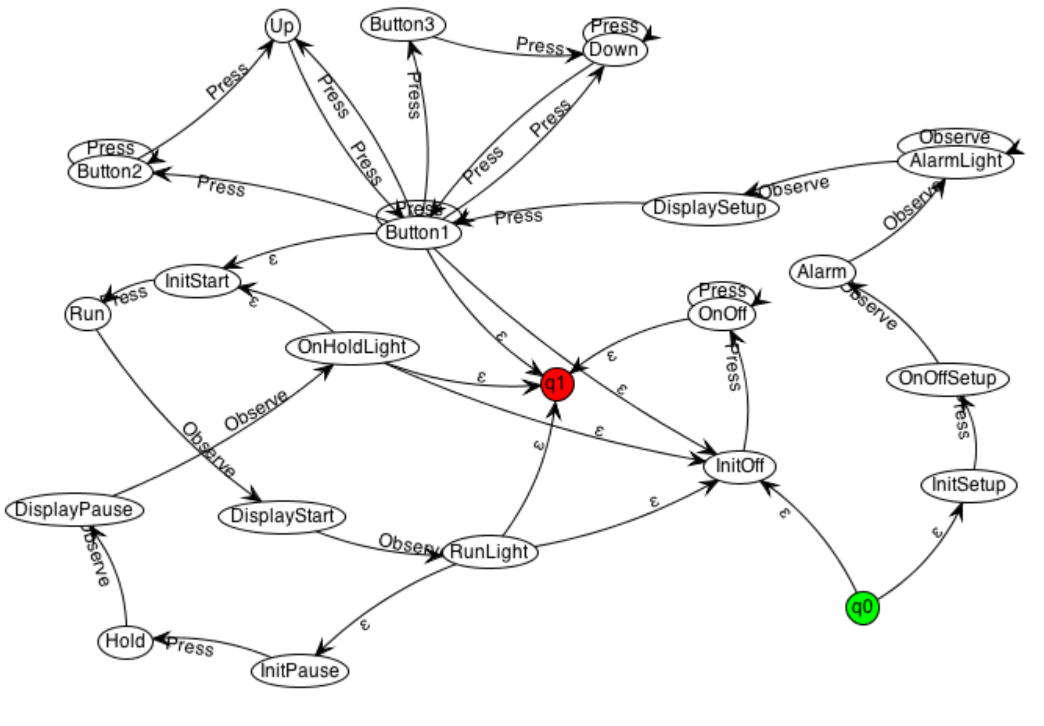


Figure 4.8: Automaton F

as defined by Hopcroft *et al.* a concatenation can be constructed by adding an ϵ transition from the final states of the previous automaton to the start states of the next automaton (see [29, p. 31]). An ϵ transition represents a path in the automaton which does not process an alphabet symbol, that is it represents the "empty" symbol. Therefore, given the automaton for set up a task M_{setup} (figure 4.9) and M_{start} (figure 4.10) we can concatenate M_{setup} to M_{start} using their final and start states respectively. That is, we create a new automata $M_{SetupStart}$ (figure 4.11 which has all the states, transitions, alphabet, start states, and final states of M_{setup} and M_{start}). We add the transition $(Button1, \epsilon, InitStart)$ to concatenate.

Secondly, as defined by Hopcroft *et al.* a union can be constructed by adding a new start state and final state with outgoing and ingoing ϵ transitions, to the start states and final states of the automata respectively [29, p. 31]. Therefore, given the automata for 'start an infusion' M_{start} and 'turn device

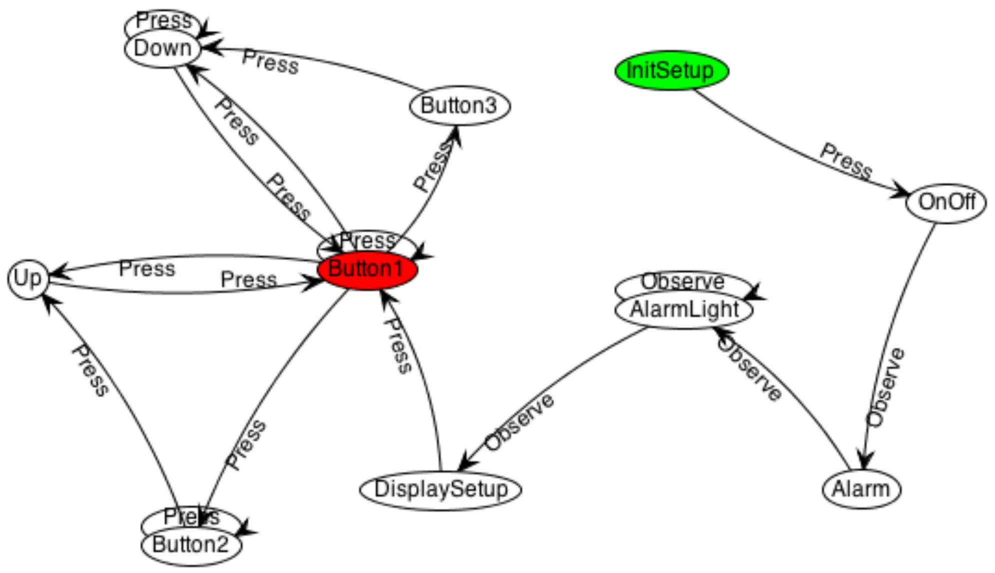


Figure 4.9: Automaton M_{setup}

off' M_{off} (figure 4.12), we apply the union construction by building a third automaton $M_{StartOff}$. Automaton $M_{StartOff}$ has all the states, alphabet, and transitions from M_{start} and M_{off} with new states $q0$ and $q1$ where $q0$ is the start state and $q1$ is the final state. We add ϵ transitions to the start states of M_{start} and M_{off} as follows: "$(q0, \epsilon, InitStart), (q0, \epsilon, InitOnOff)$". Similarly for the final states, we add transitions from the final states of M_{start} and M_{off} as follows: "$(RunLight, \epsilon, q1), (OnOff, \epsilon, q1)$". $M_{StartOff}$ is shown in figure 4.13.

To obtain automaton F (shown in figure 4.8) we have applied concatenation and union as specified by the task ordering. That is, there are the following concatenations: 'set up infusion' to 'start an infusion'; 'start an infusion' to 'pause an infusion'; 'pause an infusion' to 'start an infusion'; 'set up infusion' to 'turn device off'; 'start an infusion' to 'turn device off'; and lastly 'pause an infusion' to 'turn device off'. In addition to the concatenation, there are the following unions: 'set up infusion' and 'turn device off'; 'start an infusion' and 'turn device off'; and lastly 'pause an infusion' and 'turn device off'.

Task orderings can quickly become complex, resulting in intractable mod-

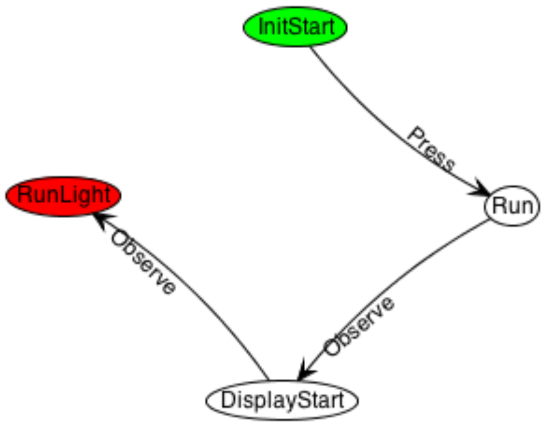


Figure 4.10: Automaton M_{start}

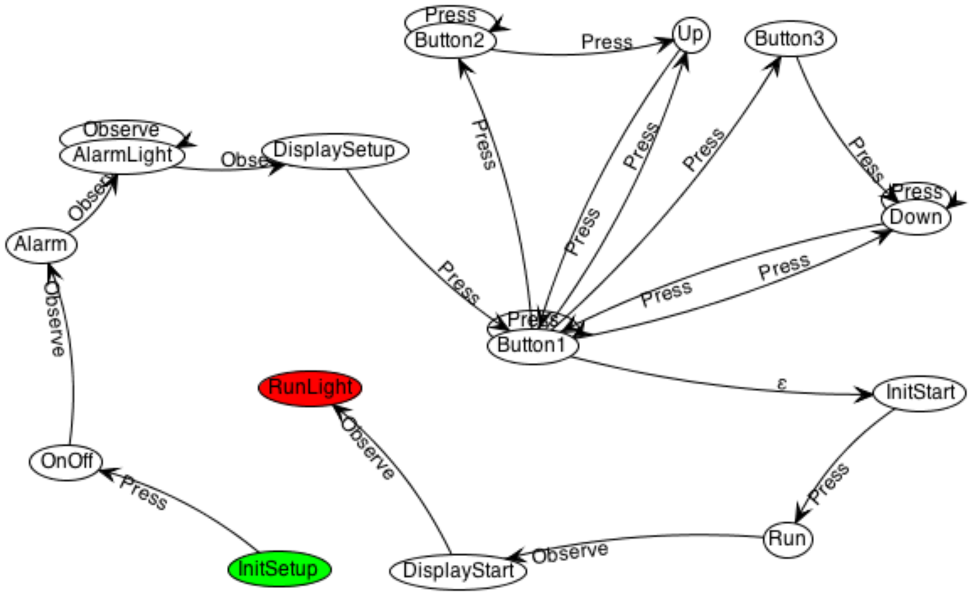


Figure 4.11: Automaton $M_{SetupStart}$

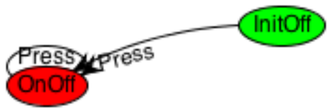


Figure 4.12: Automaton M_{Off}

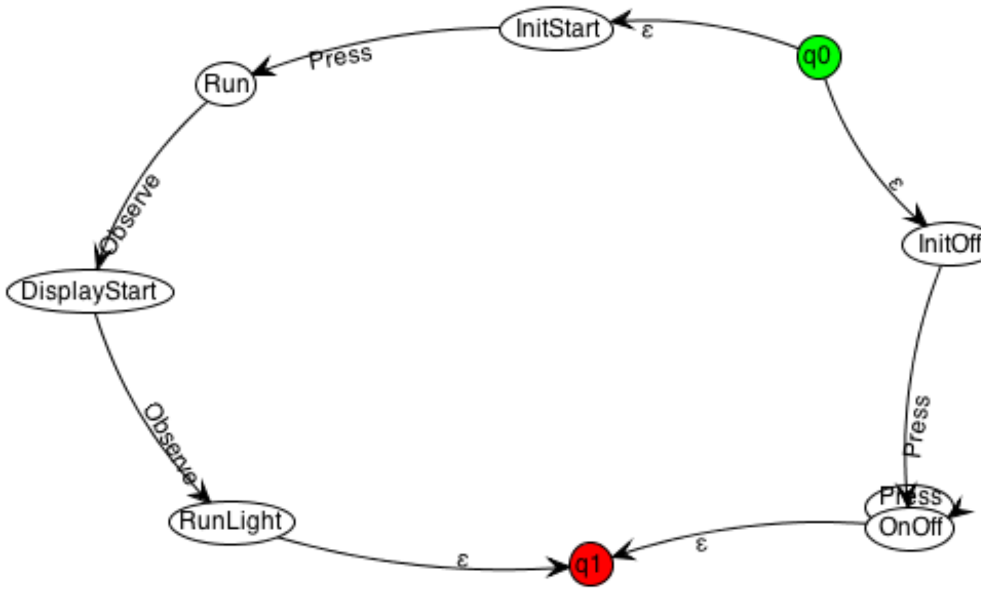


Figure 4.13: Automaton $M_{StartOff}$

els. For example, in automaton F we have multiple states which refer to the display of the device such as “DisplayStart”, “DisplayPause”, and “DisplaySetup”, unnecessarily increasing the state space of automaton F. This is because in order to use concatenation and union the states of each FSA must be labelled uniquely. Therefore, we needed to investigate ways in which we could improve these abstractions. It became clear through further investigations and exploration with interaction sequences that certain widgets, and consequently tasks, could be grouped together in “self-contained” units. A “self-contained” unit is defined as a group of states with one incoming transition to the unit and one outgoing transition, with all other transitions going to some other state within the group. We decided to investigate this idea further which led to the development of the self-containment property which we formally define in chapter 5 where we also demonstrate that is has interesting and useful properties.

4.4 Summary

In this chapter we discussed modelling interaction sequences as FSA. We began with a discussion on modelling formalised interaction sequences as FSA and introduced the formal definition of these automata, as we will use them for interaction sequences. We defined a path within an automaton, reachability, and connectedness. In addition we demonstrated how FSA can be visualised as a directed graph. We also described the process of converting an interaction sequence to a finite state automaton and followed this with a short example based on the Alaris GP Pump.

This was followed by a discussion on the use of existing automata theory and the benefits and drawbacks of existing techniques. We demonstrated how the removal of non-determinism and minimisation led to over-abstraction and hiding details of widget selection.

Lastly, we explored the idea of task ordering and how this can be used to build more complete models of the interactive system behaviour using interaction sequences and automata constructions. We discussed potential issues with using the existing constructions, such as scalability. This led to further exploration and resulted in the development of the self-containment property, which we present in the next chapter.

Chapter 5

Constraining Interaction Sequences

5.1 Introduction

In this chapter we discuss constraining interaction sequences, with a focus on the state space of FSA. We review the existing techniques in FSA theory which were used to attempt to address this issue and the problems with using these techniques. We discuss how this led to the identification of the self-containment property and formally introduce definitions for this, followed by proofs which demonstrate the usefulness and correctness of this property. This is followed by a discussion on the ways in which this technique can be used with FSA to model interaction sequences and control the state space. Finally, we finish with a short example using the Alaris GP Pump.

5.2 Problems with Existing FSA Theory

In chapter 4 we discussed and demonstrated how the removal of non-determinism and minimisation could be used to constrain interaction sequences. The removal of non-determinism allows for deterministic choices when generating interaction sequences, while minimisation allows for a reduction in the state space of these models.

However, there were several issues identified when applying this existing theory. The primary concern was the way in which equivalence was defined which affected these techniques. As defined by Hopcroft *et al.* in [29], equivalence is defined as two automata which accept the same language. This equivalence definition is used throughout the removal of non-determinism and minimisation techniques to demonstrate that the resulting automaton accepted the same language as the original automaton.

While the two automata were equivalent according to this definition, this "broke" the relationship between widgets and states in task-widget based interaction sequences. This is because of the relationship we had defined between states and widgets, with one state per widget. Essentially the "collecting" of states in these techniques made this relationship false. That is, there was more than one widget per state. Therefore, while the two automata were equivalent from a language perspective, they were not equivalent from a widget perspective. In terms of the interaction sequence, this means that each widget is assumed to be unique (even if that is not necessarily the case).

This issue would be acceptable if the widgets which were grouped together were equivalent, in that they executed the same functions and affected the system in exactly the same way. However, this was not always the case, with states sometimes including every widget within a system. Therefore, the removal of non-determinism and minimisation could not be used to help constrain the interaction sequences.

While this may suggest reconsidering the use of interaction sequences, as they are too large or complex to deal with, and that another type of sequence or abstraction may be better used, as stated in chapter 3, the other types of sequences have limitations which prevent their use here. Furthermore, it is unclear how to change the type of sequence and abstraction as the most obvious solution is widgets and their interactions (hence task-widget based) or user based (user based meaning based on a human end users and their interactions with the system). Using user based would force us to model the human component, which can be unpredictable and complex, and cannot be achieved without assumptions to eliminate this. Therefore, a task-widget based approach is the best selection based on our requirements, and avoids the issues associated with other types of sequences as discussed in chapter 3.

As discussed in chapter 3, interaction sequences provide us with a clear view of the overlap component of an interactive system, and therefore would allow for a testing approach to be built using this abstraction for the overlap. In addition to this, as described in chapter 2, interaction sequences are already used as an abstraction in different ways by other researchers, therefore they are already widely accepted as an appropriate abstraction of the system.

However, it is clear that interaction sequences must be constrained somehow. In the existing research in this area this issue is prominent in addition to the conceptual issue where sequences can be never ending, with never ending combinations of these never ending sequences. Furthermore, as we investigated more interactive systems it was evident that the number of widgets could be quite large, resulting in a large state space due to the one widget to state relationship. For example, in a WIMP system there is often more than one widget which performs the same function, such as a save icon and save menu item, both of which allow an end user to save a document. Therefore, it is necessary to find other ways in which to constrain the interaction sequences, as existing FSA theory is not sufficient.

As investigation continued into applying interaction sequences to different types of interactive systems, and in particular larger interactive systems, it became evident that certain groups of widgets could be "collected" together. For example, in a software system that has a "print preview" as part of its functionality there is a set of widgets associated with printing the document and then another set associated with formatting that document (often in different windows), as a result these widgets are distinct. We expect that it is likely an end user will finish formatting a document before printing that document, as a result these widgets are essentially contained to their own individual groups.

Further investigation suggested that we could take advantage of these groupings to define which areas of the interactive system to model and consequently test. This would provide greater control over the state space of the models to a tester, in that these groupings could be utilised to constrain the test coverage to specific areas of the system. We will discuss this idea in further detail later.

Therefore, to constrain interaction sequences we explored the idea of grouping these widgets together into self-contained units. These self-contained units could be used as an abstraction within the models. That is, we could group a set of widgets together into one state, effectively "hiding" these states, and could return to them later if required. This led to the creation of the self-containment property.

This type of abstraction is not new and has been used previously for strongly connected components within a directed graph as an abstraction (as discussed in chapter 2). That is, the idea of using a property or rather component to abstract inside FSA is not new. We build on this approach and use it in a new way to allow abstraction on the self-containment property. The new material described here is the definition and proofs for the self-containment property which we can use to abstract and expand our interaction sequence automata. This self-containment property allows us to abstract a larger subset of automata than strongly connected components, we will discuss this in more detail later.

5.3 The Self-containment Property

Here we introduce the self-containment property. In what follows we define: the self-containment property (definition 4); abstraction (definition 7); and expansion of these automata (definition 8) also supporting definitions for: alphabet function (definition 5) and override function (definition 6). We follow this with lemmas (and their proofs) to show that these definitions have the useful properties we expect.

5.3.1 Definitions

Definition 4 *Given automaton* $M = (Q, \Sigma, \delta, S, F)$ *we define an automaton* $M_s \overset{def}{=} (Q_s, \Sigma_s, \delta_s, S_s, F_s)$ *which is* self-contained with respect to M *iff:*

1. $Q_s \subseteq Q$, $\Sigma_s \subseteq \Sigma$, $\delta_s \subseteq \delta$,

2. *M_s is closed with respect to M, which means that if any transition in δ starts and ends in Q_s then it is in δ_s too:* $\delta_s = \{(q_s, x, q'_s) \mid (q_s, x, q'_s) \in \delta \wedge q_s, q'_s \in Q_s\}$,

3. *The only transitions of M that start outside M_s and end inside M_s are those that end in start states of M_s: for all $(q, x, q') \in \delta$, if $q \in Q \setminus Q_s$ and $q' \in Q_s$ then $q' \in S_s$,*

4. *The only transitions of M that start inside M_s and end outside M_s are those that start in final states of M_s: for all $(q, x, q') \in \delta$, if $q \in Q_s$ and*

$q' \in Q \setminus Q_s$ then $q \in F_s$.

The automaton A (see figure 5.1) has six automata which have the self-containment property. Note that by this definition we could have each individual state as a self-contained automaton, however this is not useful in terms of abstraction as it does not allow us to reduce the state space. Furthermore, as we will prove later in lemma 2 every automaton is self-contained with respect to itself. This means we could reduce the state space down to a single state with no transitions using the self-containment property, again this is not useful as while we reduce the state space we hide any interesting information about the automaton.

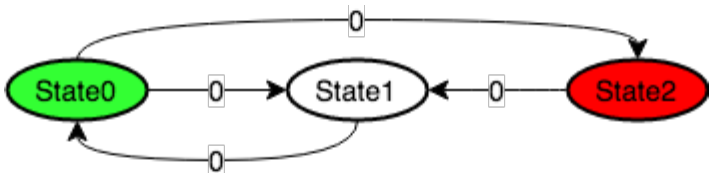


Figure 5.1: Directed Graph of Automaton A

Definition 5 *There is an* alphabet function *such that, for any automaton $M = (Q, \Sigma, \delta, S, F)$ we have $\alpha(\delta) \overset{def}{=} \{x \mid (q, x, q') \in \delta\}$.*

Definition 6 *For any automaton $M = (Q, \Sigma, \delta, S, F)$ we can override its set of transitions δ as follows with the* override function*:*

$$ {}^{P}_{p'}\delta^{Q}_{q'} \overset{def}{=} \left\{ \begin{cases} (p'x, r'), & \text{if } r \in P \\ (r, x, r'), & \text{otherwise} \end{cases} \mid (r, x, r') \in \delta' \right\} $$

where

$$ \delta' \overset{def}{=} \left\{ \begin{cases} (r, x, q'), & \text{if } r' \in Q \\ (r, x, r'), & \text{otherwise} \end{cases} \mid (r, x, r') \in \delta \right\} $$

The alphabet function defined in definition 5 allows us to retrieve the symbols from an automaton based on its transitions, this will become useful in the following definitions as we modify transitions. The override function defined in definition 6 is what allows us to modify the transitions in order to perform abstraction.

Note: In what follows, we are dealing specifically with interaction sequences, thus FSA will always be connected, however, the proofs do not rely on this. We also assume that an automaton's alphabet is exactly the set of symbols that label its transitions, i.e. for all FSAs $(Q, \Sigma, \delta, S, F)$ we have $\alpha(\delta) \overset{def}{=} \Sigma$.
End note.

Definition 7 *Given automaton $M = (Q, \Sigma, \delta, S, F)$ where $S \neq \emptyset$ and $F \neq \emptyset$ (we call M the* automaton to be abstracted on*), automaton $M_s = (Q_s, \Sigma_s, \delta_s, S_s, F_s)$ where M_s is self-contained with respect to M, and an* abstract state Ω *where $\Omega \notin Q, Q_s$ there exists an* abstract automaton $M_a \overset{def}{=} (Q_a, \Sigma_a, \delta_a, S_a, F_a)$ *where:*

1. $Q_a = (Q \backslash Q_s) \cup \{\Omega\}$,
2. $\Sigma_a \subseteq \Sigma$,
3. $\delta_a = {}^{F}_{\Omega}(\delta \setminus \delta_s)^{S}_{\Omega}$,
4. $(S \cap Q_s = \emptyset \Rightarrow S_a = S) \wedge (S \cap Q_s \neq \emptyset \Rightarrow S_a = \{\Omega\})$,
5. $(F \cap Q_s = \emptyset \Rightarrow F_a = F) \wedge (F \cap Q_s \neq \emptyset \Rightarrow F_a = \{\Omega\})$.

The abstract automaton is essentially the original automaton we started with except with the self-contained automaton hidden. In contrast to hiding the self-contained automaton, removing it would result in an automaton which is not connected, indicating a non-connected interaction sequence. This would be an incorrect model of a sequence as it would be unclear how to process a path through the states which were originally connected to the self-contained automaton. For example, if we abstract the first two states of A (figure 5.1), without the addition of a final state we have an automaton as seen in figure 5.2. Therefore, we introduce the abstract state to indicate that an abstraction has taken place and at which point this has occurred. The transitions that originally included the self-contained automaton states are then overridden to reflect this change. Note that visually we represent abstract states as rectangles as opposed to circles to further indicate this difference between the types of states. In the definitions we refer to abstract states using the omega symbol Ω, in practice we label states beginning with an Ω symbol to denote that they are abstract.

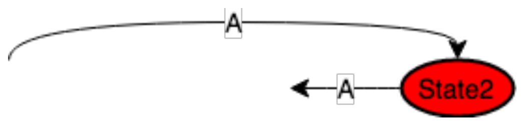


Figure 5.2: Abstract Automaton M_a without Abstract State

Definition 8 *Given abstract automaton* $M_a = (Q_a, \Sigma_a, \delta_a, S_a, F_a)$ *with abstract state* $\Omega \in Q_a$ *and any automaton* $M = (Q, \Sigma, \delta, S, F)$ *with* $\Omega \notin Q$ *(automaton* M *is represented by abstract state* Ω*), there is an automaton* M_b*, which we call the* expansion of M_a with respect to M*, and* $M_b \overset{def}{=} (Q_b, \Sigma_b, \delta_b, S_b, F_b)$ *where:*

1. $\Sigma_b = \Sigma_a \cup \Sigma$,

2. $Q_b = (Q_a \backslash \{\Omega\}) \cup Q$,

3. $\delta_b = \delta \bigcup_{s \in S, f \in F} ({}_f^{\{\Omega\}}(\delta_a)_s^{\{\Omega\}})$, *which is to say* Ω *as a "from" state in a transition is replaced by the final states of* M*, and* Ω *as the "to" state in any transition is replaced by the start states of* M,

4. *If* S_a *contains only* Ω *then* S_b *contains only* s*. Otherwise* $S_b = S_a$,

5. *If* F_a *contains only* Ω *then* F_b *contains only* f*. Otherwise* $F_b = F_a$.

At some point we may wish to explore the sequence in the self-contained automaton, therefore we needed a way to expand the abstract state. Definition 8 shows how we can correctly expand this state, allowing us to reconstruct the original automaton. As a result we can reduce and expand the number of states in the automaton.

5.3.2 Results

In this section we will prove some results that give some evidence that our definitions correctly capture our intuitions.

Lemma 1 *For any automaton* $M = (Q, \Sigma, \delta, S, F)$ *with* $s, f \notin Q$ *and* $\|S\| > 1 \wedge \|F\| > 1$, *there is an equivalent automaton* $M_c \overset{def}{=} (Q_c, \Sigma_c, \delta_c, S_c, F_c)$ *where:*

1. *S is not a singleton set and*

   (a) $Q_c = Q \cup \{s\}$,

   (b) $\Sigma_c = \Sigma \cup \{\epsilon\}$ *where* ϵ *is the blank symbol,*

   (c) $\delta_c = \delta$ *and for all* $(q, x, q') \in \delta_c$, *if* $q \in S$ *then* $\delta_c = \delta_c \cup (s, \epsilon, q)$,

   (d) $S_c = \{s\}$,

   (e) $F_c = F$.

2. *F is not a singleton set and*

   (a) $Q_c = Q \cup \{f\}$,

   (b) $\Sigma_c = \Sigma \cup \{\epsilon\}$,

   (c) $\delta_c = \delta$ *and for all* $(q, x, q') \in \delta_c$, *if* $q' \in F$ *then* $\delta_c = \delta_c \cup (q', \epsilon, f)$,

   (d) $S_c = S$,

   (e) $F_c = \{f\}$.

Proof: Section 2.2 [29, p. 26] states that a string w with which contains ϵ (ϵ representing the blank symbol) is equivalent to w. Therefore, by theorem 3.8 from [29, p. 65] the new automaton is equivalent to M as it accepts the same language.

□

Task-widget based interaction sequences have a defined single start and end point to the sequence due to the nature of tasks, and thus have singleton start and final state sets. However, we could have automata which do not. Lemma 1 shows that for any automaton there is an equivalent automaton with singleton start and final state sets, thus we do not have to include this as a restriction. Note that in addition to this the original definition of equivalent from Hopcroft *et al.* is sufficient as we are comparing two FSA in this proof, not FSA for interaction sequences.

Lemma 2 *Given an automaton $M = (Q, \Sigma, \delta, S, F)$, M is self-contained with respect to itself.*

Proof:

1. Immediate.
2. Immediate.
3. There are no states of M outside M, therefore implication is true (since false implies anything, *ex falso quod libet*).
4. Similarly to 3.

□

Lemma 2 proves that for any given automaton it is self-contained with respect to itself. This addresses the state space explosion problem in the most extreme case as we can now take any automaton and reduce the state space to exactly one state, the abstract state. However, this also results in lack of all information for that automaton as it is hidden inside this abstract state. While this solves the state space explosion problem, it is not particularly useful or interesting, especially not in consideration of adapting the sequences and their consequent models for testing. However, as we will discuss later, it is useful when building models of large systems from their components.

Our main result is that, under certain circumstances, we can take an automaton M, abstract it with respect to automaton M_s (where M_s is self-contained with respect to M) to get abstract automaton M_a, and then expand M_a with respect to M_s to get automaton M again. While we have all of the component parts in the definitions above, there is still a crucial relationship amongst the various automata that we are missing, and this is that we have, of course, to be able to re-connect the start and final states as originally intended when expanding the abstract automaton. The definitions so far, while allowing reconnection, lose crucial information about the original start and final states. The property that we require for our main result ensures that this information can be recovered. The property is that if *any* state of the self-contained automaton M_s is also a start state of the automaton M it is self-contained with respect to, then the start states of the self-contained automaton *must be* the start states of the original automaton. We therefore use the start and final states as "markers" to show how the various automata fit together properly when we do the expansion. This also requires that all the automata involved have singleton start and final state sets, but we already know (by lemma 1) that this is not a restriction.

Here we provide an example to demonstrate the information lost if we do not mark states as we describe, that is, the motivations for the SF property (definition 9). Figure 5.1 is the original automaton where we will search for

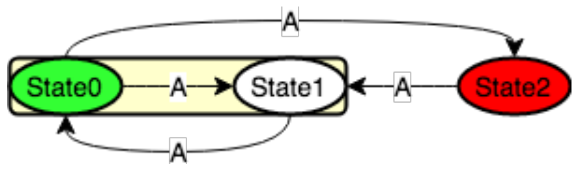


Figure 5.3: Original Automaton M with Self-containment Property

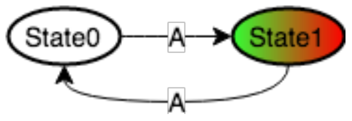


Figure 5.4: Self-contained Automaton M_s

the self-containment property as seen in figure 5.3.

Using definition 4 we can build the following automaton M_s as seen in figure 5.4. Note that "State1" is the start and final state of this automaton. Given the automata M and M_s we can build a new abstract automaton M_a as seen in figure 5.5.

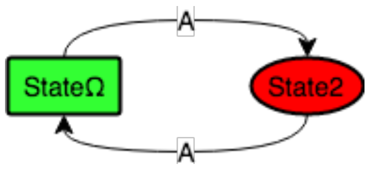


Figure 5.5: Abstract Automaton M_a

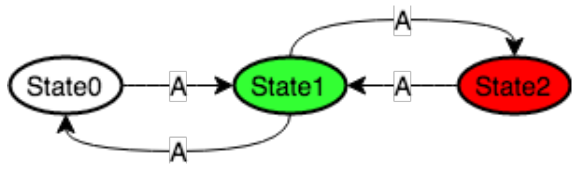


Figure 5.6: Expanded Automaton M_b

However, when we expand M_a with M_s to construct automaton M_b (see figure 5.6) as per definition 8 we do not have an equivalent automaton to M. This is because we did not preserve the start "State0" in the self-contained automaton M_s, hence the need for this extra restriction on start and final state sets.

All this leads to needing the following:

Definition 9 *Given automaton $M = (Q, \Sigma, \delta, S, F)$ and automaton $M_s = (Q_s, \Sigma_s, \delta_s, S_s, F_s)$ which is self-contained with respect to M, then M and M_s have the* Start Final (SF) property *iff: if* any *state of M_s is also a start state M, then the start states of M_s* must be *the start states of M, i.e.*

$$Q_s \cap S \neq \emptyset \Rightarrow S_s = S$$

and similarly for final states

$$Q_s \cap F \neq \emptyset \Rightarrow F_s = F$$

Note that in our case where we can assume all automata have singleton start and final state sets, these conditions simplify to

$$s \in Q_s \Rightarrow s_s = s$$

and

$$f \in Q_s \Rightarrow f_s = f$$

because $S = \{s\}, F = \{f\}, S_s = \{s_s\}$ and $F_s = \{f_s\}$.

Lemma 3 *Let $M = (Q, \Sigma, \delta, \{s\}, \{f\})$ be any automaton for modelling interaction sequences and $M_s = (Q_s, \Sigma_s, \delta_s, \{s_s\}, \{f_s\})$ be a self-contained automaton with respect to M. We are assuming without loss of generality that automata M and M_s have singleton start and final sets, by lemma 1. We require that M and M_s have the SF property (definition 9). Further, let $M_a = (Q_a, \Sigma_a, \delta_a, S_a, F_a)$ be an abstract automaton with abstract state $\Omega \notin Q, Q_s$,* *where M_s is the automaton abstracted on. Finally, we assume an automaton $M_b = (Q_b, \Sigma_b, \delta_b, S_b, F_b)$ which is the expansion of M_a with respect to M_s. Then our result is that automaton M_b is equivalent to automaton M.*

Proof

We have

$$\delta_a = {}^{\{f_s\}}_{\Omega}(\delta \setminus \delta_s)^{\{s_s\}}_{\Omega} \qquad \text{from definition 7} \qquad (5.1)$$

and

$$\begin{aligned}
\delta_b &= \delta_s \cup {}^{\{\Omega\}}_{f_s}(\delta_a)^{\{\Omega\}}_{s_s} && \text{from definition 8} && (5.2)\\
&= \delta_s \cup {}^{\{\Omega\}}_{f_s}({}^{\{f_s\}}_{\Omega}(\delta \setminus \delta_s)^{\{s_s\}}_{\Omega})^{\{\Omega\}}_{s_s} && \text{substituting from 5.1} && (5.3)\\
&= \delta_s \cup (\delta \setminus \delta_s) && \text{over-riding and then reversing} && (5.4)\\
&= \delta && \delta_s \subseteq \delta \text{ from definition 4 and set theory} && (5.5)
\end{aligned}$$

So also

$$\begin{aligned}
\Sigma &= \alpha(\delta) && \text{by our Note above} && (5.6)\\
&= \alpha(\delta_b) && \text{by substitution and (2)-(5)} && (5.7)\\
&= \Sigma_b && \text{by our Note above} && (5.8)
\end{aligned}$$

Then

$$Q_b = (Q_a \setminus \{\Omega\}) \cup Q_s \qquad \text{by definition 8} \qquad (5.9)$$

$$= (((Q \setminus Q_s) \cup \{\Omega\}) \setminus \{\Omega\}) \cup Q_s \qquad \text{by definition 7 } Q_a = (Q \setminus Q_s) \cup \{\Omega\} \tag{5.10}$$

$$= (Q \setminus Q_s) \cup Q_s \qquad \text{by definition 7 } \Omega \notin Q, Q_s \tag{5.11}$$

$$= Q \qquad Q_s \subseteq Q \text{ from definition 4 and set theory} \tag{5.12}$$

Turning to the start states, recall from definition 8 if S_a contains only Ω then S_b contains only s_s. Otherwise $S_b = S_a$. Within those cases each has to consider whether or not $s \in Q_s$. We proceed by nested cases.

Assume S_a contains only Ω, so $S_a = \{\Omega\}$. (5.13a)

Now we have further cases depending on $s \in Q_s$.

Assume $s \in Q_s$ (5.13ba)

$$\{s\} = \{s_s\} \qquad \text{by definition of 9 and 5.13ba} \tag{5.13bb}$$

$$= S_b \qquad \text{by consequence of 5.13a and definition 8} \tag{5.13bc}$$

Assume $s \notin Q_s$ (5.13ca)

$$\{s\} = S_a \qquad \text{by definition 7, since 5.13ca means } S \cap Q_s = \emptyset \tag{5.13cb}$$

$$= \{\Omega\} \qquad \text{by 5.13a} \tag{5.13cc}$$

contradiction definition 7 requires $\Omega \notin Q$, but $s \in Q$ (5.13cd)

Assume $S_a \neq \{\Omega\}$ (5.13d)

Now we have further cases depending on $s \in Q_s$

Assume $s \in Q_s$ (5.13ea)

$$S_a = \{\Omega\} \quad \text{by definition 7 and 5.13ea} \quad (5.13eb)$$

contradiction by 5.13d (5.13ec)

Assume $s \notin Q_s$ (5.13fa)

$$\{s\} = S_a \quad \text{by 5.13fa and definition 7} \quad (5.13fb)$$

$$= S_b \quad \text{by 5.13d and definition 8} \quad (5.13fc)$$

By cases (twice) we conclude that $S_b = \{s\}$ (5.13g)

Finally to the final states, recall that definition 8 gives if F_a contains only Ω then F_b contains only f_s. Otherwise $F_b = F_a$. Within those cases each has to consider whether or not $f \in Q_s$. We proceed by nested cases.

Assume F_a contains only Ω, so $F_a = \{\Omega\}$. (5.13h)

Now we have further cases depending on $f \in Q_s$.

Assume $f \in Q_s$ (5.13ia)

$$\{f\} = \{f_s\} \quad \text{by definition of 9 and 5.13ia} \quad (5.13ib)$$

$$= F_b \quad \text{by consequence of 5.13h and definition 8} \quad (5.13ic)$$

Assume $f \notin Q_s$ (5.13ja)

$\{f\} = F_a$ by definition 7, since 5.13ja means $F \cap Q_s = \emptyset$ (5.13jb)

$= \{\Omega\}$ by 5.13h (5.13jc)

contradiction definition 7 requires $\Omega \notin Q$, but $f \in Q$ (5.13jd)

Assume $F_a \neq \{\Omega\}$ (5.13k)

Now we have further cases depending on $f \in Q_s$

Assume $f \in Q_s$ (5.13la)

$F_a = \{\Omega\}$ by definition 7 and 5.13la (5.13lb)

contradiction by 5.13k (5.13lc)

Assume $f \notin Q_s$ (5.13ma)

$\{f\} = F_a$ by 5.13ma and definition 7 (5.13mb)

$= F_b$ by 5.13k and definition 8 (5.13mc)

By cases (twice) we conclude that $F_b = \{f\}$ (5.13n)

We have, in 5.2-5.5, 5.6-5.8, 5.9-5.12, 5.13g and 5.13n, that $M = M_b$ as required.

□

Lemma 3 demonstrates the ability to reduce and expand FSA using the self-containment property which provides control over the state space. However, it is important to note that while this provides control over this space, we cannot conclusively state whether or not an abstraction will result in a smaller state space. The reason for this is that the self-contained automata identified may not have a state space smaller than the abstract state and transitions which are used to abstract. For example, the trivial case of abstracting a single state would not result in a smaller number of states or transitions, and as a result does not reduce the state space. Therefore, we leave it up to human reasoning to determine if a particular abstraction is useful or not, we discuss this limitation further in section 5.4.4.

5.4 Controlling the State Space

Using the self-containment property to reduce and expand FSA as required provides more control over the state space, in that we may reduce or expand the state space as desired. In addition to this benefit, we can choose which areas of an interaction sequence to model and which to abstract. This is particularly useful in using the models for testing techniques as we can more easily control which parts of the interactive system to test. The test suite flexibility is increased as a result of this property, as it allows us to make logical groupings about the tests we create. In particular, we can easily document which parts of the system are tests and which are not. In this section we demonstrate potential uses of the self-containment property for this purpose.

5.4.1 A Short Example for an Interaction Sequence

In this section we give a short example of abstraction using the self-containment property. We abstract and expand using this property to construct two different automata, it will be evident from this example that the two automata are equivalent, in that they are exactly the same automaton.

We use automaton $K = (Q_K, \Sigma_K, \delta_K, S_K, F_K)$ (figure 5.7) as follows:

$Q_K = \{State0, State1, State2, State3\}$

$\Sigma_K = \{0\}$

$\delta_K = \{(State0, 0, State0), (State0, 0, State1), (State1, 0, State2), (State1, 0,$
$State3), (State2, 0, State1), (State2, 0, State2), (State3, 0, State0)\}$

$S_K = \{State0\}$

$F_K = \{State3\}$

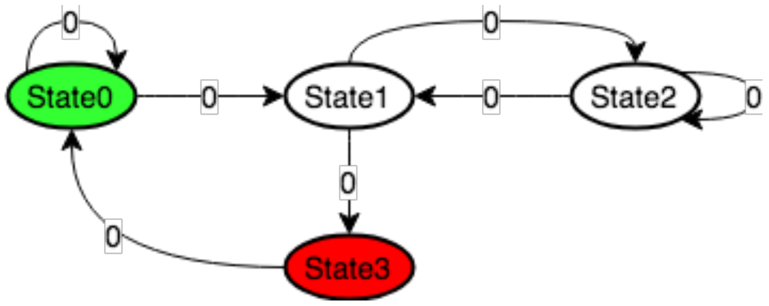


Figure 5.7: Automaton K

Automaton K has eight valid self-contained automata (with respect to automaton K) which we identify using definition 4. We list these automata by the names of the states which they contain:

- The trivial self-contained automata, consisting of one state: $State0$, $State1$, $State2$ and $State3$.

- The self-contained with respect to itself automaton as per Lemma 2: $State0, State1, State2, State3$.

- $State1, State2$.

- $State1, State2, State3$.

- $State0, State1, State2$.

Note that as long as the self-containment property still holds we could select more than one of these automata to self-contain. For this property to hold the states inside each self-contained automaton must not overlap. For example, we can abstract both automaton $State0$ and $State1, State2$ but not $State1$ and $State1, State2$ as $State1$ is contained within automaton $State1$ and automaton $State1, State2$. This may prove useful in larger more complex FSA.

We construct the self-contained automaton $L = (Q_L, \Sigma_L, \delta_L, S_L, F_L)$ (figure 5.8) with respect to automaton K for the states "$State1, State2$" (which were randomly selected to best illustrate reduction and expansion) using definition 4 as follows:

$Q_L = \{State1, State2\}$

$\Sigma_L = \{0\}$

$\delta_L = \{(State1, 0, State2), (State2, 0, State1), (State2, 0, State2)\}$

$S_L = \{State1\}$

$F_L = \{State1\}$

Note that automaton L satisfies the self-containment property as:

- $Q_L \subseteq Q_K, \Sigma_L \subseteq \Sigma_K, \delta_L \subseteq \delta_K$,

- L is closed with respect to K is true,

- There is one ingoing transition from K to L: $(State0, 0, State1)$,

- There is one outgoing transition from L to K: $(State1, 0, State3)$.

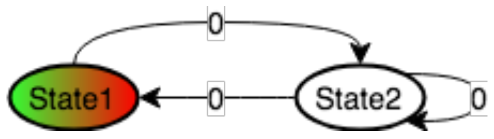


Figure 5.8: Automaton L

Using definition 7 we construct the abstract automaton $N = (Q_N, \Sigma_N, \delta_N, S_N, F_N)$ (figure 5.9) using automata K and L as follows:

- $Q_N = (Q_K \setminus Q_L) \cup \{\Omega 3\} = \{State3, State0, \Omega 3\}$.
- As $\Sigma_N = \alpha(\delta_N)$: $\Sigma_N = \{0\}$.
- $\delta_N = {}^{F_L}_{\Omega 3}(\delta_K \setminus \delta_L)^{S_L}_{\Omega 3} = \{(State0, 0, State0), (State0, 0, \Omega 3), (\Omega 3, 0, State3), (State3, 0, State0)\}$.
- $(S_K \cap Q_L = \emptyset \Rightarrow S_N = S_K) \wedge (S_K \cap Q_L \neq \emptyset \Rightarrow S_N = \{\Omega 3\})$, therefore: $S_N = \{State0\}$.
- $(F_K \cap Q_L = \emptyset \Rightarrow F_N = F_K) \wedge (F_K \cap Q_L \neq \emptyset \Rightarrow F_N = \{\Omega 3\})$, therefore: $F_N = \{State3\}$.

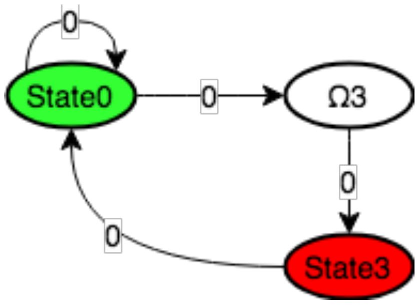


Figure 5.9: Automaton N

Note that state "Ω3" represents the automaton L which we have abstracted from automaton K. At this point we have successfully reduced the state space of automaton K. Using definition 8 and automata N and L we can expand state "Ω3" to construct a new automaton O. This is achieved by "re-connecting" the transitions to and from state "Ω3" as defined, that is the ingoing transitions of state "Ω3" are the ingoing transitions to the start state of L (figure 5.10), and the outgoing transitions of state "Ω3" are the outgoing transitions to the final state of L (figure 5.11). The final step is to remove the abstract state and return "State1" to a non-start and non-final state. This completes the construction of automaton O.

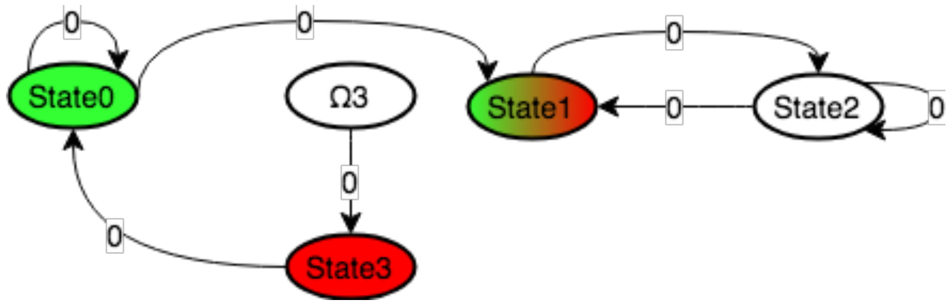


Figure 5.10: Expanding Ingoing Transitions of State "Ω3"

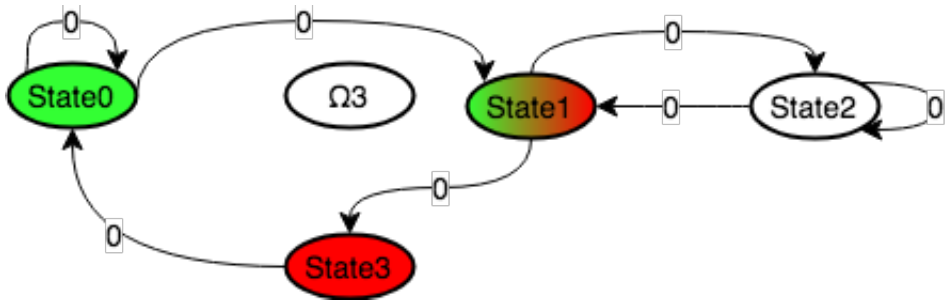


Figure 5.11: Expanding Outgoing Transitions of State "Ω3"

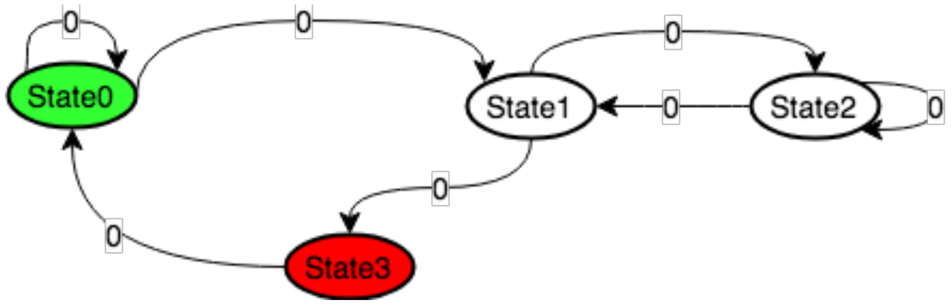


Figure 5.12: Automaton O

Formally, using definition 8 automaton $O = (Q_O, \Sigma_O, \delta_O, S_O, F_O)$ is as follows:

1. $Q_O = (Q_N \backslash \{\Omega 3\}) \cup Q_L = \{State0, State1, State2, State3\}$,

2. $\Sigma_O = \Sigma_N \cup \Sigma_L = \{0\}$,

3. $\delta_O = \delta_L \bigcup_{s\in S_L, f\in F_L} ({}_{f}^{\{\Omega 3\}}(\delta_N)_{s}^{\{\Omega 3\}}) = \{(State0, 0, State0), (State0, 0, State 1), (State1, 0, State2), (State1, 0, State3), (State2, 0, State1), (State2, 0, State2), (State3, 0, State0)\}$,

4. If S_N contains only $\Omega 3$ then S_O contains only s. Otherwise $S_O = S_N$, therefore: $S_O = \{State0\}$.

5. If F_N contains only $\Omega 3$ then F_O contains only f. Otherwise $F_O = F_N$, therefore: $F_O = \{State3\}$.

Note that automaton O (figure 5.12) is equivalent to automaton K by lemma 3, that is automaton M, in this case automaton K, is equivalent to automaton M_b, in this case automaton O. This is true of any self-contained automaton abstracted and expanded on using the self-containment property.

5.4.2 Task Ordering and Self-Containment

In the previous section we demonstrated a short example of using the self-containment property in a "backward" approach. That is, we started with an automaton, found the self-contained automata within that automaton, then abstracted on this property. However, we can also adopt a "forward" approach using this definition as we have proved in lemma 3 that automaton M and M_b will always be equivalent, provided they adhere to the definitions given.

Therefore, we can construct an automaton with abstract states, without knowledge of the information hidden in those abstract states, and expand these states later if required. That is, we can construct the self-contained automata after the construction of the abstract automaton or before. This provides greater flexibility as we need only model the parts of an interaction sequence (and consequently interactive system) that are of interest.

This “forward” approach can be used with task ordering, as we can use the task as a representation of the abstract state, then create interaction sequences to satisfy that task as required. As a result we can focus on certain tasks while ignoring others. This is particularly useful in testing as we can design test suites which acknowledge that other tasks exist but are not important in terms of the test suite at the present time.

This also provides greater control over the state space as we abstract the information not under investigation into a single abstract state to “ignore”. We can simply expand this state later if required. For example, we may wish to focus on testing the safety critical aspects of a system and ignore the non-safety critical aspects. We can use self-containment to contain the non-safety critical tasks and focus solely on those that are safety-critical (provided that they are self-contained). If these non-safety critical tasks appear important for whatever reason we can expand these tasks as required.

It is common that when implementing interactive systems pre-existing libraries from the target programming language or operating system are used for standardised functions. For example, a ‘save dialog’ provided by a library allows an end user to save a document. Testing these standard libraries is often not performed, as it is assumed the creator of the library will have done so. Therefore, we can also abstract the tasks associated with pre-existing libraries into abstract states, assuming that it behaves correctly. This allows us to define clear boundaries around what is being tested and what is not. This highlights when abstracting to a single state using the self-containment property is useful, in that we can either ignore the abstracted information, model only the parts of the system we are interested in, or combine with other sub-models.

We demonstrate this “forward” approach using the task ordering example discussed in chapter 4. We refer to the task ordering directed graph depicted in figure 4.7. In chapter 4 we demonstrated how we could use FSA constructions to create a more complete model of the interactive system behaviour via task-widget based sequences. The main issue with this technique was that the constructions of the task ordering can become too complex resulting in intractable models. We demonstrate how we have solved this issue using the self-containment property and abstraction.

We can construct finite state automaton P for these task orderings as depicted in figure 5.13. All transitions are labelled with the "ϵ" character to denote the empty character or rather "blank" transition. This allows us to define the automaton $P = (Q_P, \Sigma_P, \delta_P, S_P, F_P)$ (figure 5.13) as follows:

$Q_P = \{Set\ up\ an\ Infusion, Start\ an\ Infusion, Pause\ an\ Infusion, Turn\ Off\}$

$\Sigma_P = \{\epsilon\}$

$\delta_P = \{(Set\ up\ an\ Infusion, \epsilon, Turn\ Off), (Set\ up\ an\ Infusion, \epsilon, Start\ an\ Infus-ion), (Start\ an\ Infusion, \epsilon, Pause\ an\ Infusion), (Start\ an\ Infusion, \epsilon, Turn\ Off), (Pause\ an\ Infusion, \epsilon, Start\ an\ Infusion), (Pause\ an\ Infusion, \epsilon, Turn\ Off)\}$

$S_P = \{Set\ up\ an\ Infusion, Turn\ Off\}$

$F_P = \{Set\ up\ an\ Infusion, Start\ an\ Infusion, Pause\ an\ Infusion, Turn\ Off\}$

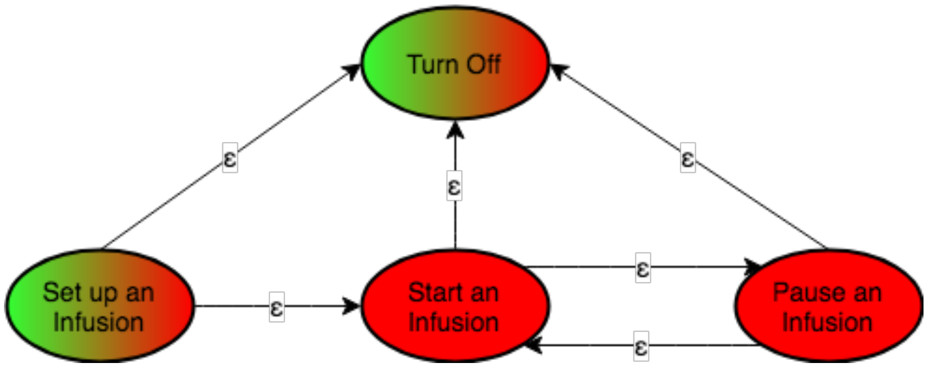


Figure 5.13: Automaton P

However, as stated previously, to use the self-containment property to ensure correct expansion we must observe definition 9. This means that we must have singleton start and final state sets for automaton P. Therefore, as per lemma 1 we can construct a new automaton $Q = (Q_Q, \Sigma_Q, \delta_Q, S_Q, F_Q)$ such that S_Q and F_Q are singleton state sets (see figure 5.14).

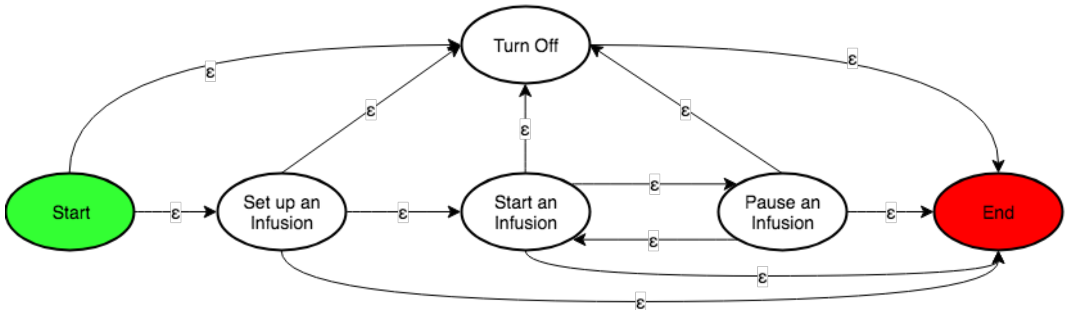


Figure 5.14: Automaton Q

We treat each state in automaton Q as an abstract state (with the exception of the singleton start and final sets, as these are a result of construction). Whether a state is reachable from the start state or the final state is reachable from a state is dependent on the task ordering. In the Alaris GP Pump, an end user may start with either the set up an infusion or turn off tasks, however, they cannot start or pause an infusion until an infusion has been set up correctly and started. This is why the start an infusion and pause an infusion tasks are not reachable from state “Start”. However, an end user may end at any point in the tasks, therefore the final state is reachable from every abstract state.

Firstly, we begin with expanding the “Set up an Infusion” state to demonstrate how the abstraction works. We refer to the automaton $M_{setup} = (Q_{setup}, \Sigma_{setup}, \delta_{setup}, S_{setup}, F_{setup})$ as depicted in figure 4.9. Following definition 8 we simply rewrite the transitions and sets to construct automaton $R = (Q_R, \Sigma_R, \delta_R, S_R, F_R)$ (figure 5.15) as follows:

- $Q_R = (Q_Q \setminus \{Set\ up\ an\ Infusion\}) \cup Q_{setup}$, therefore:

  $Q_R = \{Start\ an\ Infusion, Pause\ an\ Infusion, Turn\ Off, InitSetup, OnOff, Alarm, AlarmLight, DisplaySetup, Button1, Button3, Down, Button2, Up,$

$\mathit{Start}, \mathit{End}\}$;

- $\Sigma_R = \Sigma_P \cup \Sigma_{setup}$, therefore: $\Sigma_R = \{\epsilon, \mathit{Press}, \mathit{Observe}\}$.

- $\delta_R = \delta_{setup} \bigcup_{s \in S_{setup}, f \in F_{setup}} (_f^{\{Set\ up\ an\ Infusion\}}(\delta_P)_s^{\{Set\ up\ an\ Infusion\}})$, therefore: $\delta_R = \{(\mathit{Button1}, \epsilon, \mathit{Turn\ Off}), (\mathit{Button1}, \epsilon, \mathit{Start\ an\ Infusion}), (\mathit{Start\ an\ Infusion}, \epsilon, \mathit{Pause\ an\ Infusion}), (\mathit{Start\ an\ Infusion}, \epsilon, \mathit{Turn\ Off}), (\mathit{Pause\ an\ Infusion}, \epsilon, \mathit{Start\ an\ Infusion}), (\mathit{Pause\ an\ Infusion}, \epsilon, \mathit{Turn\ Off}), (\mathit{InitSetup}, \mathit{Press}, \mathit{OnOff}), (\mathit{OnOff}, \mathit{Observe}, \mathit{Alarm}), (\mathit{Alarm}, \mathit{Observe}, \mathit{AlarmLight}), (\mathit{AlarmLight}, \mathit{Observe}, \mathit{AlarmLight}), (\mathit{AlarmLight}, \mathit{Observe}, \mathit{DisplaySetup}), (\mathit{DisplaySetup}, \mathit{Press}, \mathit{Button1}), (\mathit{Button1}, \mathit{Press}, \mathit{Button3}), (\mathit{Button3}, \mathit{Press}, \mathit{Down}), (\mathit{Down}, \mathit{Press}, \mathit{Down}), (\mathit{Down}, \mathit{Press}, \mathit{Button1}), (\mathit{Button1}, \mathit{Press}, \mathit{Button1}), (\mathit{Button1}, \mathit{Press}, \mathit{Down}), (\mathit{Button1}, \mathit{Press}, \mathit{Button2}), (\mathit{Button2}, \mathit{Press}, \mathit{Button2}), (\mathit{Button2}, \mathit{Press}, \mathit{Up}), (\mathit{Up}, \mathit{Press}, \mathit{Button1}), (\mathit{Button1}, \mathit{Press}, \mathit{Up}), (\mathit{Start}, \epsilon, \mathit{InitSetup}), (\mathit{Start}, \epsilon, \mathit{Turn\ Off}), (\mathit{Button1}, \epsilon, \mathit{End}), (\mathit{Start\ an\ Infusion}, \epsilon, \mathit{End}), (\mathit{Pause\ an\ Infusion}, \epsilon, \mathit{End}), (\mathit{Turn\ Off}, \epsilon, \mathit{End})\}$.

- *If* S_P *contains only Set up an Infusion then* S_R *contains only* s. *Otherwise* $S_R = S_P$, therefore: $S_R = \{\mathit{Start}\}$.

- Similarly, $F_R = \{\mathit{End}\}$.

The benefit of using the self-containment property in this way over FSA constructions is that we can expand and abstract states as required. Therefore, we do not require a complete model to investigate certain areas of the interactive system via interaction sequences, that is we can construct the automata and abstract as necessary. If every state is expanded this automaton R will be equivalent to automaton F, therefore as stated previously this technique does not reduce the state space but provides control over the state space as required.

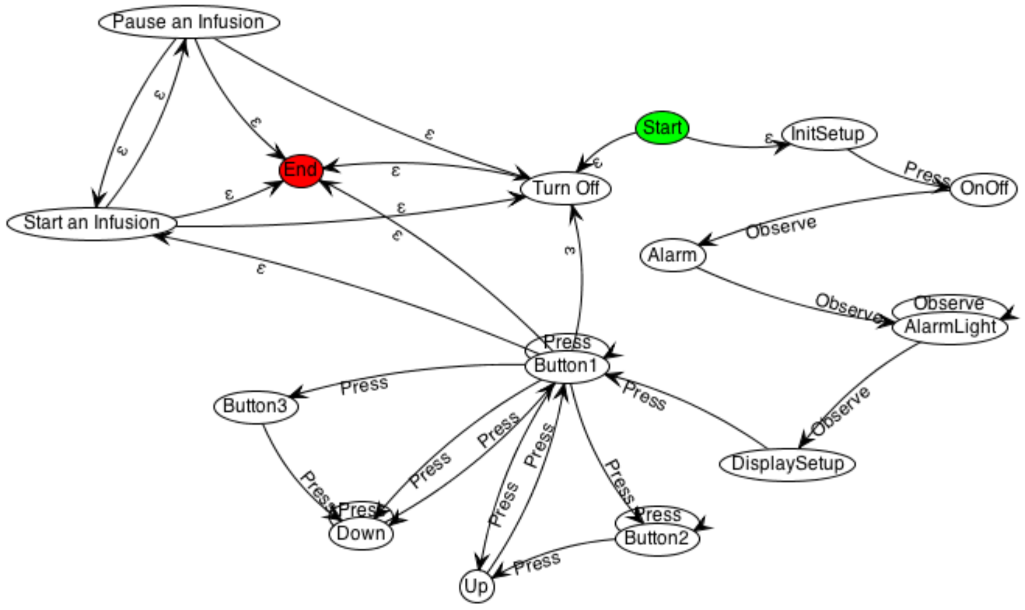


Figure 5.15: Automaton R

This is particularly useful for large complex interactive systems as self-containment allows us to focus on certain parts of the interactive system only. In particular, the parts of the interactive system that we wish to test. Therefore, the self-containment property does not explicitly "solve" the state space explosion problem, but provides control over the state space, in order to keep the model tractable.

Note that as discussed in chapter 2 Task Models such as CTT explore a similar abstraction. The point of difference is that in these techniques tasks start at a higher level and are decomposed into smaller and smaller components. Here we take advantage of the pre-existing groupings within the system and can either decompose similarly or begin from a low level and simplify into larger components.

5.4.3 Self-Containment and Strongly Connected

In graph theory a strongly connected component is defined as: "a directed graph

G where for each pair of vertices v, w in G a path exists from v to w and from w to v"; [65]. White and Almezen demonstrate how strongly connected components can be used to construct *sub-automata* (sub-automata are automata within a given automaton) from an automaton [79]. Similarly to using the self-containment property, they demonstrate how a new automaton can be constructed with the strongly connected components abstracted.

In this section we define the *strongly connected property* (definition 10 for well-formed FSA as per definition 1). We demonstrate that self-contained FSA which are strongly connected are simply one subset of the self-contained automata we can construct. That is, the self-containment property includes the strongly connected property and allows us to construct a larger set of abstract automaton when compared with the strongly connected property. This highlights that the self-containment property allows us to abstract FSA in a wider variety of ways, providing more flexibility.

Definition 10 *Let $M = (Q, \Sigma, \delta, S, F)$ be a well-formed finite state automaton as per definition 1. If for each pair of states $q, q' \in Q$, q' is reachable from q and q is reachable from q', we can say that the automaton M is strongly connected.*

Lemma 4 *As per lemma 2 every automaton is self-contained with respect to itself, therefore it follows that all strongly connected automata are also self-contained automata. Conversely, a self-contained automaton is not necessarily strongly connected. For example, let $M_C = (Q_C, \Sigma_C, \delta_C, S_C, F_C)$ be a self-contained finite state automaton with respect to itself. Let states $a, b \in Q_C$ and by definition 2 let state a be reachable from state b but b not reachable from state a. By definition 10, all states must be reachable from every other state, therefore automaton M_C is not strongly connected but self-contained (with respect to itself).*

The self-containment property includes, and allows us to abstract, self-contained automata which are not strongly connected as well as those that are. For example $H = (Q_H, \Sigma_H, \delta_H, S_H, F_H)$ (figure 5.16) is an automaton as follows:

$Q_H = \{1, 2, 3, 4, 5\}$

$\Sigma_H = \{a\}$

$\delta_H = \{(1, a, 2), (2, a, 3), (3, a, 4), (4, a, 5), (5, a, 1)\}$

$S_H = \{1\}$

$F_H = \{5\}$

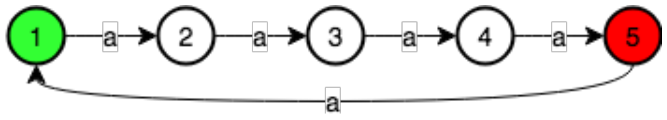


Figure 5.16: Automaton H

We construct self-contained automaton $I = (Q_I, \Sigma_I, \delta_I, S_I, F_I)$ (figure 5.17), with respect to H, which is not strongly connected as follows:

$Q_I = \{2, 3, 4\}$

$\Sigma_I = \{a\}$

$\delta_I = \{(2, a, 3), (3, a, 4)\}$

$S_I = \{2\}$

$F_I = \{4\}$

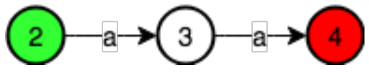


Figure 5.17: Automaton I

Automaton I satisfies the self-containment property as:

- $Q_I \subset Q_H, \Sigma_I \subset \Sigma_H$, and $\delta_I \subset \delta_H$.

- For each transition (q, x, q'), $q, q' \in Q_I$ and $x \in \Sigma_I$.

- Start state “2” has one incoming transition from H: “$(1, a, 2)$”.
- Final state “4” has one outgoing transition to H: “$(4, a, 5)$”.

However, this does not satisfy the strongly connected property as:

- While states “3” and “4” are reachable from state “2”, state “2” is not reachable from states “3” and “4”.
- While state “4” is reachable from state “3”, state “3” is not reachable from state “4”.

Therefore, H is self-contained but not strongly connected.

We can construct a self-contained automaton J (with respect to H) which is strongly connected, simply $J = H$, that is H is self-contained with respect to itself (as per lemma 2). This allows us to abstract a larger number of self-contained automata when compared to abstracting only on the strongly connected property. This allows for more flexibility in these abstractions in addition to allowing us to focus on different parts of an interaction sequence.

5.4.4 Limitations

While there are many benefits to using the self-containment property, the main benefit being the ability to control the state space, we acknowledge that there are some limitations. The self-containment property provides control over the state space but does not solve the state space explosion problem. This is because it is impossible to prove that every automaton will always have a tractable number of states or self-contained components. However, as we have demonstrated, we have provided enough control over the state space that an entire state space could be explored piece by piece (such as in task ordering).

It is possible to have an intractable model which does not have the self-containment property, resulting in no abstraction (beyond the trivial case of abstracting to a single state or each single state to an abstract state). In this instance we would not be able to abstract further using this method. It is possible that this could occur in highly connected systems.

Different types of systems have different types of connectedness, for example a wizard type system has low connectedness as it prompts end users to follow a specific path. In contrast, a calculator system has a high level of connectedness as every button is available at any time for the end user to push. In a highly connected system it is possible that the self-containment property is not applicable. Further investigation into this issue is required.

We can generate the sub-automata from interaction sequences either beforehand or after we have constructed the abstract automaton. Therefore, the question can be asked, do we store this information or re-generate the model using the sequence should we need to revisit it? We have addressed this limitation with the inclusion of assumptions to ensure that each sequence is reproducible.

While we can detect the self-containment property and construct the abstract automaton automatically, we cannot detect if this abstraction will be useful or not (in terms of adapting the sequences for testing purposes). Keeping in mind that we can abstract an entire automaton to a single abstract state, we leave it to human reasoning to determine if abstracting a self-contained automaton provides benefits or not. Therefore, our approach is semi-automated. However, we do not see this semi-automation as a limitation of our work as human reasoning is always required to create meaningful test suites (that is, we cannot fully automate this process).

5.5 Self-Containment and the Alaris GP Pump

In this section we use the Alaris GP Pump to demonstrate the self-containment property using the model for the "Set VTBI over time" task. The interaction sequence assumptions are as follows:

Start Assumptions:
State: RateOnHold
Profile: Pediatrics
Drug category: ABCDE
Drug: Dobutamine
Rate: 100
VTBI: 1000
VTBI Bag Size: 1000
End Rate: Stop
Battery status: Charging
Bolus: Hands On Only
Dose Rate Soft Min: 1
Dose Rate Soft Max: 61
Dose Rate Hard Max: 100
Infusing: No

End Assumptions:
State: RateOnHold
Profile: Pediatrics
Drug category: ABCDE
Drug: Dobutamine
Rate: 3000
VTBI: 3000
VTBI Bag Size: 3000
End Rate: Stop
Battery status: Charging
Bolus: Disabled
Dose Rate Soft Min: 1
Dose Rate Soft Max: 61
Dose Rate Hard Max: 100
Infusing: No

Using these assumptions we build the interaction sequence as follows:

1. Press Options 1.
2. Press DownButton 2.
3. Press Button1 1.
4. Press Button2 1.
5. Press DoubleDown 1.
6. Press Up 5.

7. Press Button1 2.

8. Press Up 10.

9. Press Button1 1.

10. Press Up 1.

11. Press Button1 1.

12. Observe Display.

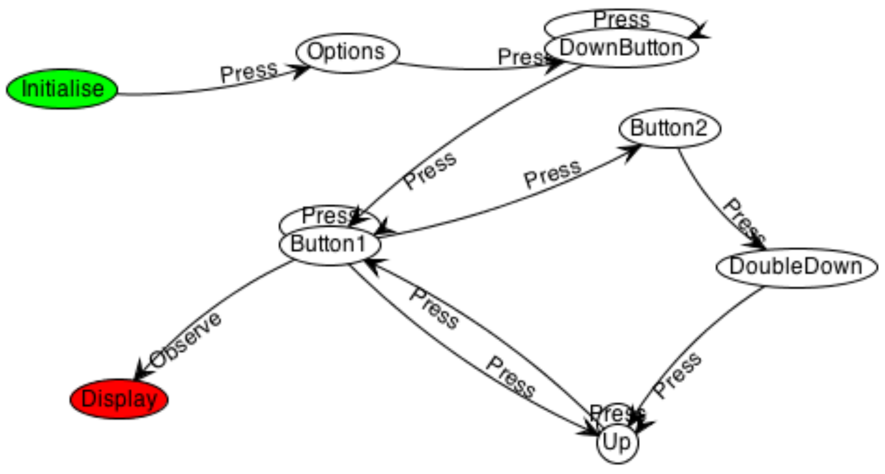


Figure 5.18: Automaton $M_{VTBI/Time}$

This allows us to construct the following automaton

$M_{VTBI/Time} = (Q_{VTBI/Time}, \Sigma_{VTBI/Time}, \delta_{VTBI/Time}, S_{VTBI/Time}, F_{VTBI/Time})$ (figure 5.18):

$Q_{VTBI/Time} = \{Initialise, Options, DownButton, Button1, Button2, Double Down, Up, Display\}$

$\Sigma_{VTBI/Time} = \{Press, Observe\}$

$\delta_{VTBI/Time} = \{(Initialise, Press, Options), (Options, Press, DownButton), (DownButton, Press, DownButton), (DownButton, Press, Button1), (Button1, Press, Button2), (Button2, Press, DoubleDown), (DoubleDown, Press, Up), (Up, Press, Up), (Up, Press, Button1), (Button1, Press, Button1), (Button1, Press, Up), (Button1, Observe, Display)\}$

$S_{VTBI/Time} = \{Initialise\}$

$F_{VTBI/Time} = \{Display\}$

Using definition 4 we can automatically detect 20 self-contained automata with respect to automaton $M_{VTBI/Time}$. They are as follows:

- The trivial self-contained automata: $Display$, Up, $DoubleDown$, $Button2$, $Button1$, $DownButton$, $Options$ and $Initialise$.
- The self-contained with respect to itself automaton as per lemma 2: $Initialise$, $Options$, $DownButton$, $Button1$, $Button2$, $DoubleDown$, Up, $Display$.
- $Button2$, $DoubleDown$.
- $Button1$, $Button2$, $DoubleDown$, Up.
- $Button1$, $Button2$, $DoubleDown$, Up, $Display$.
- $DownButton$, $Button1$, $Button2$, $DoubleDown$, Up.
- $DownButton$, $Button1$, $Button2$, $DoubleDown$, Up, $Display$.
- $Options$, $DownButton$.
- $Options$, $DownButton$, $Button1$, $Button2$, $DoubleDown$, Up.
- $Options$, $DownButton$, $Button1$, $Button2$, $DoubleDown$, Up, $Display$.
- $Initialise$, $Options$, $DownButton$.
- $Initialise$, $Options$.
- $Initialise$, $Options$, $DownButton$, $Button1$, $Button2$, $DoubleDown$, Up.

Figure 5.20: Automaton $M_{complete}$

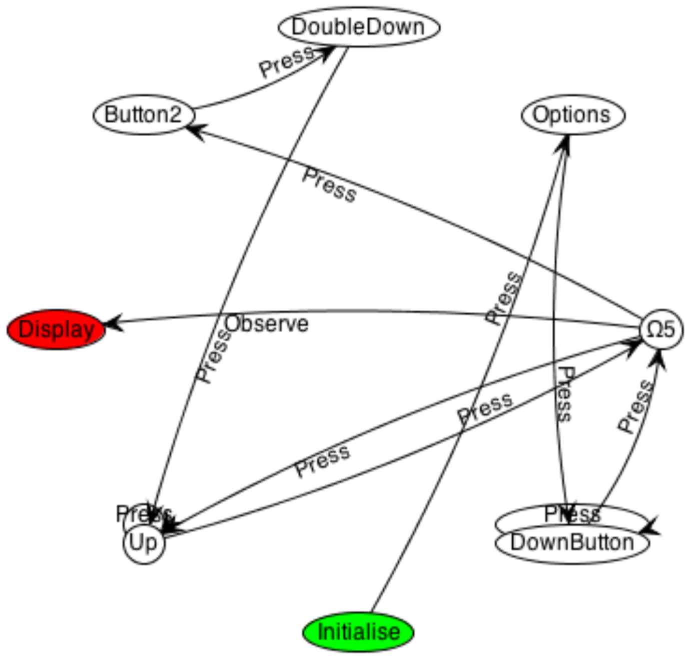


Figure 5.19: Automaton $M_{Button1_S}$

Next we demonstrate a single state abstracted using definition 7. We abstract the "Button1" state into abstract state "Ω5" constructing automaton $M_{Button1_S}$ as shown in figure 5.19. Note that automaton $M_{Button1_S}$ is equivalent to automaton $M_{VTBI/Time}$.This abstraction does not result in a reduction of the state space, and as such is not useful for this purpose.

Next we demonstrate the entire state space abstracted to a single state (again using definition 7). Again in this context this is not useful as we hide all interesting information about the interaction sequence. However, as shown in the task ordering section this abstraction may prove useful in forward modelling approaches. See figure 5.20 which depicts $M_{complete}$.

Lastly, we demonstrate a "useful" abstraction using definition 7. In this

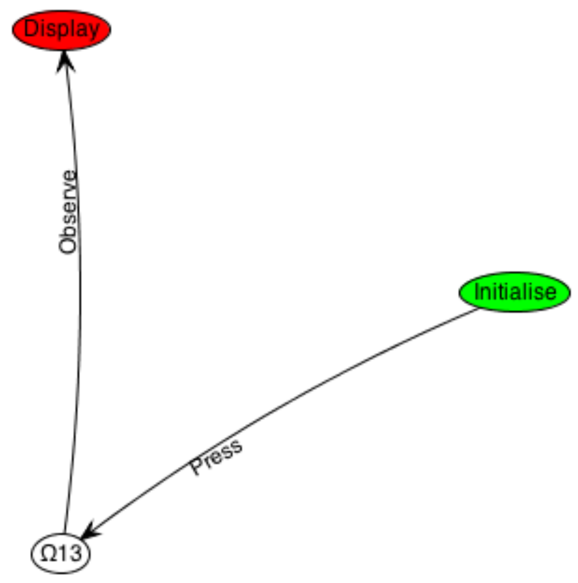


Figure 5.21: Automaton $M_{VTBI/TIME_S}$

case we select states "$Options$, $DownButton$, $Button1$, $Button2$, $DoubleDown$, Up" as these specifically change and confirm the values related to the task "Set VTBI Over Time". The automaton $M_{VTBI/TIME_S}$ is depicted in figure 5.21. We can expand the state "Ω13" simply by using definition 8. This would result in an automaton which is equivalent, in that it is exactly the same, as automaton $M_{VTBI/Time}$.

5.6 Summary

In this chapter we discussed constraining interaction sequences in particular with reference to the state space of FSA. We first discussed issues with existing techniques in FSA theory, specifically how the grouping of states introduces ambiguity and unnecessary complexity to the models. We described how this prompted us to investigate other ways to constrain the state space. This led to the development of the self-containment property.

We followed this with the formal definitions of the self-containment property (definition 4); abstraction (definition 7); and expansion of these automata (definition 8) in addition to supporting definitions. We demonstrated that these definitions had useful properties and had captured our intuitions. Specifically, our main result showed that we could reduce and expand the state space of an automaton as required using the self-containment property.

We demonstrated how we could use the self-containment property to constrain the state space, beginning with a simple demonstration of reducing and expanding an automaton. This was followed by a discussion on task ordering and how the self-containment property could be used as an alternative to FSA constructions. Lastly, we introduced the strongly connected property (definition 10) and showed how the self-containment property allows for a larger set of abstract automata when compared with strongly connected automata.

We gave a short example of the self-containment property using an example with the Alaris GP Pump, specifically for the task of setting up a VTBI value over time. This demonstrated that even for a small model there can be several self-contained automata. We specifically discussed the two cases of abstraction considered not useful (abstraction to a single abstract state and a single state abstraction) and a useful abstraction using the self-containment property.

Chapter 6

Simulating Interaction Sequences

6.1 Introduction

In this chapter we explore simulating interaction sequences using FSA as defined in chapter 4. We begin with a discussion on model checking and a description of the models required for simulation. This is followed by the introduction of the sequence simulator tool. Interaction sequence simulation is demonstrated using models of the Alaris GP Pump. In particular we demonstrate the self-containment property along with supporting functions. Finally, we finish with a discussion on the benefits and limitations of this approach.

6.2 Model Checking vs. Testing

As stated by Sakib *et al.* in [61], "*Model checking* is a formal method for automatic verification of software systems, which offers distinct advantages over conventional testing and simulation techniques". Model checking can be used in a variety of different ways to prove that certain properties hold true for a given model [25, 27]. Several different techniques have been developed for model checking interactive systems [9, 6, 45, 12]. In particular, these techniques focus on safety-critical interactive systems in order to prove that certain safety and liveness properties (as discussed in chapter 2) are apparent within the models.

As interaction sequences are models of the overlap component of an interactive system we can take advantage of model checking techniques. For example, we could use the FSA to demonstrate that deadlock does not exist in the model, deadlock refers to a state in which no further progress can be made. This situation can be inherently important in safety-critical situations, particularly if deadlock causes harm to end users [6].

The models of the interaction sequences as FSA allow for exploration of sequences of varying lengths for specific tasks. This variation could be useful in a model checking approach to ensure that properties hold despite variation. It would be interesting to see the effect this variation would have on different properties such as safety or liveness (as defined in chapter 2).

Using interaction sequence models for the purposes of model checking is as feasible as testing using interaction sequences. In the work presented in this chapter we demonstrate interaction sequence simulation which is a precursor step to performing either model checking or testing using the interaction sequences. This is because it allows for exploration of the interactive sequence models which provides us with an easy way to visually check that the start and end assumptions are correct. However, while this simulation allows for understanding and exploration of the models, it is not a rigorous or necessarily formal process when compared with testing and model checking, as we simply allow the end user of the tool to explore sequences as they choose.

In chapter 1 we state that one of the main issues with testing of interactive systems is that the overlap component is often not part of the testing process and one of the contributions of this thesis is to present a solution to address this issue. For this reason we focus on a testing approach using interaction sequences in this thesis rather than model-checking. However we are aware that model checking is another possible use of interaction sequences and is an option for future work by extending the interaction sequence simulator tool described here.

6.3 Modelling the Different Components of an Interactive System

In order to simulate each component of an interactive system we must have appropriate models for each of the components. That is, we must have a suitable model which describes the behaviour for each of the interactive, functional, and overlap components. In this section we describe the models used for each of the different components. In addition we describe how we combine these in order to simulate the interaction sequences. We also describe how these models are implemented as part of our proof-of-concept tool.

The user interface represents the interactive component of an interactive system, specifically the way in which a user can interact with the underlying functionality of a system. As defined previously in chapter 2, PModels describe the elements of user interfaces (widgets) grouped within modes or windows of the system.

In the PModel each widget is described by a triple which associates a widget name and category to specific behaviours in the following form: "$(WidgetName, WidgetCategory, (Behaviour(s)))$". There are two types of behaviours, $I\_Beh-aviour$ and $S\_Behaviour$. An $I\_Behaviour$ describes an interface behaviour, while an *S_Behaviour* describes a system behaviour.

As modes and widgets of the interactive system are described using PModels, this maps directly to the interaction sequence models. There is a one-to-one relationship between the widgets in the PModels and widgets used in the interaction sequence models steps (as discussed in chapters 3 and 4). In fact as discussed PModels can be used to generate task-widget based interaction sequences with appropriate task knowledge. Using PModels to describe the interactive component is therefore the obvious choice for simulation.

However, the PModels do not define the *I_Behaviours* and *S_Behaviours* formally by themselves, that is these behaviours are labelled but not defined in the PModels. Therefore, these behaviours require further modelling, we achieve this in the form of a PIM and PMR, where the PIM describes the transitions between different modes using *I_Behaviours* and the PMR relates the *S_Behaviours* to consequent operations from a formal specification.

The PIM describes the transitions between the different modes of the PModel using *I_Behaviours*, that is it gives meaning to the *I_Behaviours*. Each transition specifies the state the system is currently in (where a state represents the Component Presentation Model (CPModel) used to describe a mode or window), the associated *I_Behaviour*, and the state the system transitions to. This can be used to create a state transition chart. The PMR relates the *S_Behaviours* to the corresponding operations described formally within a specification.

The functional component can be modelled through the use of a formal specification, in this case we chose to use the Z language (as described in chapter 2). This allows us to define exactly the expected behaviour of each function. The Z specification describes all possible operations of the SUT in terms of changes to observations of the state space. As Z is not an operational language, we made use of the ProB Tool[1].

The ProB tool was created to allow the simulation of formal specifications written in the B language, in addition to other uses. Over time this tool has been extended to include several different specification languages, such as the Z language. Using this tool we can perform model checking in addition to exploring the state space animation of specifications. Of interest here is the ability to simulate the specifications and resulting changes to the observations.

The ProZ plugin of the ProB tool allows us to convert a Z specification to a B specification for simulation. The ProB2 library[2] allows us to load this specification and simulate a trace from this specification, it also enables animation of the model. Therefore, using the PMR and the associated B specification, in addition to the PModels and PIM, we have models of the functional and interactive components necessary for simulation.

Note that these models are not necessarily complete in that they may not describe every aspect of the system. However, as long as they are complete enough (include all of the details needed) for the interaction sequences we are modelling they are sufficient for simulation purposes. Furthermore, there may be parts of the system not modelled that are not related to the interactive system, such as parts of the hardware (for example we do not model the physical battery of the Alaris GP Pump and its associated behaviour). We use the defined tasks to constrain the PModels, PIM, PMR, and Z models that we create of the interactive system, similarly to the interaction sequences.

The process for simulation is as follows: a step of the interaction sequence is chosen, from this we can get the associated widget name. The mode the system is in is retrieved from the PIM, we use this mode to select the appropriate PModel. Using the PModel we can trigger the behaviours for the

[1]See `https://www3.hhu.de/stups/prob/index.php/Main_Page`.

[2]See `https://github.com/bendisposto/prob2`.

chosen widget, where *I_Behaviours* are passed to the PIM and *S_Behaviours* are passed to the PMR. The PIM allows us to change the mode of the system as specified, while the PMR allows us to retrieve the associated schema in the B specification. Using the ProB2 library we can simulate the *S_Behaviour* functionality, and the observations are updated accordingly.

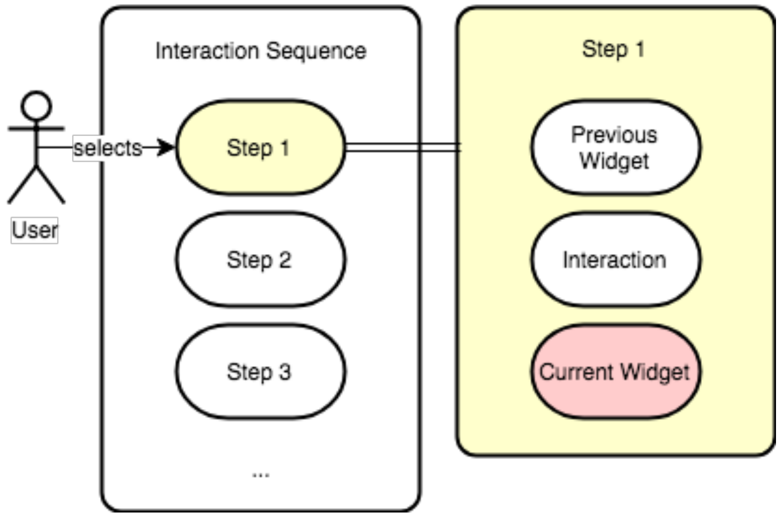


Figure 6.1: Select a Step from an Interaction Sequence

We now describe the different parts of the simulation and the information retrieved from the various models necessary for this simulation. In figure 6.1 the end user selects a step from the interaction sequence for simulation. The information we get from this selection is the current widget and therefore its associated behaviours the end user would like to simulate. Note that the previous widget is either the last widget interacted with or the place-holder state "Initialise" for the first step.

In figure 6.2 we depict the process of getting the behaviours for the associated selected widget. To achieve this we input the current widget selected from the interaction sequence and the current mode of the simulation and select the appropriate CPModel from the set of PModels. For this CPModel we select the triple which has the same widget name as the current widget. In the

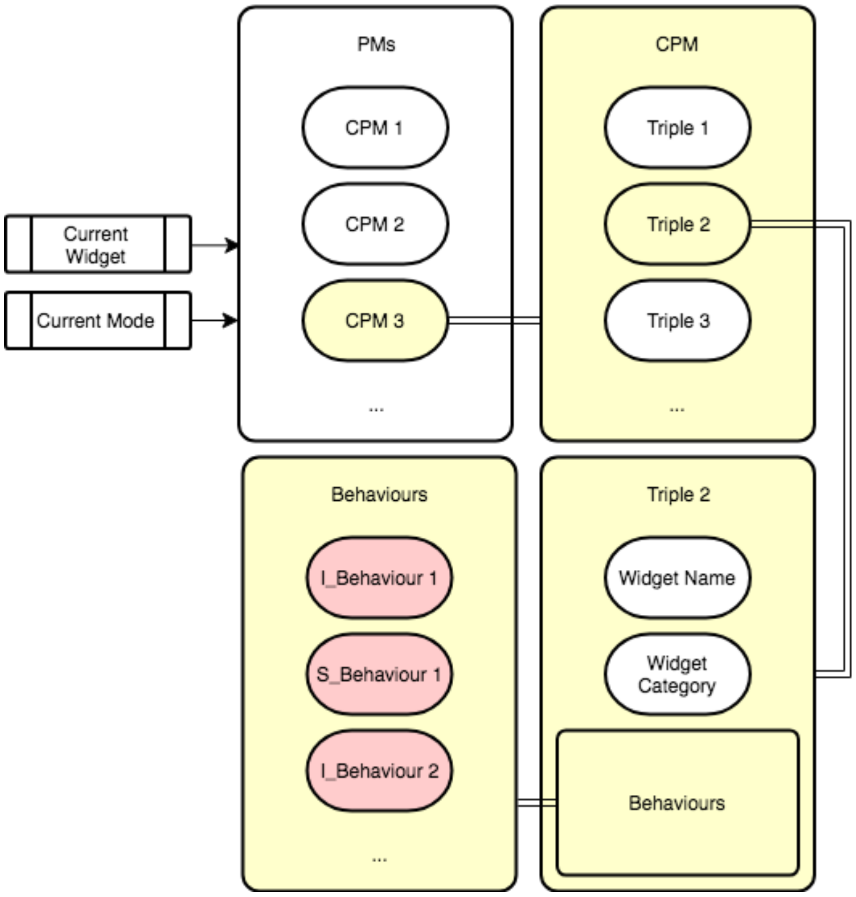


Figure 6.2: Select Behaviour(s) for the Current Widget

triple, the behaviours of the widget for this mode are stored, we select these behaviours for simulation.

As stated previously, *I_Behaviours* are passed to the PIM which take the current mode of the simulation and the *I_Behaviour* as inputs as seen in figure 6.3. Using this information the appropriate transition which has a matching current mode and *I_Behaviour* is selected. This transition is then simulated to change the current mode of the simulation.

There is a two step process to simulate the *S_Behaviours* triggered from the step selection. In figure 6.4 we depict the process of selecting the appropriate operation from the PMR. The PMR takes the *S_Behaviour* as an input and we select the associated relation with matching *S_Behaviour*. From this relation

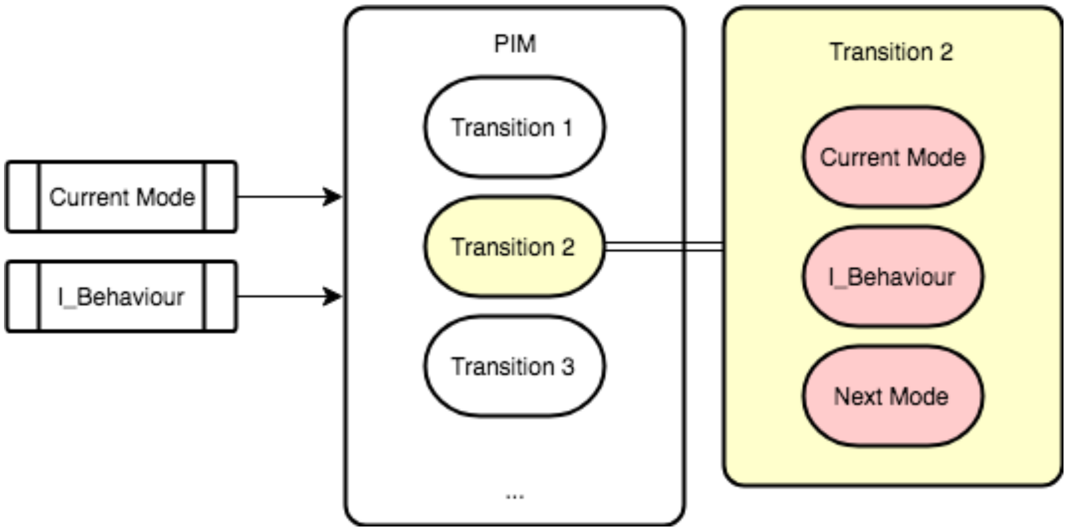


Figure 6.3: Simulating *I_Behaviours*

we can get the name of the operation to simulate for the specification.

In figure 6.5 we depict the simple process of selecting the correct operation schema from the specification. The specification takes the operation name as input, which was selected from the PMR and matches this with the appropriate operation schema. Using the ProB2 library this schema is then simulated. Once each of the behaviours has been simulated in the corresponding way the step simulation is complete.

The simulation of the interaction sequences is an informative process which helps us to better understand the interaction sequence and the modelled behaviour. Using the interaction sequence models in this way allows us to clearly observe the behaviour of the overlap component. The PModels and PIM allow us to observe the interactive component for a given step and view the instructions sent to the PMR and consequently to the B specification (which represents the functional component).

One could argue that these existing models are already used to observe the overlap component via the PModels and PMR, as the PMR maps the *S_Behaviours* to the Z specification. However, this is only one part of the over-

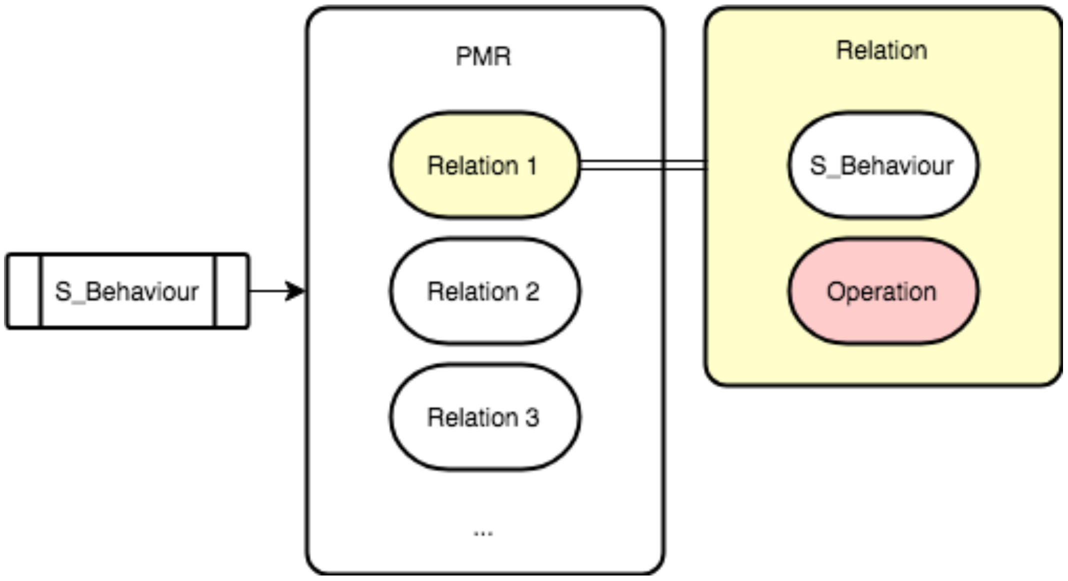


Figure 6.4: Simulating *S_Behaviours*

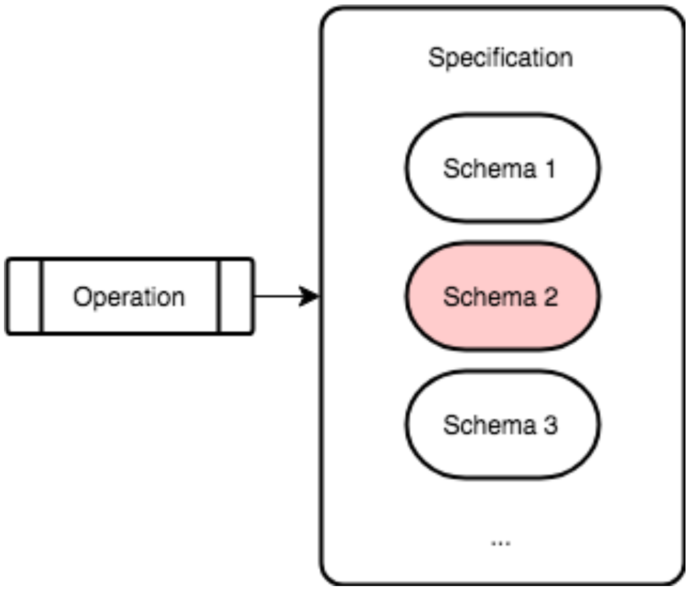


Figure 6.5: Selecting Operation Schema from A Specification

lap component. The PModel models the relationship between a widget and *S_Behaviour* while the PMR models the relationship between that behaviour and the associated specified behaviour. This is a uni directional relationship and does not include the instructions or information returned to the interactive component from the functional component. That is, the specification observations are not directly linked to the interactive component models.

Using assumptions for the interaction sequences, we can trace both parts of this relationship (albeit in an abstract manner). The start and end assumptions specify the observations we expect from the specification for a given sequence. Specifically, the start assumptions define the state the functional component should be in, while the state from the PIM defines the mode of the system. The end assumptions specify the expected changes indicating correctness of the logic as well as associated returned data. In addition to this, we can assume that the changes to the interactive and functional components have been carried out successfully, as the end assumptions are correct. This will be a particularly useful property when adapting the interaction sequences for the purposes of oracle generation in testing.

Therefore, simulation of the interaction sequences allows us to observe the bi-directional relationship between the interactive and functional components, in addition to the changes to these components for a given interaction sequence step. Note that the individual changes in these components are not considered here as several strategies exist to test the functional and interactive components separately, our focus is singularly on the overlap component. For example, we are not interested in that the interaction sequence can be simulated using a model of the interaction component, but that this in combination with the functional component achieves a desired result (the end assumptions are as expected). As interaction sequences allow us to observe this relationship directly, they provide us with a more complete view of the overlap component.

6.4 The Sequence Simulator Proof-of-Concept Tool

The Sequence Simulator tool was created as a proof-of-concept to demonstrate the ability to automatically identify the self-containment property to abstract and expand FSA as per our definitions given in chapter 5. This is a proof-of-concept tool as we have not explicitly proved that the functions implemented match the definitions, however, while it would be possible to demonstrate this, it is out of the scope of this research. The tool is used to demonstrate the applicability of our approach and show one possible way in which it might be implemented for development use.

In addition to demonstrating the self-containment property and interaction sequence simulation we added other useful features to this tool, including the ability to automatically convert a formalised interaction sequence to a finite state automaton, modify automata, convert and display automata as directed graphs, input and generate interaction sequences. These features enhance the tool functionality by providing options to modify the model and generate sequences of varying lengths.

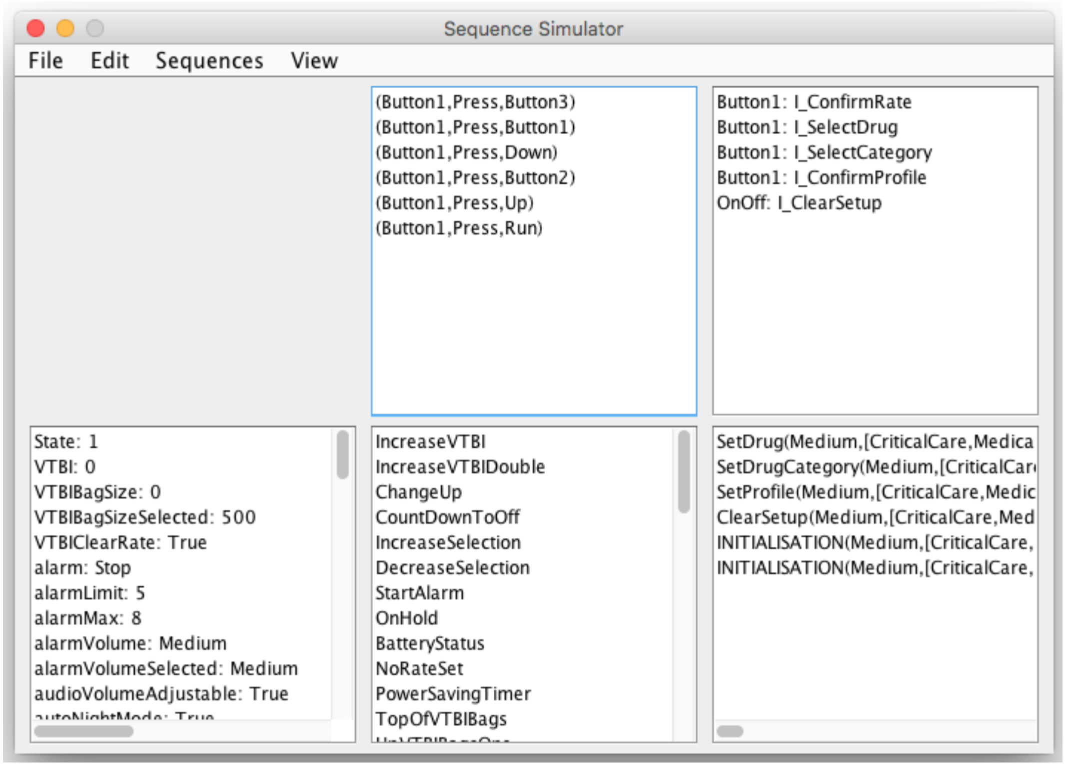
To build this tool we required access to the ProB2 library in order to simulate the functional component as a B specification. Therefore, we made use of the ProB2 Tooling Template[3] to allow us to easily access the ProB2 library for sequence simulation. This tool is available as a Java Library and so we chose to use the Java programming language and Java Swing to create our tool. In addition to this we used the JUNG library[4] which allowed for easy

[3]See `https://github.com/bendisposto/prob2_tooling_template`.

[4]See `http://jung.sourceforge.net/`.

creation of directed graphs in Java using existing functions.

The first step in using the tool is to load each of the required models: a specification; interaction sequence model; PModels; PIM; and PMR. The tool processes each of these models so that they may be used for simulation in addition to functions described above. Note that when we refer to the 'end user' in this context we mean the end user of the tool, rather than the end user of the SUT.



```
(Button1,Press,Button3)
(Button1,Press,Button1)
(Button1,Press,Down)
(Button1,Press,Button2)
(Button1,Press,Up)
(Button1,Press,Run)
```

```
Button1: I_ConfirmRate
Button1: I_SelectDrug
Button1: I_SelectCategory
Button1: I_ConfirmProfile
OnOff: I_ClearSetup
```

```
State: 1
VTBI: 0
VTBIBagSize: 0
VTBIBagSizeSelected: 500
VTBIClearRate: True
alarm: Stop
alarmLimit: 5
alarmMax: 8
alarmVolume: Medium
alarmVolumeSelected: Medium
audioVolumeAdjustable: True
```

```
IncreaseVTBI
IncreaseVTBIDouble
ChangeUp
CountDownToOff
IncreaseSelection
DecreaseSelection
StartAlarm
OnHold
BatteryStatus
NoRateSet
PowerSavingTimer
TopOfVTBIBags
```

```
SetDrug(Medium,[CriticalCare,Medica
SetDrugCategory(Medium,[CriticalCar
SetProfile(Medium,[CriticalCare,Medic
ClearSetup(Medium,[CriticalCare,Med
INITIALISATION(Medium,[CriticalCare,
INITIALISATION(Medium,[CriticalCare,
```

Figure 6.6: The Sequence Simulator

Once the models are loaded the tool displays elements from each of the models as shown in figure 6.6. In the sequence simulator an end user may select the steps of the interaction sequence from the loaded interaction sequence model and the tool will respond by simulating the responses of the interactive system using the appropriate models. In figure 6.6 we can see the modes the PIM has transitioned through, the observation changes from the specification, available behaviours from the functional component, and a history of the functions simulated. For a more in-depth description of the sequence simulator tool see [74].

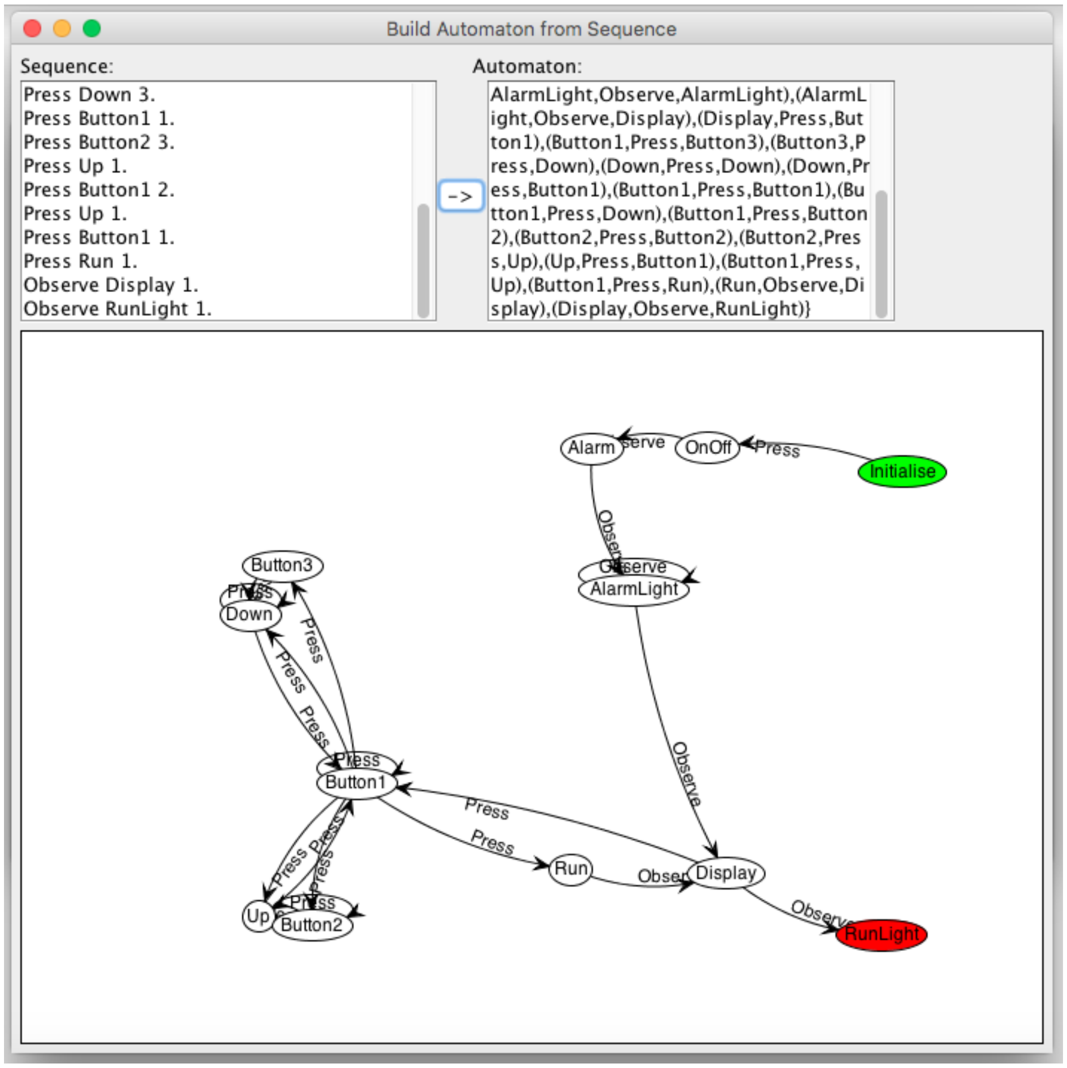
6.4.1 Building FSA from Interaction Sequences in the Sequence Simulator

Figure 6.7 depicts the sequence converter which allows a user to input a formalised interaction sequence and automatically convert this to a finite state automaton. The user may input the formalised sequence either by typing it in or pasting it from some other source into the appropriate text area. After conversion the automaton is displayed in the automaton window and visualised as a directed graph using the JUNG library.

In addition to automatically converting a formalised interaction sequence to a finite state automaton, a user may build a new automaton, modify an existing one and load or save an automaton. The user may modify the states, alphabet, transitions, start and final state sets of the automaton.

When generating interaction sequences we often start with the most direct sequence (as described previously in chapter 3), that is based on the question "what is the shortest sequence possible to finish with the correct end assumptions?" The ability to modify automaton allows us to add or remove widgets in order to explore more variations of sequences for the same task.

For example, in a system which allows a user to input positive numeric values using down and up buttons, a direct sequence would assume that an end user will not make mistakes when inputting these values. Therefore, only the 'up' button would need to be included in the automaton, as it is assumed the user will not exceed the value to be input and require the 'down' button. However, this is not a true reflection of typical user behaviour, as we know this


Build Automaton from Sequence

Sequence:

```
Press Down 3.
Press Button1 1.
Press Button2 3.
Press Up 1.
Press Button1 2.
Press Up 1.
Press Button1 1.
Press Run 1.
Observe Display 1.
Observe RunLight 1.
```

Automaton:

```
AlarmLight,Observe,AlarmLight),(AlarmLight,Observe,Display),(Display,Press,Button1),(Button1,Press,Button3),(Button3,Press,Down),(Down,Press,Down),(Down,Press,Button1),(Button1,Press,Button1),(Button1,Press,Down),(Button1,Press,Button2),(Button2,Press,Button2),(Button2,Press,Up),(Up,Press,Button1),(Button1,Press,Up),(Button1,Press,Run),(Run,Observe,Display),(Display,Observe,RunLight)}
```

Figure 6.7: Convert a Sequence to an Automaton

behaviour is quite likely (hence the inclusion of the down button). Therefore, we can modify the automaton to include this widget and explore less direct sequences which may capture this type of user behaviour.

6.4.2 Interaction Sequence Generation and Simulation

The sequence simulator also allows an end user of the tool to input specific interaction sequences for simulation. Each step of an interaction sequence may be selected and simulated within the system. A history of simulated sequences is stored and can be saved for future reference. These sequences will become important for testing purposes as we will use these to help generate abstract tests.

In addition to inputting specific sequences, an end user may generate a random sequence. These random sequences are created from the set of next possible steps based on the interaction sequence model by randomly selecting the next step from this set. The randomly generated sequences are also added to the history of simulated sequences.

6.4.3 Self-containment Property and Interaction Sequence Simulation

The sequence simulator tool has the ability to automatically detect all the self-contained sub-automata for a given automaton, based on the definitions given in chapter 5. An end user may select from these automata and the tool automatically constructs the associated abstract automaton and displays it as a directed graph. This abstract automaton can then be used for interaction sequence simulation, and as a result the end user may now explore sequences for this abstract automaton. The abstract automaton may also be automatically expanded by selecting the abstract state to expand. Note, an end user may select more than one self-contained automaton from the list of self-contained automata, provided that the self-containment property is preserved.

6.5 Interaction Sequence Simulation

In this section we demonstrate two different sequence simulations. We begin with a simulation of a complete sequence, that is, a sequence which does not contain abstract states. The second sequence demonstrates the same task simulated with a self-contained sub-automaton abstracted.

6.5.1 Complete Sequence Simulation

In this section we demonstrate an interaction sequence simulation using the Alaris GP Pump models. The sequence we use describes the task of the end user of the Alaris GP Pump to view the pump details after an infusion has already begun. The assumptions for this sequence are as follows:

Start and End Assumptions:

State: RateInfusing

Profile: Pediatrics

Drug category: ABCDE

Drug: Dobutamine

Rate: 60

VTBI: 1000

VTBI Bag Size: 1000

End Rate: Stop

Battery status: Charging

Bolus: Hands On Only

Dose Rate Soft Min: 1

Dose Rate Soft Max: 61

Dose Rate Hard Max: 100

Infusing: Yes

As this particular sequence does not modify these values we expect the start and end assumptions to be the same. The sequence is as follows:

1. Press Options 1.
2. Press Button1PumpDetails 1.
3. Observe Display 1.
4. Press Button3RateInfusing 1.
5. Observe Display 1.

From this interaction sequence we generate the following finite state automaton $T = \{Q_T, \Sigma_T, \delta_T, S_T, F_T\}$ (using the method described in chapter 4) which is depicted as a directed graph in figure 6.8:

$Q_T = \{Options, Button1PumpDetails, Display, Button3RateInfusing,$
$Initialise\}$

$\Sigma_T = \{Press, Observe\}$

$\delta_T = \{(Initialise, Press, Options), (Options, Press, Button1PumpDetails),$
$(Button1PumpDetails, Observe, Display), (Display, Press, Button3Rate$
$Infusing), (Button3RateInfusing, Observe, Display)\}$

$S_T = \{Initialise\}$

$F_T = \{Display\}$

Note that we have included widgets in this example named "Button1Pump-Details" as opposed to "Button1" as per figure 3.2. This is an artefact of using the PModels to model the interactive component of the interactive system. That is, the PModel is a model of the Alaris GP Pump interactive system, not a direct re-implementation of that system, therefore some widgets are named differently. As long as the sequences affect the system in the same way this has no significant effect on the simulation (that is, the assumptions remain the same).

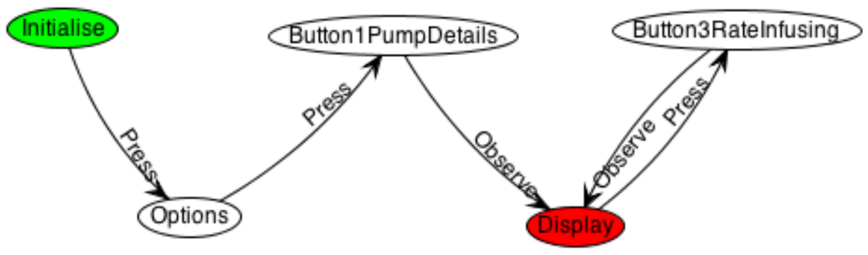


Figure 6.8: Automaton T

We use the PIMed tool[5] to create the necessary PModels, PIM, and PMR for this interactive system. This tool allows us to export these models as XML

[5]See `https://sourceforge.net/projects/pims1/`.

files for ease of loading into the Sequence Simulator tool. We have included the PModel, PIM, and PMR for the Alaris GP Pump, constrained to the interaction sequences we explore, in a repository[6].

The PModels of the Alaris GP Pump begin by defining a list of each CPModel by name. This is followed by a list of all the widgets contained in the PModels, and similarly for widget categories and behaviours. This is followed by the hierarchy of the PModels of the interactive system. Finally, each CPModel is described in the following format:

"CPModel Name is

(WidgetName, WidgetCategory, (Behaviour(s)))

...

(WidgetName, WidgetCategory, (Behaviour(s)))"

The ellipsis is used to represent extra triples for the widgets. In this example some of the widgets' behaviours are empty, denoted by an empty set of parentheses. This is because for the given mode those widgets have no behaviour. It is important to add these triples to ensure that it is clear we do not expect interaction with these widgets to trigger functions in the given mode.

The PIM of the Alaris GP Pump defines the transitions between the modes in the following format: "Current Mode $\rightarrow$ *I_Behaviour* $\rightarrow$ Next Mode". Each mode transition must be defined for each *I_Behaviour* to ensure correct interaction sequence simulation.

The PMR of the Alaris GP Pump defines the mapping of the *S_Behaviours* to the corresponding operation schema in the specification. Each mapping is denoted in the form: "*S_Behaviour* $\rightarrow$ Operation". It is essential for simulation that these mappings are correct in order to ensure the correct schema is selected from the specification when the *S_Behaviour* is triggered.

[6]See `https://github.com/jessicaturner11/AlarisModels`.

Lastly, in the repository[7] we include the operation schema for each of the *S_Behaviours* described in the PMR. Note that this is not a complete specification of the functional component for the Alaris GP Pump, as we have restricted the models to those operations that are relevant to the interaction sequences for the tasks selected. That is, we have only included the functionality relevant to the tasks described in chapter 3. For simulation we export this Z specification as a B specification for use with the ProB2 library to simulate the behaviour specified. The ProB specification is not included in the repository as it describes the same functionality as the Z specification.

Note that minor modification to this specification is required due to conversion format issues with the ProB tool. However, this is not a limitation on this approach as if the model was initially created in the B specification language this conversion would not be necessary.

Each of these models can be loaded into the sequence simulator tool in their various formats. Here we describe the automatic generation of the model from the sequence using the sequence converter to build the sequence described above. Finally, we show and describe the output we receive from the simulator.

When the PIM is loaded, the user of the sequence simulator is prompted to select a start state. This is because for different interaction sequence models we may assume a different start state for the interaction component. In this case we select the "RateInfusing" state as the start state of the PIM.

When a specification is loaded the values of the observations are not initialised. Initialisation is the only operation available in order to set up the B model with valid starting values. Once initialisation has taken place the user may begin selecting different available operations.

The user may now either load an interaction sequence model or convert a

[7]See `https://github.com/jessicaturner11/AlarisModels`.

formalised interaction sequence for simulation. Note that in order to ensure the system is in the right state according to the start assumptions a user may make changes to the observations by selecting the appropriate functions to simulate from the specification. This is to ensure that the system is in the correct state before simulating any interaction sequences using this model.

Once the appropriate models are selected the user may input sequences for simulation. The result of this for our example is as follows:

Assumptions After Simulation:
State: RateInfusing
Profile: Pediatrics
Drug category: ABCDE
Drug: Dobutamine
Rate: 60
VTBI: 1000
VTBI Bag Size: 1000
End Rate: Stop
Battery status: Charging
Bolus: Hands On Only
Dose Rate Soft Min: 1
Dose Rate Soft Max: 61
Dose Rate Hard Max: 100
Infusing: Yes

Which tells us the observations after the simulation match our end assumptions as expected. Note that this finite state automaton allows a user to generate more than just the sequence which was used to create it. We could also generate several different random sequences. This allows us to explore several different sequences for the same task, and to see what effect this has on the observations and modes. While the example given here is simplified for demonstration we describe in the following chapter how this can be useful for testing purposes.

6.5.2 An Abstract Sequence Simulation

In this section we demonstrate an abstract sequence simulation which relies on the definitions from chapter 5. We continue using the example from the previous section of a sequence for the task of "View Pump Details". The user first opens the "Abstract Sequence Model" window and is presented with a directed graph of the finite state automaton (see figure 6.9) in addition to a list of self-contained automata for that particular automaton.

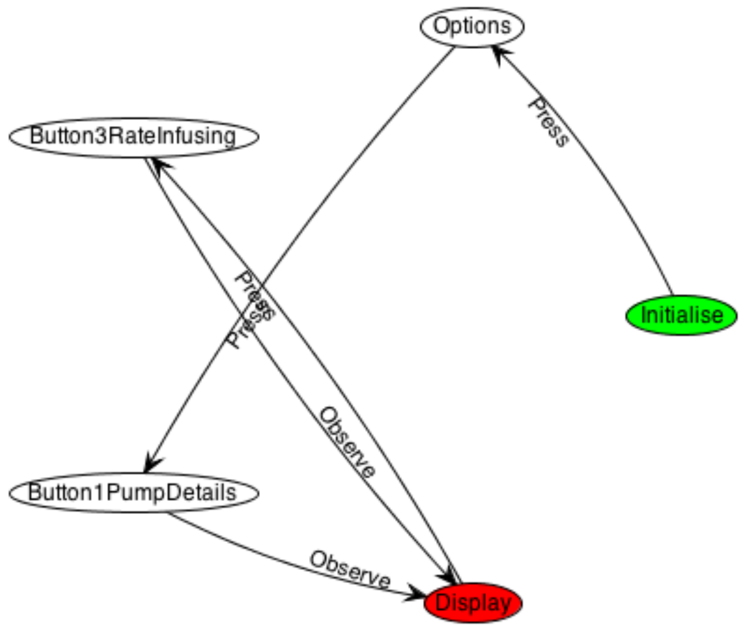


Figure 6.9: Abstract Sequence Model for View Pump Details Task

There are 12 possible self-contained FSA with respect to automaton T which are named according to the states which are self-contained within them.

- Button3RateInfusing.
- Button1PumpDetails.
- Options.
- Options, Button1PumpDetails.
- Display.
- Display, Button3RateInfusing.
- Display, Button1PumpDetails, Button3RateInfusing.
- Display, Options, Button1PumpDetails, Button3RateInfusing.
- Initialise.

- Initialise, Options.
- Initialise, Options, Button1PumpDetails.
- Initialise, Display, Options, Button1PumpDetails, Button3RateInfusing.

As stated previously in chapter 5, a user may select more than one automaton to construct an abstract automaton, provided that the states inside those automaton do not overlap (to preserve the self-containment property within the abstract automaton constructed). Figure 6.10 shows the result of selecting the "Initialise,Options" and "Display, Button1PumpDetails, Button3RateInfusing" self-contained automata. This results in a very simple abstract automaton $T_1 = (Q_{T_1}, \Sigma_{T_1}, \delta_{T_1}, S_{T_1}, F_{T_1})$: $Q_{T_1} = \{\Omega_6, \Omega_9\}$
$\Sigma_{T_1} = \{Press\}$
$\delta_{T_1} = \{(\Omega_9, Press, \Omega_6\}$
$S_{T_1} = \{\Omega_9\}$
$F_{T_1} = \{\Omega_6\}$

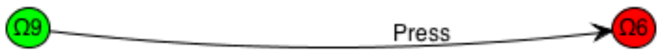


Figure 6.10: Abstract Automaton with Multiple Self-contained Automata Abstracted

It is important to note that when an abstract automaton is constructed the assumptions and associated interaction sequences change, as the sequences that can be generated from that automaton change. In automaton T_1 the original interaction sequence and assumptions are no longer valid, as those states no longer exist. Therefore, the assumptions and sequence must be modified. It is possible to generate these assumptions automatically using the simulation, however we have not included this functionality here in the tool and performed this step manually.

In the case of automaton T_1, δ_{T_1} has been reduced to a single transition "$(\Omega_9, Press, \Omega_6)$" translating to a single interaction sequence step "Press Ω_6". This may seem like an odd sequence at first, however as an abstract automaton is a high level view of the interaction sequence, this is expected. Essentially this abstract machine has split the interaction sequence into two parts, the self-contained automaton represented by state Ω_9 changes the mode of the system to "Options", while the self-contained automaton represented by state Ω_6 allows the user to view the details and return to the "RateInfusing" state.

In this small example this does not appear very useful, however, as discussed in chapter 5 we can take advantage of these abstractions to incorporate task ordering. By abstracting these two self-contained automata for the "View Pump Details" task we have simply achieved the same effect of task-ordering on a smaller scale. That is, we have reduced the state space of this task and captured two separate tasks, as opposed to starting with an abstract automaton with a state per task and expanding as necessary. This abstraction is helpful when investigating longer more complex task-widget based sequences.

To demonstrate a sequence simulation and the changes to the sequence and assumptions we focus on a simpler abstraction. In this case we construct automaton $T_2 = (Q_{T_2}, \Sigma_{T_2}, \delta_{T_2}, S_{T_2}, F_{T_2})$ by abstracting self-contained automaton "Initialise,Options" with respect to automaton T. The automaton T_2 is as follows:

$Q_{T_2} = \{Display, Button1PumpDetails, Button3RateInfusing, \Omega_9\}$

$\Sigma_{T_2} = \{Press, Observe\}$

$\delta_{T_2} = \{(\Omega_9, Press, Button1PumpDetails), (Button1PumpDetails, Observe, Display), (Display, Press, Button3RateInfusing), (Button3RateInfusing, Observe, Display)\}$

$S_{T_2} = \{\Omega_9\}$

$F_{T_2} = \{Display\}$

The self-contained automaton $T_{Initialise,Options} = (Q_{Initialise,Options},$ $\Sigma_{Initialise,Options}, \delta_{Initialise,Options}, S_{Initialise,Options}, F_{Initialise,Options})$ is as follows:

$Q_{Initialise,Options} = \{Initialise,\ Options\}$

$\Sigma_{Initialise,Options} = \{Press\}$

$\delta_{Initialise,Options} = \{(Initialise,\ Press,\ Options)\}$

$S_{Initialise,Options} = \{Initialise\}$

$F_{Initialise,Options} = \{Options\}$

Simulating sequences for the self-contained automaton is much the same as for automaton T, however the start and end assumptions are different. This is because the self-contained automaton only represents part of the overall sequence, therefore we must define new assumptions for the part of the sequence that the self-contained automaton represents.

In automaton $T_{Initialise,Options}$ we have only abstracted a single step "Press Options 1" from the original sequence, which also happens to be the first step in the sequence. Therefore the start assumptions will remain the same. However, the end assumptions will be different for the "State" observation as this part of the sequence only changes the system into the "Options" mode. Therefore, the end assumptions for the self-contained automaton $T_{Initialise,Options}$ are as follows:

End Assumptions:	VTBI Bag Size: 1000
State: Options	End Rate: Stop
Profile: Pediatrics	Battery status: Charging
Drug category: ABCDE	Bolus: Hands On Only
Drug: Dobutamine	Dose Rate Soft Min: 1
Rate: 60	Dose Rate Soft Max: 61
VTBI: 1000	Dose Rate Hard Max: 100

Infusing: Yes

Another artefact of abstracting the first step of the sequence is that the end assumptions of the self-contained automaton becomes the start assumptions for the abstract machine. This means that automaton T_2 and the associated interaction sequence has the same start assumptions as the end assumptions of self-contained automaton $T_{Initialise,Options}$. The Sequence Simulator prompts the user to select the appropriate start state for the PIM based on the new abstract automaton.

As the abstract automaton T_2 contains the remaining steps, and the first step has no other modification to the system other than to change the mode to "Options", the end assumptions for this automaton remain the same as automaton T. Therefore, in this instance we do not need to redefine the end assumptions for the interaction sequence associated with the abstract automaton T_2.

Using these new assumptions and the abstract automaton T_2 we can manually "shorten" the original interaction sequence as follows:

1. Press Button1PumpDetails 1.

2. Observe Display 1.

3. Press Button3RateInfusing 1.

4. Observe Display 1.

As expected, the first step of the original sequence is missing as this is abstracted into the self-contained automaton $T_{Initialise,Options}$. It is important to note that as we have abstracted the "Initialise" place holder we now treat the state "Ω_9" as the placeholder in the automaton T_2. This "special case" arises as a side effect of having abstracted the original start state.

A user may also generate different, and random sequences with this new abstract automaton, we need not only be limited to the sequence which we started with. However, the assumptions for these sequences would again change depending on the expected effect they have on the overall system.

With these assumptions and sequence in place, we may input the following interaction sequence into the Sequence Generator:

$(\Omega 9, \mathit{Press}, \mathit{Button1PumpDetails})$

$(\mathit{Buttom1PumpDetails}, \mathit{Observe}, \mathit{Display})$

$(\mathit{Display}, \mathit{Press}, \mathit{Button3RateInfusing})$

$(\mathit{Button3RateInfusing}, \mathit{Observe}, \mathit{Display})$

After this sequence is simulated we can see that "Options: I_Options" PIM transition is no longer present for the abstract automaton. This is as expected, as we have abstracted the part of the sequence which allows for this PIM state change. If we were to simulate the self-contained automaton $T_{\mathit{Initialise},\mathit{Options}}$ it would only contain this state change in the PIM.

6.6 Discussion

The sequence simulator is a proof-of-concept tool designed to illustrate the self-containment property functions in addition to sequence simulation. One side effect of creating a tool which is proof-of-concept is that we only demonstrate the possible functionalities using one particular set of models. However, the interactive and functional components could be modelled using different formalisms, provided that these formalisms are able to clearly map to the interaction sequence model in some way. Therefore, the general techniques described here are not restricted to one particular set of models.

The models used here are highly connected in the sense that for every widget in the interaction sequence model we require appropriate mappings and access to the corresponding behaviours for those widgets in order to simulate. We acknowledge that this is a possible weak point of the simulation, in that a single incorrect mapping could result in an incorrect simulation (incorrect here meaning that it does not have the intended effect). However, given the start and end assumptions for the interaction sequences, it is easy to identify if a simulation is as expected. A user may diagnose where a simulation “failed” by following the history of mode changes and functions simulated. Changes may be made to the models as required in the appropriate tools (PIMed, ProB Model Animator and Checker, or Sequence Simulator).

One of the main benefits of tool implementation is that we may take advantage of automatic execution of certain functions. This helps to address our initial requirements stated in chapter 3. This will be particularly useful in the testing stage, as we will demonstrate in chapter 7, in that we are able to automatically generate tests from the models we create.

Using the sequence simulator tool allows us to easily generate random sequences, in the sense that the next steps are randomly selected from valid options. This allows us to explore a large number of sequences for any given task based on a valid model generated from a formalised interaction sequence. These random sequences could be used to inform a robustness testing strategy.

Using the abstract sequences results in changes to the assumptions of that sequence. We use these changes to assumptions to specify different tests for the task we are testing. However, one limitation of this is that we will not create tests as comprehensive as those with the fully expanded model. Thus we trade off flexibility of the tests we create for high comprehensiveness. However, as stated previously in chapter 5, we could use self-containment and task ordering to control the parts of the interactive system we wish to test, as a result this is not a limitation.

We acknowledge that the limitations which come with the models we have chosen are present in the Sequence Simulator. For example, the ProB2 Library can only expand the state space of the specification to a defined point, as full expansion is impossible. Therefore, it is essential to ensure that the interaction sequences are within these boundaries, if we intend to use these for model checking. However, as the focus is on creating tests from these models, this is not a limitation of the overall approach.

6.7 Summary

In this chapter we described the necessary models required for interaction sequence simulation and discussed possibilities for model checking versus testing. We discussed the PModels, PIM, PMR, and Z specification and introduced the process for sequence simulation. In addition we discussed how interaction sequences give a more complete overview of the overlap component of an interactive system, which will become useful in our testing approach.

We introduced the Sequence Simulator as a proof-of-concept tool to demonstrate how we use the definitions in chapter 5 to simulate interaction sequences and automatically identify the self-containment property for abstraction and expansion. We described the different functions of this tool, in particular for sequence generation and simulation in addition to the self-containment property functions.

We then demonstrated the sequence simulation using the Alaris GP Pump as an example of an interactive system. We focussed on two different sequences, a complete one and one that uses the self-containment property to abstract the model, for sequence simulation. The task "View Pump Details" was used to demonstrate the process for sequence simulation. This was followed by using an abstract version of the automaton to demonstrate the simulation for the same sequence, showing how this changed the start and end assumptions.

Lastly, we finished with a discussion on the techniques we introduced, focussing on limitations and benefits of this approach. In the next chapter we will address some of these limitations and show how we can generate abstract tests to inform a comprehensive testing approach.

Chapter 7

Testing Interactive Systems using Interaction Sequences

7.1 Introduction

In this chapter we present an approach for testing interactive systems using interaction sequences. It is intended that this approach will be used within a comprehensive testing process to support existing testing techniques (such as testing the interactive and functional components separately) to improve test coverage of the interactive system in terms of its components. We discuss the different types of testing for which interaction sequences are applicable. This is followed by a description of the types of tests we can generate using the interaction sequences. We discuss how abstract tests can be generated automatically from interaction sequences and their assumptions, in addition to the benefits to this approach. We show how this might be implemented using the sequence simulator tool. This is followed by an example using the Alaris GP Pump to demonstrate the generation of specific abstract tests and how they may be implemented for a specific programming language and associated test libraries. Finally, we finish with a discussion on the benefits and limitations to this approach and outline further possibilities for testing with interaction sequences.

7.2 Interactive System Testing and Interaction Sequences

In chapter 2 we introduced several different types of testing, specifically for interactive systems. In this section we review some of these testing strategies and discuss how we could use interaction sequences as part of these for interactive system testing. We also discuss a selection of the testing strategies which we use to inform the testing approach presented later in this chapter.

7.2.1 Motivations

In chapter 3 we introduced the main motivations for this work and identified requirements for testing. Requirement three is addressed in this chapter as we discuss the different types of testing techniques for interaction sequences, and how to ensure they provide clear identification of errors (why the SUT did not behave as expected).

The motivation behind using interaction sequences is to create a more comprehensive interactive system testing approach which allows testing of the overlap component of an interactive system. There are extensive techniques used for testing the interactive and functional components of interactive systems independently. Therefore, we do not consider using interaction sequences for testing these components.

In order to test interactive systems comprehensively, we must have specified ways to define this comprehensiveness. A test oracle, as defined previously in chapter 2, allows us to determine whether a system has failed or passed a test. The inclusion of oracles therefore, provides us with comprehensiveness as these oracles can be used to determine at which point the system failed a set of tests. In this research oracles are used to define how we expect a system to behave, therefore, a failure indicates that the system behaved in some unexpected way. Using oracles in this way allows for clear identification of errors within a system.

Note that we are not considering usability testing in this approach. Similarly to testing the interactive and functional components, there is extensive research in the area of human computer interaction which helps us to better design systems to help prevent usability issues (in addition to other benefits). The focus in this testing technique is on the errors identified using interaction sequences.

Human error can cause significant problems in interactive systems, particularly for those used in safety-critical contexts. For example, the way in which the user interacts with the system is highly likely to be unexpected, particularly if there is plenty of user interaction freedom provided in terms of the number of actions they can choose. Using interaction sequences we hope to explore sequences which reflect these types of behaviours and expose potential errors. In order to do this we can take advantage of using FSA which allow us to explore interaction sequences of varying lengths for a specified task.

7.2.2 Testing Strategies applicable to Interaction Sequences

In this research we rely heavily on the ideas in lightweight formal methods and model-based testing. In this section we discuss various types of testing which interaction sequences could be applied to. This helps to demonstrate why we present a model-based testing approach.

As discussed in chapter 2, in white-box testing techniques the tester has knowledge of the internal structure of the system, while in black-box testing they do not. As we require knowledge of both the internal and external structure of the system the approach which we define later in this chapter falls under grey-box testing.

The most commonly applied testing strategy for interactive systems is robustness testing, as stated previously in chapter 2. We do not believe that robustness testing is adequate to test interactive systems as it lacks the comprehensiveness which the use of oracles can provide. However, as sequences of varying lengths can be randomly generated, we could take advantage of this to also create a robustness testing approach. In this kind of approach random sequences could be automatically generated and run on the interactive system to determine failure points. As several strategies exist for this type of testing, such as [51], we do not explore this idea further.

In chapter 2 we introduced the concept of fault prevention. In particular, our focus will be on fault removal. By using interaction sequences to inform a testing approach we intend to identify errors (or faults) for removal before they occur in use. Therefore, fault prevention is a key element of the testing approach we present.

Bottom-up testing involves testing components from the lowest level of some predefined hierarchy and moving up through the components. With the use of abstraction in the interaction sequence models we could envision a similar testing strategy, starting at the lowest levels of the abstraction (the self-contained sub-automata) and moving up to the highest levels of the abstraction (the abstract automata). However, this would require us to model and test every sub-automata within an abstract automaton, which removes the benefits provided by abstraction. That is, it removes the control the tester has over which parts of the interaction sequence to use to design tests, as all parts must be incorporated explicitly. Therefore, we do not investigate a bottom-up testing approach further. For this same reason we do not investigate top-down testing either.

A simple hazard analysis could be included in the approach, in that tasks could be identified as "hazardous" depending on the number of errors associated with each task. Tasks which have higher number of errors would be described as more hazardous while lower numbers would be considered "safer". However, as this is a simple form of hazard analysis, there is little benefit to carrying out this type of analysis using the interaction sequences.

As interaction sequences are used to define designated "paths" through the interactive system, this approach also falls under path testing. Specifically, we utilise expected paths to define interaction that is expected of the end user, in addition to the expected response behaviour from the SUT.

7.2.3 Summary of Testing Strategies

To summarise, in this chapter we will present a model-based testing approach informed using models constructed in a lightweight formal methods approach. To ensure comprehensiveness in the testing approach, in that we can clearly identify that the system either does or does not behave as expected, we include the use of oracles in defining tests for the interactive system. While we have described in 7.2.2 how interaction sequences could be used with several existing testing approaches they do not add any particular benefit to these and so our focus instead is on describing a new approach. The presented approach is from a grey-box perspective, incorporating fault prevention concepts and elements of path testing.

7.3 Generating Abstract Tests with Interaction Sequences

In this section we introduce the types of tests which can be automatically generated from the interaction sequences. We discuss the types of errors each type of test identifies. In addition to this, we discuss the benefits of abstract tests and how these are used to implement concrete tests.

7.3.1 Abstract vs. Concrete Tests

In this research we generate abstract tests from interaction sequences. An abstract test is a general description of a test that can be concretised for any testing language or framework. The main benefit of this approach is that we can design tests for the interactive system regardless of the programming language or testing language used. This prevents the tests generated from interaction sequences being tied to a specific framework or language.

Conversely, concrete tests do adhere to a specific testing language or framework and are therefore runnable. If concrete tests were generated automatically from the interaction sequences it would force a specific testing language or framework to be used. Due to the significant number of different types of interactive systems available, as evidenced in chapter 2 by the several different types of existing modelling and testing techniques available, this would create a restrictive testing approach.

This restriction would be a significant limitation of this work, in particular as interaction sequences are easily applicable to all different types of interactive systems, developed on many different platforms. Therefore, these issues are avoided by simply generating abstract tests, leaving it up to the tester which language or framework to use to generate the concrete tests.

In later sections we demonstrate the process of converting abstract tests to a particular set of concrete tests. However, the programming language and testing libraries used are easily exchangeable for other languages and tools. This is a significant benefit of combining interaction sequences with abstract tests, as this ensures the approach is applicable across different types of interactive systems which are able to be abstracted as interaction sequences on various platforms.

Bowen *et al.* describe abstract tests created from PModels, PIMs and PMRs in [7]. The abstract tests they define have a focus on the interactive component, however they are relevant here as we describe our abstract tests in a similar manner. The tests they describe define predicates for widgets of the interface dependent on the mode of the interactive system, as such they describe a static view of each widget in each mode. This aims at a component or unit testing approach rather than the integration testing we will describe here, as we describe tests with a focus on the overlap component at different points of the interaction sequence. Furthermore, they do not provide support to concretise the abstract tests. We demonstrate the ability to convert our abstract tests to concrete tests in later sections.

In addition, the abstract tests described in [7] focus on testing singular interactions with different widgets in specific modes as described by the PModels and PIMs. We expand on this by creating abstract tests to investigate behaviours are as expected in combinations of interactions. This is a direct result of using interaction sequences as an abstraction of the interactive system.

7.3.2 Generating Abstract Tests

We generate abstract tests from interaction sequences by defining the oracle for each specific test. That is, the “oracle” is the condition we check before, after, or during the simulation or execution of the interaction sequences specified.

First we discuss the different types of tests that are possible from the interaction sequence model. The tests we design have a specific focus on the overlap component, since the overlap component is often ignored in the testing process. This is as intended since the interaction sequences are one form of an abstraction for this component. As a result, they are designed to ensure that the interactive and functional components can communicate as expected.

There are several ways in which the interaction sequences can be used to define abstract tests. We discuss the different types of tests we can generate, divided into four specific categories: step tests, mapping tests, assumption tests, and values tests. We define each of these categories next.

7.3.3 Step tests

A step test involves defining an abstract test for a single step of an interaction sequence. These tests are defined as follows: "$\textit{onStep}((q, x, q')) \wedge \textit{property}((x, q'))$", where $q, q' \in Q$, $x \in \Sigma$ and $(q, x, q') \in \delta$ for a well-formed finite state automaton M as per definition 1. In this format the *property* can be replaced with each of the following, in addition to combinations of these: *isActionValid*, *isActionActive*, *isNext*, *isPrevious*, *isCurrent*. We will explain each of these properties in detail next.

In a step test an oracle is created to determine if properties for a particular step are true or false, such as validity, sequencing and availability. Validity allows us to identify if the step of an interaction sequence is allowed by the interactive system. A sequence which includes invalid steps is likely to prevent an end user from being able to complete a task successfully and may even lead to error. Validity is the primary focus of task models such as CTT or HAMSTERS (as described in chapter 2), as such we do not explore these further here, but they can similarly be included in our approach.

Sequencing tests can be devised on a step by step basis to ensure that the current, previous or next step is as expected in the interaction sequence. This is to ensure that the model allows for this sequence as specified and consequently the SUT does also. The oracle can be defined to include the current, previous or next step to check validity or availability.

Availability tests are used to ensure that a step is active and available (validity is assumed). We define availability as a sequencing test to ensure that the selected next or previous step was active and available to simulate or execute. As interaction sequences begin and end at a specific point with a sequential progression, availability tests for the next step are of particular interest here.

For example, the availability tests for a next step are defined for each different step of the interaction sequence. On each step of the interaction sequence an oracle is defined to determine if the following step is "active". This test is used to ensure that the interaction sequence defined is possible according to the model. That is, if an end user were to follow this interaction sequence, each step of the sequence would be active and available at the appropriate point.

The availability next step tests are defined in abstract tests of the form "$\mathit{onStep}((q, x, q')) \wedge \mathit{isNextActionActive}((x, q'))$" where $q, q' \in Q$ and $x \in \Sigma$ for a well-formed finite state automaton M as per definition 1. In this oracle we specify that on a step from a valid triple in δ the next expected action specified by x and q' in that triple is one of the next actions active. To check that this is true a transition must exist in the model from q to q' on alphabet symbol x, that is, there exists a relevant triple $(q, x, q') \in \delta$ for automaton M.

Oracles are typically described as an input output pair $(\mathit{input}, \mathit{output})$, where provided a specific input we expect a certain output. In terms of this type of abstract test the input is the test statement, we expect this statement to either be true or false as output. For example, an oracle may be as follows $(\mathit{onStep}(\mathit{Button}1, \mathit{Press}, \mathit{Button}2) \wedge \mathit{isNextActionActive}((\mathit{Press}, \mathit{Button}2)),$ $\mathit{False})$ (we often refer to such oracles as test assertions). This structure is the same for all the different types of tests described here.

7.3.4 Mapping Tests

Mapping tests allow us to check that a given step of the interaction sequence maps to the correct behaviours. This allows us to specify tests which specifically address the overlap behaviour. For example, we can check to ensure a widget triggers the correct underlying functionality. In terms of interaction sequences, the focus is to ensure that a specified step triggers the correct behaviour for the current mode.

The mapping tests adhere to the following format "$\mathit{onStep}((q, x, q')) \wedge \mathit{behaviourMap}((q', (\mathit{behaviour}_1, ..., \mathit{behaviour}_n)))$" where $q, q' \in Q$ and $x \in \Sigma$ for well-formed finite state automaton M as per definition 1, and the associated behaviours are from the PModels (taking into account the expected state for this step). The step of the interaction sequence is specified first followed by the behaviours triggered. In the "behaviourMap" a specific widget and the behaviours that it should trigger in this mode is defined for this sequence and assumptions. This abstract test is used to build a concrete test to ensure the correct functions are triggered on each step of the interaction sequence.

The PModel allows us to identify in a specific mode the behaviours a widget should trigger, in these tests we map this to specific steps of the interaction sequence. This is particularly important as different functions may be triggered depending on the sequence and assumptions. These tests essentially allow us to ensure that the interactive component is sending the correct instructions to the functional component in the given mode.

7.3.5 Assumption Tests

These tests are a direct result of the way in which the interaction sequences have been formalised. For reproducibility interaction sequences must be in some specified form, as discussed in chapters 3 and 4. Therefore, assumptions are used as part of interaction sequence generation to ensure the sequence always produces the same result, provided the SUT is in the correct start state.

These assumptions can be used to generate abstract tests to ensure that the SUT is in a specific state. That is, we check the observations we have specified before the first step of the interaction sequence, and after sequence execution or simulation. As a direct result of using the observations in this way we can specify exactly what these values are.

The assumption abstract tests are in the following format "$\mathit{beforeStep}((q, x, q')) \wedge \mathit{observation}(\mathit{value})$" where $(q, x, q') \in \delta$, "observation" is the name of the observation and "value" is the expected value for that observation (as specified in the formalised interaction sequence assumptions). Note that values can be any format, numerical or otherwise. Start assumptions use the format "beforeStep" while end assumptions use the format "afterStep" as appropriate. In this way we test to ensure that under these specific circumstances the interaction sequence and consequently the interactive system produces the correct output. This allows us to ensure that the functional component sends the correct responses to the interactive component, the inverse of mapping tests.

These assumption tests can be specified at any point in the interaction sequence. For example, we could specify the values at different "significant" points of the interaction sequence, such as before and after an abstract state, to determine that the observation values are as expected. This could be taken even further to specify an assumption on every step, however, we leave it up to the tester as to the amount of detail they wish to incorporate.

7.3.6 Values Testing

For each task-widget based interaction sequence, certain observations are changed throughout that sequence. These values can be specified at different points of the interaction sequence to ensure those values are as expected. One type of values testing we incorporate is boundary case testing.

In boundary case testing a test is designed to ensure that the SUT responds in an expected way around the boundaries of some value range. These types of tests can also be automatically generated from the interaction sequence and observations, provided we have access to the appropriate information about those boundaries.

For example, in the Alaris GP Pump there are specific boundary values associated with different drugs, such as indicating limits for safe amounts of medication to be infused to a patient. There can be several variables which factor into this, such as the drug being dispensed, in addition to the gender, age and weight of a patient. In particular, there are default values set for the “doseRateSoftMin”, “doseRateSoftMax”, and “doseRateHardMax”. These values specify the safe values for the rate of an infusion. We can use this boundary information for the rate observation to specify different tests for each of the drugs using interaction sequences. This would allow us to ensure for specific tasks, such as “set up an infusion”, that harmful doses are not possible.

This idea can be translated to other types of interactive systems for interaction sequences, that is for interactive systems which expect certain boundaries on values, tests can be designed for those boundaries and values. The tests are specified on certain subsequences of steps in the following format "$onSequence((q,x,q')_0,(q,x,q')_1,...,(q,x,q)_n) \wedge (value \leq observationValue \leq value)$" where steps $(q,x,q')_0$ *to* $(q,x,q')_n \in \delta$, we specify a sequence of steps from step "0" to up to step "n", while "$(value \leq observationValue \leq value)$" defines the boundaries for that value. This allows us to specify an interaction sequence which inputs values by selecting several different widgets in the interactive system. We specify that on this particular subsequence we expect this value to remain within certain boundaries.

Note that in addition to being able to specify the boundary range we can also create tests to check expected behaviour at either side of the boundaries. Following on from the Alaris GP Pump example, we could specify a test where the rate is equivalent to "doseRateHardMax + 1". In this instance, we would expect the Alaris GP Pump to respond by preventing the rate from being set to this value. This demonstrates how the values tests can be used to generate more than one type of abstract or concrete test.

In addition to boundary tests, we can specify tests for specific values by replacing "$(value \leq observationValue \leq value)$" with "$(observationValue = value)$". The value here is what we expect the observation to be equal to at a certain point in the interaction sequence.

7.3.7 Testing with the Self-containment Property

As mentioned previously, the self-containment property provides control over the state space of a finite state automaton. In terms of testing we may use either the abstract automaton or self-contained automata to generate tests of the different types described above.

The self-contained automata are useful for testing, in that we can specify tests specifically for the part of the interaction sequence that is self-contained, similarly to a fully expanded automaton. We can specify tests for the overall abstract automaton and the self-contained automata as appropriate. This provides greater flexibility over the tests which the tester may generate and use.

Of more interest is the abstract automaton and the steps of the interaction sequence which include the abstract state, we refer to them as *abstract steps*. These represent that some behaviour of the interaction sequence has been hidden and consequently have an effect on the types of tests we can generate from this type of automaton.

As behaviour is hidden for an abstract step, no behaviour is expected. Therefore, we must generate tests similarly to non-abstract step tests to reflect this. Mapping tests provide us with a way to specify that the abstract step maps or rather triggers none of the available behaviours. For example: "$\mathit{onStep}((\mathit{Button}1, \mathit{Press}, \Omega 1)) \wedge \mathit{behaviourMap}(\Omega 1, ())$". This type of test would allow a tester to ensure that no behaviours are triggered from an abstract state. This is to be expected as the behaviours which would be triggered at that point of the interaction sequence are hidden within the abstract state.

In terms of assumptions tests, as the abstract state has hidden certain behaviours, the assumptions must update accordingly. Therefore, the assumption tests are specified similarly to non-abstract sequences but must reflect this change in the assumptions. Lastly, for values testing we could specify that values do not change on a given abstract step, this would be to again reaffirm that the abstract state or rather "abstract widget" triggers no behaviours.

While we can generate these abstract tests from the models we cannot convert these directly to concrete tests. The reason for this is that the interactive system itself does not have the abstract state implemented as a widget, therefore, we cannot actually test this widget in the same way as the other parts of the interaction sequence.

One possibility for solving this issue would be to ensure that no behaviours are triggered at the point of the sequence at which the abstract step takes place. However, this would involve checking every widget within a system does not trigger a behaviour and as a result is not feasible in systems which have a large number of widgets. Therefore, we suggest using the abstract step to represent exactly that in the testing process, a point where widgets and consequently behaviours have been hidden and as a result tests cannot be created for those behaviours as they are also hidden.

While we cannot test the behaviours captured by an abstract state directly, we can test the sub-sequences before and/or after the abstract state. We can also test the overall sequence by simply skipping or ignoring the abstract step. In addition, we can test the sub-sequence represented by the abstract state as we would a sequence without abstract states. This provides control over what should or should not be tested, as we may simply abstract parts of the sequence which are self-contained to define clear boundaries around which parts of the SUT are of interest for testing purposes.

In summary, generating abstract tests for the abstract steps are possible but we cannot convert these to concrete tests due to the abstract widget being unimplemented in the actual interactive system. Therefore, we suggest that these tests be ignored in order to focus on the parts of the sequence which have not been abstracted. This provides a clear focus on what should or should not be tested in an abstract automaton. Furthermore, should testing be required or necessary on the sub-sequence hidden by the abstract state in the abstract automaton, this automaton can simply be expanded and the resulting automaton tested or the self-contained automaton may be tested itself. As a result, testers have control over which parts of the sequence should or should not be tested when using the self-containment property.

7.3.8 Summary

In this section we discussed testing supported by interaction sequences. In particular, we discussed the types of tests and test oracles generated from the interaction sequence models. Furthermore, we defined each of the four categories of tests that can be generated and defined the format for each of these. In the next section we demonstrate the abstract test generation for each of these types of tests followed by implementing these as concrete tests using a pre-defined test tool suite.

7.4 Extending the Sequence Simulator to Generate Abstract Tests

The sequence simulator presented in section 6.4 allows us to simulate interaction sequences provided we have the appropriate associated models as presented in chapter 6. The information from these models can be used to generate the associated abstract tests automatically with minimal input from the end user of the sequence simulator, who we refer to from now on as the “tester”.

The tester may use the associated models loaded to generate tests from a specific interaction sequence. Available, assumptions, values and mapping tests can be generated as per the examples given in the previous section.

In order to generate boundary tests extra information is required from the tester in terms of the specific boundaries of values for a given set of interactions and observations. This information is not captured in the interaction sequences, hence the need for further input on the behalf of the tester. This is true for all values tests as this type of information is not captured in the interaction sequence model (or other associated models).

Note that in the extension of the sequence simulator we have added only a subset of the abstract tests we can generate from the interaction sequences to illustrate the types of tests available. As discussed in the previous section there are several types of tests that we could generate from the interaction sequences.

7.5 Testing the Alaris GP Pump

In this section we demonstrate examples of abstract tests using the Alaris GP Pump. In particular, we demonstrate the abstract tests generated from the interaction sequence model and give an example for each different type of test. This is followed by a demonstration of using the abstract tests to generate concrete tests.

7.5.1 The Interaction Sequence Model

Using the techniques we described in previous chapters for formalising and modelling interaction sequences we use the sequence simulator to assist in modelling the tasks of 'set up' and 'start an infusion' using the Alaris GP Pump. We refer back to the example given in section 3.3.2. The assumptions here will remain the same as that example, however, due to the modelling process the sequence has changed slightly. This is a direct result of using the PModel to inform the interaction sequence process as opposed to the Alaris GP Pump implementation, as the PModel is an abstraction of the SUT.

While using a PModel to inform the interaction sequence has no effect on the abstract tests which we generate, this will have an effect when we concretise these tests. This is because it is possible that a widget is represented differently in the PModel compared with the actual implementation, as the PModel is an abstraction of the SUT. For example, a button widget may perform different behaviours on a click and double click. In the PModel this may be represented by two widgets "ButtonClick" and "ButtonDoubleClick", while in the implementation there is only a single "Button". This simply creates a pre-cursor step to converting the abstract tests to concrete tests in that we must have a clear definition of which widgets the model refers to in the actual implementation.

The sequence is as follows:

1. Press OnOff 1.
2. Observe Alarm 1.
3. Observe AlarmLight 2.
4. Observe Display 1.
5. Press Button1 1.
6. Press Button3 1.
7. Press Down 2.
8. Press Button1 2.
9. Press Down 1.
10. Press Button1 1.
11. Press Down 3.
12. Press Button1 1.
13. Press Button2 3.
14. Press Up 1.
15. Press Button1 2.
16. Press Up 1.
17. Press Button1 1.
18. Press Run 1.
19. Observe Display 1.
20. Observe RunLight 1.

This generates the following automaton $V = (Q_V, \Sigma_V, \delta_V, S_V, F_V)$ (figure 7.1):

$Q_V = \{\mathit{Initialise}, \mathit{OnOff}, \mathit{Alarm}, \mathit{AlarmLight}, \mathit{Display}, \mathit{Button}1, \mathit{Button}3,$
$\mathit{Down}, \mathit{Button}2, \mathit{Up}, \mathit{Run}, \mathit{RunLight}\}$

$\Sigma_V = \{\mathit{Press}, \mathit{Observe}\}$

$\delta_V = \{(\mathit{Initialise}, \mathit{Press}, \mathit{OnOff}), (\mathit{OnOff}, \mathit{Observe}, \mathit{Alarm}), (\mathit{Alarm}, \mathit{Observe},$
$\mathit{AlarmLight}), (\mathit{AlarmLight}, \mathit{Observe}, \mathit{AlarmLight}), (\mathit{AlarmLight}, \mathit{Observe},$
$\mathit{Display}), (\mathit{Display}, \mathit{Press}, \mathit{Button}1), (\mathit{Button}1, \mathit{Press}, \mathit{Button}3), (\mathit{Button}3,$
$\mathit{Press}, \mathit{Down}), (\mathit{Down}, \mathit{Press}, \mathit{Down}), (\mathit{Down}, \mathit{Press}, \mathit{Button}1), (\mathit{Button}1, \mathit{Press},$
$\mathit{Button}1), (\mathit{Button}1, \mathit{Press}, \mathit{Down}), (\mathit{Button}1, \mathit{Press}, \mathit{Button}2), (\mathit{Button}2, \mathit{Press},$
$\mathit{Button}2), (\mathit{Button}2, \mathit{Press}, \mathit{Up}), (\mathit{Up}, \mathit{Press}, \mathit{Button}1), (\mathit{Button}1, \mathit{Press}, \mathit{Up}),$
$(\mathit{Button}1, \mathit{Press}, \mathit{Run}), (\mathit{Run}, \mathit{Observe}, \mathit{Display}), (\mathit{Display}, \mathit{Observe}, \mathit{RunLight})\}$

$S_V = \{\mathit{Initialise}\}$

$F_V = \{\mathit{RunLight}\}$

Using this interaction sequence and associated finite state automaton we generate the abstract tests as described. The sequence simulator has been extended to allow for the automatic generation of these abstract tests.

7.5.2 Abstract Tests for the Alaris GP Pump

We explore each of the different types of abstract tests we can generate for the Alaris GP Pump example. In this section, we give the details of one example for each type of the tests described in the previous section. The full set of these tests are included in the repository[1].

First, we discuss the available tests which ensure that on the provided step the next action is available. This means we must specify a test for each step of the interaction sequence, this is achieved by using automaton V and automatically specifying the appropriate triple for each step of the sequence. The interaction sequence described as triples of automaton V is as follows:

[1]See `https://github.com/jessicaturner11/AlarisModels`.

1. (Initialise,Press,OnOff).
2. (OnOff,Observe,Alarm).
3. (Alarm,Observe,AlarmLight).
4. (AlarmLight,Observe,AlarmLight).
5. (AlarmLight,Observe,Display).
6. (Display,Press,Button1).
7. (Button1,Press,Button3).
8. (Button3,Press,Down).
9. (Down,Press,Down).
10. (Down,Press,Button1).
11. (Button1,Press,Button1).
12. (Button1,Press,Down).
13. (Down,Press,Button1).
14. (Button1,Press,Down).
15. (Down,Press,Down).
16. (Down,Press,Down).
17. (Down,Press,Button1).
18. (Button1,Press,Button2).
19. (Button2,Press,Button2).
20. (Button2,Press,Button2).
21. (Button2,Press,Up).
22. (Up,Press,Button1).
23. (Button1,Press,Button1).
24. (Button1,Press,Up).
25. (Up,Press,Button1).
26. (Button1,Press,Run).
27. (Run,Observe,Display).
28. (Display,Observe,RunLight).

Note that in this interaction sequence the number of steps has increased from 20 to 28 due to the repeated steps being expressed explicitly as opposed to by the number of steps being modelled as in the formalised sequence. For each step of this interaction sequence we specify tests as shown in the repository[2]. A short example follows:

$\mathit{onStep}((\mathit{Initialise}, \mathit{Press}, \mathit{OnOff})) \wedge \mathit{isNextActionActive}(\mathit{Press}, \mathit{OnOff})$

$\mathit{onStep}((\mathit{OnOff}, \mathit{Observe}, \mathit{Alarm})) \wedge \mathit{isNextActionActive}(\mathit{Observe}, \mathit{Alarm})$

[2]See `https://github.com/jessicaturner11/AlarisModels`.

...

$onStep((Display, Observe, RunLight)) \wedge isNextActionActive(Observe, Run$
$Light)$

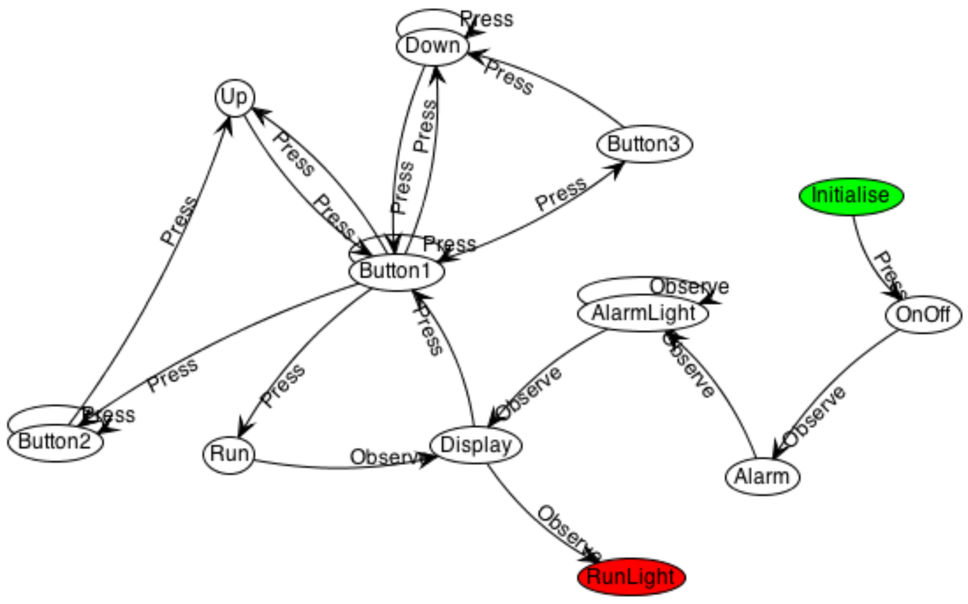


Figure 7.1: Automaton V

For the system to pass these tests on execution of this interaction sequence we must be able to ensure in some way that each step is active. For a step to be active the widget which performs that step must be enabled and/or visible. This is dependent on the widget category, for example a button widget must be able to be interacted with requiring it to be visible and enabled, while a display needs only to visible (assuming it is a non-touch interface).

The assumptions tests, as described previously, can be automatically generated from the interaction sequence assumptions. For this particular interaction sequence an important assumption to check is that the system has changed the infusing value from "no" at the start of the sequence to "yes" at the end of the sequence. This can be specified by the following pair of oracles:

$beforeStep((Initialise, Press, OnOff)) \wedge infusingIs(No)$

$afterStep((Display, Observe, RunLight)) \wedge infusingIs(Yes)$

Note that the steps are specified as before or after the first and last step of the interaction sequence respectively and the value of infusing is specified. These abstract test oracles could be used to specify all the start assumptions as one test, before execution or simulation of the interaction sequence, followed by all the end assumptions as a single test after execution or simulation. This ensures that the functional component responds as expected before and after the interaction sequence is executed (see the repository[3] for full descriptions).

As defined previously, the mapping tests are used to ensure that on a given interaction step the widgets map to the correct behaviours. These are created using the interaction sequence in combination with knowledge from the presentation models, and therefore cannot be generated from the interaction sequence FSA alone. We generate these automatically in the sequence simulator:

$onStep((Initialise, Press, OnOff)) \wedge behaviourMap(OnOff, (S\_SwitchOn,$
$I\_ClearSetup))$

$onStep((OnOff, Observe, Alarm)) \wedge behaviourMap(Alarm, ())$

...

$onStep((Button2, Press, Button2)) \wedge behaviourMap(Button2, (I\_VTBIBags))$

...

$onStep((Display, Observe, RunLight)) \wedge behaviourMap(RunLight, (S\_Infusing))$

The full set of abstract test oracles generated can be found in the repository[4]. In the oracles included above we see the various possible mappings. The widget of the step is selected from the presentation model for the current mode of the interactive system. As seen in the example above a widget can

[3]See `https://github.com/jessicaturner11/AlarisModels`.

[4]See `https://github.com/jessicaturner11/AlarisModels`.

map to *S_Behaviours* and/or *I_Behaviours* in addition to no behaviours. It is important to check that these mappings are valid as they ensure the correct instructions are sent from the interactive component to the functional component on a given step of an interaction sequence.

The last set of abstract tests which can be automatically generated from the interaction sequences are boundary tests, which are one form of value tests. In this section we include an example for the rate observation values. Remember that the end user must specify these boundaries and a sub-sequence. This sub-sequence may be the entire sequence or stop at a specific point within the sequence to capture the changes to the observation we are testing. However, with this extra information the boundary tests can be generated using the provided information. For example:

$onSequence((Initialise, Press, OnOff), (OnOff, Observe, Alarm), (Alarm, Observe, AlarmLight), (AlarmLight, Observe, AlarmLight), (AlarmLight, Observe, Display), (Display, Press, Button1), (Button, Press, Button3), (Button3, Press, Down), (Down, Press, Down), (Down, Press, Button1), (Button1, Press, Button1), (Button1, Press, Down), (Down, Press, Button1), (Button1, Press, Down), (Down, Press, Down), (Down, Press, Down), (Down, Press, Button1))$
$\wedge\ (doseRateSoftMin \leq Rate \leq doseRateHardMax) \wedge (0 \leq doseRateSoftMin \leq doseRateSoftMax) \wedge (doseRateSoftMin \leq doseRateSoftMax \leq doseRateHardMax) \wedge (doseRateSoftMax \leq doseRateHardMax \leq infusionRateMax)$

The sequence which is specified by the tester is a subsequence of the original interaction sequence, and it is not necessarily self-contained. The point of specifying this sub-sequence is to identify at which point in the interaction sequence we expect the Rate, doseRateSoftMin, doseRateSoftMax, doseRateHardMax, and infusionRateMax values to be set. The range for these values is then specified as appropriate. See the repository[5] for the remaining boundary test examples using the Alaris GP Pump for this interaction sequence.

Note that in boundary testing the use of data tables and random generation of values is typically used (see [16] and [75, p .123-127]). As we do not have access to appropriate information for the Alaris GP Pump to create these tables we have not included this in the abstract tests. However, with access to appropriate information the sequence simulator and these examples could be extended to include these techniques.

It is important to mention that while we have described the above abstract tests using the Alaris GP Pump and a non-abstract automaton we can specify tests for abstract automaton in exactly the same way (with the exception of the abstract step which is simply ignored). As a result of this we have not included these examples here.

7.5.3 Creating Concrete Tests for the Alaris GP Pump from Abstract Tests

To demonstrate that the abstract tests can be converted into concrete tests we implemented a pseudo Alaris GP Pump[6] using Java and the Swing UI library (as we do not have access to the source code for the Alaris GP Pump). Essentially, the models specified in chapter 6 were used to create this "pseudo infusion pump" which allowed conversion of the abstract tests into concrete tests for a specified programming language.

In order to specify the abstract tests we must be able to "execute" the interaction sequence on the pseudo infusion pump system. To do this for Java Swing we selected a testing tool which would allow us to programmatically

[5]See `https://github.com/jessicaturner11/AlarisModels`.
[6]See `https://github.com/jessicaturner11/AlarisJava`

specify the interaction sequence and consequently "run" it on the application. The testing library AssertJ[7] allows a tester to do this, as it is a dedicated tool for designing assertions and unit tests for Java Swing applications and their interfaces.

AssertJ was built as an extension to the pre-existing JUnit 4[8] testing tool for Java. For this type of testing JUnit 4 is used to design unit tests for the functional component while AssertJ is used for the interactive component. In order to test the overlap component it was essential that the tools selected allowed tests for both the interactive and the functional components to be specified.

In the repository[9] the relevant concrete tests created from the abstract tests given in the previous subsection are included. Note that these are only the tests generated from the 'set up' and 'start infusion' interaction sequence as specified. Several tests can be generated for sequences using different assumptions, random sequences, and obviously for different tasks. In this section we briefly explain the tests in order to explain how the abstract tests have been converted to this specific programming language and testing libraries. Note that there is "set up" work involved in using the AssertJ and JUnit 4 libraries in order to create these tests, details can be found at the appropriate websites.

The first set of concrete tests are converted from the available abstract tests. An excerpt of this test follows:

```
@Test
public void availableTest1(){
//onStep((Initialise ,Press,OnOff)) ∧ isNextActionActive(
    Press,OnOff)
```

[7]See http://joel-costigliola.github.io/assertj/
[8]See https://junit.org/junit4/
[9]See https://github.com/jessicaturner11/AlarisModels.

```
window.button(``OnOff'').requireEnabled();
window.button(``OnOff'').requireVisible();
//Perform next interaction
window.button(``OnOff'').click();
//onStep((OnOff, Observe, Alarm)) ∧ isNextActionActive(
    Observe, Alarm)
window.radioButton("Alarm").requireVisible(); ...
```

Listing 7.1: Concrete Available Test Excerpt

The first part of the abstract test specifies that the next action active must be "Press" for the "OnOff" widget on the selected step. To translate this to Java, we must ensure that the OnOff button widget in the application is enabled and visible before we can execute this interaction. The "requireEnabled()" and "requireVisible()" methods from the AssertJ library ensure that both of these requirements are true. The interaction is then executed by specifying the method "click()" for this widget. On an "observe" interaction the widget which allows this interaction must simply be visible, hence a call to "requireVisible()", to be considered available. We continue to specify the tests in this manner for each of the abstract available tests generated.

The next set of concrete tests are converted from the start and end assumptions abstract tests. As the assumptions are specified using observations from the formal specification, we can simply use JUnit 4 to specify these abstract tests as assertions. For example:

```
@Test
public void startAssumptionsTest(){
//beforeStep((Initialise, Press, OnOff)) ∧
    bolusDoseDefaultIs(0)
assertEquals(0,frame.getBolusDoseDefault());
```

```
//beforeStep((Initialise ,Press ,OnOff)) ∧ valuesIs
    ([1,10,100,1000])
Assert.assertArrayEquals(new int[]{1,10,100,1000},frame.
    getValues()); ...
```

Listing 7.2: Concrete Start Assumptions Test Excerpt

For each abstract test we simply need to add a call to the static "assertEquals" method. This method takes two arguments, the expected value followed by the actual value. If this method returns true for each of the assumptions the system has passed this test. For the end assumptions we must first execute the interaction sequence itself to ensure each step is performed correctly. This simply involves using AssertJ to access the appropriate widgets and trigger the interactions. We simplify our tests by writing a method "executeSetUpStart()" which performs these interactions using AssertJ, we can call this method each time we wish to execute the full interaction sequence as specified.

```
@Test
public void endAssumptionsTest(){
//Execute the set up and start infusion interaction
    sequence
executeSetUpStart();
//afterStep((Display ,Observe ,RunLight)) ∧
    bolusDoseDefaultIs(0)
assertEquals(0,frame.getBolusDoseDefault()); ...
```

Listing 7.3: Concrete End Assumptions Test Excerpt

Converting the mapping tests to concrete tests proved impossible for this example as tools or libraries implemented in Java which would allow us to trace the order of method execution at run time from the test suite did not exist. That is, we could not find an "out of the box" way in which to inspect that the correct behaviours were being triggered for particular widgets. This highlights the fact that the overlap component is often ignored in the testing of interactive systems.

Various techniques for reverse engineering were explored to discover ways in which we could trace the methods executed as they were triggered by widgets. In particular, Walkinshaw *et al.* describe different types of system dependencies graphs and how they have adapted these for Java [77]. In addition, Lin describes an overview of these types of graphs and how they can be used for program slicing [40]. It is possible that these types of graphs could be utilised to create a dependency graph which uses reverse engineering techniques to identify the widgets and their associated dependent methods. We could then use these graphs to determine the dependent behaviours are as expected based on the abstract mapping tests generated.

In addition to this, investigation into reverse engineering tools for Java provides some examples of using method traces for debugging purposes. Tools such as MaintainJ[10], ObjectAid[11], and Diver[12] use reverse engineering techniques to generate run-time method traces. It is feasible that tools such as these could be modified in order to provide us with a way of comparing widget execution to the method calls.

For example, consider the following assertion in JUnit:

```
//onStep((Initialise,Press,OnOff)) ∧ behaviourMap(OnOff
    ,(S_SwitchOn, I_ClearSetup))
OnOff.click();
```

[10]See www.maintainj.com
[11]See www.objectaid.com/sequence-diagram
[12]See eclipsediver.wordpress.com/docs

```
assertEquals(''SwitchOn()'', frame.getFunction());
```

Listing 7.4: Potential Concrete Mapping Test Assertion

To create this kind of mapping test we could use AssertJ to execute the interaction step and use a method trace to get the latest executed function filtered to methods triggered by the user interface, represented here by "frame.getFunction()". We could then use an assertion to determine if this function was as expected, this would allow us to implement the abstract mapping tests as concrete tests. However, we leave the implementation of a reverse engineering technique to capture these method calls for future work.

Despite being unable to convert the abstract mapping tests to concrete tests, we are able to observe the effect which indicates that mappings are correct. This is reflected in the assumptions tests, in that we can assume each function has been executed in the correct order such that the end assumptions are the same as we expect. Obviously, this assumption may be incorrect, highlighting the need for further investigation into converting the mapping tests to concrete tests.

Lastly, we specify two separate boundary tests, one for the rate boundaries and subsequence, and another for the VTBI boundaries and associated subsequence. We discuss the concrete test for the rate boundaries. The concrete test is as follows:

```
@Test
public void boundaryTestRate(){
//onSequence((Initialise ,Press ,OnOff),(OnOff,Observe,
    Alarm),(Alarm,Observe,AlarmLight),(AlarmLight,Observe
    ,AlarmLight),(AlarmLight,Observe,Display),(Display,
    Press,Button1),(Button,Press,Button3),(Button3,Press,
    Down),(Down,Press,Down),(Down,Press,Button1),(Button1
    , Press , Button1 ) ,( Button1 , Press ,Down) ,(Down, Press ,
    Button1 ) ,( Button1 , Press ,Down) ,(Down, Press ,Down) ,(Down
    , Press ,Down) ,(Down, Press , Button1 ) )
RateSubSequence ( ) ;
//∧ ( doseRateSoftMin<=Rate<=doseRateHardMax )
assertThat ( frame . getDoseRateSoftMax ( ) <= frame . getRate ( )
    ) ;
assertThat ( frame . getRate ( ) <= frame . getDoseRateHardMax ( )
    ) ;
//∧ (0 <= doseRateSoftMin <= doseRateSoftMax )
assertThat (0 <= frame . getDoseRateSoftMin ( ) ) ;
assertThat ( frame . getDoseRateSoftMin ( ) <= frame .
    getDoseRateSoftMax ( ) ) ;
//∧ ( doseRateSoftMin <= doseRateSoftMax <=
    doseRateHardMax )
assertThat ( frame . getDoseRateSoftMin ( ) <=
frame . getDoseRateSoftMax ( ) ) ;
assertThat ( frame . getDoseRateSoftMax ( ) <= frame .
    getDoseRateHardMax ( ) ) ;
//∧ ( doseRateSoftMax <= doseRateHardMax <=
    infusionRateMax )
assertThat ( frame . getDoseRateSoftMax ( ) <= frame .
    getDoseRateHardMax ( ) ) ;
assertThat ( frame . getDoseRateHardMax ( ) <= frame .
    getInfusionRateMax ( ) ) ;
}
```

Listing 7.5: Concrete Boundary Tests Excerpt

Similarly to the assumption tests we create a method which allows us to execute the subsequence using methods from the AssertJ library. We then use the "assertThat" static method to ensure that the boundaries for the different values are correct. Provided that each of these assertions is true the system will have passed this boundary test.

Using these tests we were able to discover that the pseudo Alaris GP Pump did not adhere to the PModels. In particular, simple mode transitions were not behaving as expected on the given interaction sequence. We were able to use the history of the *I_Behaviours* simulated for this sequence provided by the sequence simulator to help pinpoint which part of the program was failing. We were then able to diagnose which mode changes were incorrect and fix these as appropriate.

Furthermore, certain observations values were not as expected. This was another side effect of the pseudo implementation being created from the PModels manually, as certain behaviours had not been implemented correctly. Using the assumption tests we were able to identify which functions failed using the feedback from the testing tool and correct these functions as appropriate.

In addition to these errors, errors were seeded to demonstrate that the availability tests did correctly identify when widgets were not available as expected. This was done simply by disabling certain widgets of the user interface. As expected, the availability tests identified these seeded errors.

7.5.4 Summary

In this section we gave an example of using interaction sequences to automatically generate abstract tests for the Alaris GP Pump, specifically for the task-widget based sequence of setting up and starting an infusion. We began by introducing the interaction sequence and associated finite state automaton.

This was followed by a discussion on the different abstract tests generated for this specific task-widget based sequence. We used Java Swing to create a pseudo infusion pump based on the Alaris GP Pump in order to demonstrate how we create concrete tests from the abstract tests.

7.6 Discussion

In this chapter we have discussed the different types of tests we can generate from the interaction sequences and provided a detailed example using the Alaris GP Pump for one task-widget based interaction sequence. However, it is clear that generating abstract tests from a single interaction sequence is not enough for adequate testing coverage of the SUT. Therefore, we now describe the ways in which these tests can be applied to provide a better coverage of the SUT with a specific focus on the different tasks available.

We have deliberately made this technique adaptable to several different types of interactive systems, that is, those which are able to be abstracted into interaction sequences. As a result of this we have several different ways in which to use the interaction sequences to generate tests. This is intended to give the tester freedom to use human reasoning to decide what should and should not be tested. However, to make the adaptability of this technique clearer we now describe (in a general sense) different ways in which the interaction sequences can be used for testing purposes, incorporating the techniques we have mentioned throughout this thesis.

The formalised interaction sequences we introduced in chapter 3 provided us with a fixed technique in which to generate reproducible interaction sequences. That is, given the same assumptions and task knowledge the task-widget based sequence is able to be re-created. This allowed us to define "direct" sequences in which the system behaved as expected. However, errors are more likely to occur in the sequences which we do not expect, therefore, in chapter 4 we defined a technique to convert the formalised interaction sequences to FSA to give us a controlled way in which to explore sequences of varying lengths for the same task. Furthermore, we were able to randomly explore these FSA by selecting "randomised" steps and finding new paths through automata.

It is perhaps obvious that the assumptions for these random interaction sequences will no longer match the original assumptions. Therefore, we propose two different options for using these for testing purposes. The first is to use the sequence simulator to simulate these sequences and discover the assumptions, enabling assumption tests to be created.

In contrast, we could instead create boundary tests for specific observation values which are known to be modified by the sequence, knowledge we can gain easily from the original interaction sequence assumptions. These random sequences are then automatically generated and executed to see if the values are ever outside the boundaries. This would be to take a robustness testing approach using abstract tests including boundary oracles, and thus falls under the original requirements from chapter 3. In addition to this step mapping, and values testing is still applicable to random sequences in the same way as it is for the direct sequences, provided we have a set of correct start and end assumptions.

In chapter 4 we mention task ordering and how this is used to create a "more complete" model of the interactive system by creating interaction sequences for every task of the SUT. In chapter 5 we go on to describe how this can be more easily achieved using the self-containment property. Following the approach using self-containment it is possible to incrementally build an interaction sequence model for each task as the tester requires (that is not all tasks need to be expanded or all parts of the interaction sequences). Assumptions can be specified for each different task and task sub-automata in order to explore even more interaction sequences and consequently generate more abstract tests.

In particular, using the self-containment property with task ordering would be particularly useful when there is criteria to categorise tasks. For example, we could focus on testing only the safety-critical tasks of an interactive system, or easily assign tasks to different members of a testing team. This allows us to create a comprehensive test suite using interaction sequences for all different tasks.

It is expected that this approach is used primarily by experienced testers in order to provide comprehensive testing of the overlap component. However, as interaction sequences are a simple abstraction of the interactive system we expect this approach can be easily understood by less experienced testers. It is clear from the Sequence Simulator presented in chapter 6 that the appropriate tools could be easily created to support these types of testers.

While we have presented a testing approach in this chapter using interaction sequences we have not discussed how to resolve issues once they are identified. If a general purpose tool was created for this approach we expect standard techniques could be used to identify issues, such as identifying exceptions, providing execution traces, line numbers in code, assertions and so on. In the current version of our proof-of-concept tool only the history of the interactions is stored, this could be used as a starting point to aid in clear error identification.

When concrete tests are created in a specific programming language it is expected that issues will be clearly identified by the testing tool used to implement the concrete tests, in addition to tester knowledge and experience.

As stated previously, we provide a semi-automated approach to assist with interactive system testing with a specific focus on the overlap component. If this approach is used, we highly recommend, and expect, that testing of functional and interactive components individually also take place in addition to interaction sequence testing. That is, this approach is intended to support existing testing techniques, not to replace them.

7.7 Summary

In this chapter we have presented an approach for using the interaction sequences and their associated models as described in chapter 6 to demonstrate a testing process for creating a test suite from interaction sequences using FSA. We identified the motivations for using interaction sequences to inform a testing approach and described the testing strategies applicable to interactive system testing with interaction sequences.

This was followed by a discussion of abstract and concrete tests including the benefits to each different type. We specified the four categories of tests that we can define for the interaction sequences and gave examples for each of these. We also gave a brief overview of how the self-containment property could be used with these different types of tests.

We demonstrated how the sequence simulator was extended to include the automatic generation of the abstract tests. This was followed by an example of abstract tests being generated using the Alaris GP Pump, for the tasks of setting up and starting an infusion. We converted the abstract tests into concrete tests where possible using AssertJ and JUnit 4 for Java Swing applications. Lastly, we finished with a discussion on the adaptability of the interaction sequences and techniques we have described for further testing purposes.

Chapter 8

Conclusions

8.1 Introduction

In this research a new testing approach is presented using interaction sequences to test the overlap component of an interactive system, as a support to current interactive testing best practices. Interaction sequences are used as an abstraction of interactive systems to inform a model-based testing approach using lightweight formal methods. Testing of the overlap component is used to show that this component behaves as expected in order to improve system reliability.

8.2 Research Questions

We set out to answer the following research questions:

1. How can we generate and simulate interaction sequences automatically to ensure reproducibility?

2. Can the state space of interaction sequences be controlled while preserving the properties of the interaction sequence, so that we do not lose

information?

3. How can we use interaction sequences as an abstraction so that they may be used to inform a testing suite to enhance interactive system testing?

We present the sequence simulator tool in chapter 6 which allows us to simulate interaction sequences and generate them automatically. In addition, in chapter 3 we formalise the sequences to ensure reproducibility. This effectively addresses research question one.

We address research question two in chapter 5 by presenting an approach using the self-containment property to provide control over the state space of formalised interaction sequences modelled as Finite State Automata (FSA). By using abstraction with the self-containment property we are able to hide parts of the interaction sequence, as opposed to removing them, in order to preserve information.

To address research question three we investigated the different ways interaction sequences had been used as an abstraction in previous techniques as discussed in chapter 2. From this investigation we identified the need for interaction sequences to be formalised, as described in chapter 3 and modelled in a way which allowed us to control sequence length variation, as described in chapter 4. Lastly, in chapter 7 we present a model-based testing approach using interaction sequences, addressing the second part of research question three.

8.3 Contributions

The following contributions are made in this thesis: first a technique is presented in chapter 3 for the formalisation of interaction sequences. This was introduced to ensure reproducibility, so that the sequences can be generated in a controlled way for testing purposes. Furthermore, with a focus on task-widget based sequences we demonstrated how these could be created using task knowledge and models of the interactive system. This was the first step in being able to use the interaction sequences as an abstraction of the interactive system to allow us to inspect the overlap component behaviour.

This led into a technique for modelling interaction sequences as FSA, as presented in chapter 4. While formalised interaction sequences could be generated from pre-existing task and widget knowledge of the interactive system, this could not be done automatically. The conversion of formalised interaction sequences to FSA provided a structured way in which to explore sequences of varying lengths for specific tasks automatically. In addition, this allowed us to explore lengthier interaction sequences in a controlled way as opposed to specifying pre-defined lengths (a common approach of previous techniques as discussed in chapter 2).

In previous approaches (as discussed in chapter 2), models of interaction sequences are prone to the state space explosion problem, as conceptually they can be never ending. Therefore, a technique to control the state space of interaction sequences using the self-containment property is defined in chapter 5. While this property is applied specifically to the state space explosion problem with interaction sequences, as long as the properties of each definition are preserved, this can be extended to all FSA.

In chapter 6 the simulation of interaction sequences is demonstrated as the ground work for using these interaction sequences for testing. This is followed by a discussion on how this simulation can be used for model checking. Following this simulation a technique is presented in chapter 7 which allows abstract tests to be generated directly from FSA and formalised interaction sequences in order to test the overlap component. The conversion of these tests to concrete tests is demonstrated for a possible implementation of an interactive system using the Java programming language. Therefore, interaction sequences are shown to aid the testing process of interactive systems by demonstrating that the overlap component behaves as expected.

As stated in chapter 1, testing is a necessary part of the development process. While techniques exist to test the interactive and functional components of the interactive system individually, testing of the overlap component is ignored. The model-based testing strategy we present in this thesis specifically allows for testing of this component using lightweight formal methods. Using this approach in combination with interactive and functional testing increases the coverage of the different components of the interactive system and consequently the likelihood of finding errors. While we cannot ensure that systems using this approach will be perfect (guaranteed to be completely free of error), we can state that by increasing the likelihood of finding errors for removal we improve system reliability and safety, in order to help make systems more resilient to differing interactions and environments.

In summary, the contributions of this thesis are: a technique for formalising and generating interaction sequences (chapter 3); a technique for modelling interaction sequences as FSA (chapter 4); a technique to control the state space of interaction sequences using the self-containment property (chapter 5); and simulation of interaction sequences as an aid to testing (chapters 6 and 7).

8.4 Limitations

There are some limitations to this work, the first of these being the requirements defined in chapter 3. These requirements were as follows:

1. We must be able to automatically generate sequences of varying lengths so that the testing process is faster.

2. We must be able to constrain the sequence length in order to avoid the state space explosion problem.

3. The sequences must allow us to clearly identify why the system did not behave as expected, for example by producing counter-examples.

As the modelling and testing approaches described here are created based on these requirements we have specifically tailored the solution to these requirements. Therefore, the presented approach is limited to these requirements and as such so is the type of tests which we can generate. However, this limitation is to be expected as due to the complexity of interactive systems and testing of these systems, it would be impossible to create an approach which covers all possible requirements and types of tests.

In chapter 4 we discuss existing methods for controlling the state space in order to constrain the interaction sequences. In chapter 5 we highlight why these methods are unsuitable for our purposes, the reason for this being that we have a one-to-one mapping between a state of the automaton and a widget of the interactive system. This one-to-one mapping is a limitation on this approach as it prevents us from exploring these methods further. In future work it may be possible to explore different ways to map widgets to states of the FSA.

In chapter 5 we also discuss limitations to the self-containment property, we summarise these limitations here. The self-containment property allows us to have control over the state space but does not solve the state space explosion problem. This is because we cannot be certain that every self-contained automata we abstract will result in a smaller state space (although it is likely). In addition to this, while lemma 2 proves that every automaton is self-contained with respect to itself, it is possible that intractable automata exist which cannot be abstracted beyond the trivial case of a single state abstraction. As stated in chapter 5, it is possible that this could occur in highly connected systems.

As we have demonstrated in chapter 6 in the sequence simulator tool, we can automatically detect self-contained automata for a given finite state automaton. However, we cannot detect if an abstraction of a specific self-contained automaton will be useful or not, leaving this to human reasoning. Further work into investigating metrics which help us to detect if an abstraction is useful or not is possible, however, as human reasoning is always required to create meaningful test suites this process cannot be fully automated.

We acknowledge that the sequence simulator tool presented in chapter 6 and the concrete tests which we create in chapter 7 also have the limitations of the models and tools used to create them. However, this is not a limitation on the overall modelling and testing approach as different tools and models can be used depending on the preferences of the end user and the program or testing languages being used (with the exception of the interaction sequence models).

As we have chosen FSA to model the formalised interaction sequences this approach is limited by the amount of information which can be described in this type of formalism. In future work it would be interesting to explore variations of FSA which allow us to include further information into these models for testing purposes as appropriate.

8.5 Future work

There are several unresolved questions as a result of this investigation into using interaction sequences for testing purposes which can be explored in future work. The use of existing FSA techniques to manipulate and constrain the interaction sequences is investigated in this work, including minimisation and the removal of non-determinism in chapter 4. However, due to the one-to-one mapping between a state in the automaton and a widget in the interactive system these techniques were not applicable.

Further exploration into why particular widgets are grouped together in these techniques, and what similarities they have could have interesting consequences. For example, widgets may be combined in order to simplify the interactive system, the question here would be at which point does this simplification become confusing for the end user and as a result is non-beneficial to the enhancement of the interface?

In this thesis a technique for generating abstract tests from the interaction sequence models which use oracles to identify if the system behaves differently than expected is presented. As a result, the question arises what do we do with these problems once they have been identified? We discussed briefly in this thesis about the removal of these errors in order to improve system reliability and resilience but have not defined the exact ways in which to do this, as we expect standard procedures to be used. However, this question could be investigated in more detail for future work.

In chapter 5 limitations on using the self-containment property are considered. In particular, the differing levels of connectedness of the interaction sequences, and how a high level of connectedness would prevent the self-containment property from being applicable is discussed. Therefore, further investigation is required into the self-containment property and the effect of high levels of connectedness. We hypothesise a high level of connectedness says something about the interactive system itself, whether it may be too complex or rather provide too much freedom around interaction choice to the end user.

However, further investigation is required into high levels of connectedness as we cannot definitively say what the cause is. For example consider a standard non-modal calculator, in a task-widget based interaction sequence model this calculator would have a high level of connectedness as any of the widgets are available and able to be interacted with at any time. This type of calculator is not considered too complex or to provide a user with too much freedom of interaction, therefore, our intuitions about high connectedness may be wrong. As a result, this would be an interesting investigation for future work.

In this research, the assumptions capture the expected observations for the interaction sequence. However, in future work it would be worth investigating if these observation values could be included in the FSA, possibly by using a replacement for the FSA (as discussed in section 8.4). This would eliminate the need for a formalised interaction sequence in combination with an automaton and allow representation of the interaction sequences using only automata. This is worthy of further investigation, however, note that this would introduce complexity into the FSA, eliminating the benefit of simplicity (which is why FSA were initially selected).

The Alaris General Purpose Volumetric Infusion Pump (Alaris GP Pump) as the continuing example used throughout this thesis is a safety-critical modal interactive system. While some investigations into different types of interactive systems and the applicability of interaction sequences has taken place, concrete investigation into the applicability of these models into different types of interactive systems is required. It is intended that these models are applicable to all interactive systems, but as interactions evolve along with the way in which end users prefer to interact, the applicability of these models and the nature of the interaction sequences may change. This would require future work in order to ensure this applicability.

The use of task ordering has been discussed throughout this thesis in order to build a more complete model (based on task coverage) using the task-widget based sequences. Further investigation is required into task ordering and the benefits available in terms of testing and modelling. Depending on the criteria of the test, it would be interesting to see how task coverage affects the tests generated from the interaction sequences. That is, is a task-complete model using the task-widget based sequences required?

In chapter 6 model checking is discussed and how the sequence simulator could be adapted to prove different properties about the model and consequently the underlying interactive system. This has not been investigated further here, as the focus in this work was to create a testing approach using interaction sequences, this is another possibility for future work.

We expect this testing approach to be applicable to all interactive systems on the basis that all interactive systems have some form of interaction sequences. However, we leave it human reasoning on the part of the tester whether or not this technique will be useful for their particular system. Further investigation is possible into the exact types of interactive systems for which testing the overlap component is most useful, however this is an issue for future work.

In this research we have not directly compared this technique to existing testing techniques. As discussed in chapter 2 many existing techniques are used to test either the interactive or functional components, not the overlap component. Therefore, as the technique presented here has a focus on the overlap component it would be irrelevant to compare it to these existing techniques. However, comparisons into any new techniques which also focus on the overlap component is possible as part of future work.

8.6 Concluding Remarks

In the testing of interactive systems, the interactive and functional components are often tested separately, however, issues can arise where these components overlap. In this research, this is defined as the overlap component of an interactive system which contains the instructions the interactive and functional components use to communicate. Should this component fail, neither the interactive or the functional components can behave as expected. Therefore, the testing of this overlap component is essential, particularly in safety-critical contexts where error can lead to harm or fatalities to end users.

Interaction sequences are an abstraction of the interactive system which provide a view of this overlap component. We investigated the formalisation, modelling, constraining, and simulation of interaction sequences in order to make use of this abstraction to inform a model-based testing approach. In particular, test oracles are used in order to generate comprehensive tests for the overlap component, which allowed for clear identification of where the overlap component did not behave as expected.

By using this testing approach in combination with other pre-existing testing techniques for the functional and interactive components, it is intended that this will give a more complete testing suite when compared with simply testing the functional and interactive components separately. Using this technique in this way aids in improving test coverage and consequently allows for more errors to be discovered. The removal of these errors, particularly before software deployment, will create safer more reliable interactive systems. Therefore, testing interactive systems using interaction sequences is a necessary and useful part of the software engineering process.

Bibliography

[1] S. Arlt, I. Banerjee, C. Bertolini, A. M. Memon, and M. Schäf. Grey-box GUI Testing: Efficient Generation of Event Sequences. *CoRR*, abs/1205.4928, 2012.

[2] B. Bailey, J. Biehl, D. Cook, and H. Metcalf. Adapting Paper Prototyping for Designing User Interfaces for Multiple Display Environments. *Personal and Ubiquitous Computing*, 12(3):269–277, March 2008.

[3] E. Barboni, J.-F. Ladry, D. Navarre, P. Palanque, and M. Winckler. Beyond Modelling: An Integrated Environment Supporting Co-execution of Tasks and Systems Models. In *Proceedings of the 2Nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '10, pages 165–174, New York, NY, USA, 2010. ACM.

[4] S. Bauersfeld and T. E. J. Vos. GUITest: A Java Library for Fully Automated GUI Robustness Testing. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 330–333, September 2012.

[5] F. Belli and C. J. Budnik. Minimal Spanning Set for Coverage Testing of Interactive Systems. In Z. Liu and K. Araki, editors, *Theoretical Aspects of Computing - ICTAC 2004*, pages 220–234, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[6] J. Bowen and S. Reeves. Modelling Safety Properties of Interactive Medical Systems. In *Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '13, pages 91–100, New York, NY, USA, 2013. ACM.

[7] J. Bowen and S. Reeves. UI-design Driven Model-based Testing. *Innovations in Systems and Software Engineering*, 9(3):201–215, 2013.

[8] M. Burnett, A. Peters, C. Hill, and N. Elarief. Finding Gender-Inclusiveness Software Issues with GenderMag: A Field Investigation. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, CHI '16, pages 2586–2598, New York, NY, USA, 2016. ACM.

[9] K. A. Butler, E. Mercer, A. Bahrami, and C. Tao. Model Checking for Verification of Interactive Health IT Systems. *AMIA ... Annual Symposium proceedings. AMIA Symposium*, 2015:349–358, January 2015.

[10] A. Butterfield and G. E. Ngondi. *VDM*. Oxford University Press, 7 edition, 2016.

[11] J. C. Campos, C. Fayollas, M. Gonçalves, C. Martinie, D. Navarre, P. Palanque, and M. Pinto. A More Intelligent Test Case Generation Approach Through Task Models Manipulation. *Proc. ACM Hum.-Comput. Interact.*, 1(EICS):9:1–9:20, June 2017.

[12] J. C. Campos and M. D. Harrison. Model Checking Interactor Specifications. *Automated Software Engineering*, 8(3):275–310, August 2001.

[13] J. C. Campos, J. Saraiva, C. Silva, and J. C. Silva. GUIsurfer: A Reverse Engineering Framework for User Interface Software. In A.C. Telea, editor, *Reverse Engineering*, chapter 2. InTech, Rijeka, 2012.

[14] S. Charfi, H. Ezzedine, C. Kolski, and F. Moussa. Towards an Automatic Analysis of Interaction Data for HCI Evaluation Application to a Transport Network Supervision System. In *Human-Computer Interaction. Design and Development Approaches*, volume 6761, pages 175–184. Springer, 2011.

[15] CNBC. Uber Suspends Self-driving Car Program after Arizona Crash. Website, March 2017. Retrieved March 13, 2018 from `https://www.cnbc.com/2017/03/26/uber-self-driving-car-arizona-crash-suspended.html`.

[16] L. Copeland. *A Practitioner's Guide to Software Test Design.* Artech House Computing Library: A Practitioner's Guide to Software Test Design. Artech House, Boston, Massachusetts; London, 2004.

[17] S. Couix and J.-M. Burkhardt. Task Descriptions using Academic Oriented Modelling Languages: A Survey of Actual Practices across the SIGCHI Community. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 6948, pages 555–570, 2011.

[18] Daily Mail Reporter. Mother Dies after Nurse makes Error Administering Drug. Website, February 2011. Retrieved August 8, 2018 from `http://www.dailymail.co.uk/health/article-1359778/Mother-dies-nurse-administers-TEN-times-prescribed-drug.html`.

[19] A. Degani, I. Barshi, and M. G. Shafto. Information Organization in the Airline Cockpit Lessons from Flight 236. *Journal of Cognitive Engineering and Decision Making*, 7(4):330–352, December 2013.

[20] J. Desel and W. Reisig. The Concepts of Petri nets. *Software & Systems Modeling*, 14(2):669–683, May 2015.

[21] E. W. Dijkstra. *A Discipline of Programming*, volume 1 of *Prentice-Hall series in automatic computation*. Prentice-Hall, Englewood Cliffs, N.J., 1976.

[22] Y. Dujardin, D. Vanderpooten, and F. Boillot. A Multi-objective Interactive System for Adaptive Traffic Control. *European Journal of Operational Research*, 244(2):601–610, July 2015.

[23] M. B. Dwyer, R., O. Tkachuk, and W. Visser. Analyzing Interaction Orderings with Model Checking. In *Proceedings. 19th International Conference on Automated Software Engineering, 2004.*, pages 154–163, September 2004.

[24] C. Fayollas. Addressing Dependability for Interactive Systems: Application to Interactive Cockpits. In *Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '13, pages 163–166, New York, NY, USA, 2013. ACM.

[25] M. Fisher. *An Introduction to Practical Formal Methods Using Temporal Logic*, chapter 5, pages 129–183. Wiley, Chichester, West Sussex, U.K.; Hoboken, N.J., 2011.

[26] United States Food and Drug Administration. Infusion Pumps. Website, January 2015. Retrieved August 8, 2018 from `http://www.fda.gov/MedicalDevices/ProductsandMedicalProcedures/GeneralHospitalDevicesandSupplies/InfusionPumps/`.

[27] O. Grumberg and H. Veith. 25 Years of Model Checking: History, Achievements, Perspectives. In *Lecture Notes in Computer Science (including*

subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), volume 5000, 2008.

[28] M. D. Harrison, J. C. Campos, and P. Masci. Reusing Models and Properties in the Analysis of Similar Interactive Devices. *Innovations Systems Software Engineering*, 11(2):95–111, June 2015.

[29] J. E. Hopcroft. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley series in computer science. Addison-Wesley, Reading, Mass., 1979.

[30] C.-Y. Huang, J.-R. Chang, and Y.-H. Chang. Design and Analysis of GUI Test-case prioritization using Weight-based methods. *The Journal of Systems & Software*, 83(4):646–659, 2010.

[31] International Software Testing Qualifications Board (ISTQB). Testing Definition. Website. Retrieved July 30, 2018 from `http://glossary.istqb.org/search/testing`.

[32] C. N. Ip and D. L. Dill. Efficient Verification of Symmetric Concurrent Systems. In *Computer Design: VLSI in Computers and Processors, 1993. ICCD'93. Proceedings., 1993 IEEE International Conference on*, pages 230–234. IEEE, October 1993.

[33] D. Jackson. Lightweight Formal Methods. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 2021. Springer Verlag, 2001.

[34] J. Jacky. *The Way of Z: Practical Programming with Formal Methods.* Cambridge University Press, Cambridge ; New York, NY, USA, 1997.

[35] M. Koziol. How Apple Watch changed Molly Watt's Life. Website, 5 May 2015. Retrieved August 8, 2018 from `http:`

//www.stuff.co.nz/technology/gadgets/68296129/How-Apple-Watch-changed-Molly-Watts-life.

[36] Glenn E. Krasner and Stephen T. Pope. A Cookbook for Using the Model-view Controller User Interface Paradigm in Smalltalk-80. *J. Object Oriented Program.*, 1(3):26–49, August 1988.

[37] P. Lee, F. Thompson, and H. Thimbleby. Analysis of Infusion Pump Error Logs and their Significance for Health Care. *British Journal of Nursing (Mark Allen Publishing)*, 21:S12, S14, S16–20, April 2012.

[38] M. Lesk. Safety Risks–Human Error or Mechanical Failure?: Lessons from Railways. *IEEE Security Privacy*, 13(2):99–102, March 2015.

[39] F. Lettner, C. Grossauer, and C. Holzmann. Mobile Interaction Analysis: Towards a Novel Concept for Interaction Sequence Mining. In *Proceedings of the 16th International Conference on Human-computer Interaction with Mobile Devices & Services*, MobileHCI '14, pages 359–368, New York, NY, USA, 2014. ACM.

[40] F. Lin. Analysing Reverse Engineering Techniques for Interactive Systems. Master's thesis, The University of Waikato, Hamilton, New Zealand, 2012.

[41] J. Magee and J. Kramer. *Concurrency: State Models & Java Programs*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[42] C. Martinie, P. Palanque, and M. Winckler. Structuring and Composition Mechanisms to Address Scalability Issues in Task Models. In *Human-Computer Interaction–INTERACT 2011*, volume 6948, pages 589–609. Springer, 2011.

[43] P. Masci. A Preliminary Hazard Analysis for the GIP Number Entry Software, 2014.

[44] P. Masci, A. Ayoub, P. Curzon, M. D. Harrison, I. Lee, and H. Thimbleby. Verification of Interactive Software for Medical Devices: PCA Infusion Pumps and FDA Regulation as an Example. In *Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '13, pages 81–90, New York, NY, USA, 2013. ACM.

[45] P. Masci, P. Oladimeji, P. Curzon, and H. Thimbleby. Using PVSio-web to Demonstrate Software Issues in Medical User Interfaces. In M. Huhn and L. Williams, editors, *Software Engineering in Health Care*, pages 214–221, Cham, 2017. Springer International Publishing.

[46] P. Masci, Y. Zhang, P. L. Jones, H. Thimbleby, and P. Curzon. A Generic User Interface Architecture for Analyzing Use Hazards in Infusion Pump Software. In *Proceedings of the 5th Workshop on Medical Cyber-Physical Systems*, pages 1–14, 2014.

[47] A. Matthews-King. Faulty Opiate Injection Pumps used in Gosport Scandal to be Reviewed Amid Fears Over Deaths Across NHS, Hunt says. Website, June 2018. Retrieved July 17, 2018 from `https://www.independent.co.uk/news/health/gosport-scandal-opiate-painkillers-jane-barton-patient-deaths-a8415916.html`.

[48] G. J. Myers. *The Art of Software Testing*, chapter 1, pages 1–5. John Wiley & Sons, Hoboken, N.J., 3rd ed.. edition, 2012.

[49] G. J. Myers, C. Sandler, and T. Badgett. *The Art of Software Testing*. John Wiley & Sons, Hoboken, N.J., 3rd ed.. edition, 2012.

[50] New Zealand Herald. Elderly Patient's Cancer goes Unnoticed for Eight Months due to System Failure. Website, August 2018. Retrieved August

9, 2018 from `https://www.nzherald.co.nz/nz/news/article.cfm?c_id=1&objectid=12102251`.

[51] B. N. Nguyen, B. Robbins, I. Banerjee, and A. Memon. GUITAR: An Innovative Tool for Automated Testing of GUI-driven Software. *Automated Software Engineering*, 21(1):65–105, March 2014.

[52] J. Nielsen and R. Molich. Heuristic Evaluation of User Interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '90, pages 249–256, New York, NY, USA, 1990. ACM.

[53] P. Noble and A. Blandford. You Can't Touch This: Potential Perils of Patient Interaction with Clinical Medical Devices. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 8118, pages 395–402, 2013.

[54] C. Norris IP and D. L. Dill. Better Verification through Symmetry. *Formal Methods in System Design*, 9(1):41–75, August 1996.

[55] P. Oladimeji, H. Thimbleby, and A. L. Cox. A Performance Review of Number Entry Interfaces. In P. Kotzé, G. Marsden, G. Lindgaard, J. Wesson, and M. Winckler, editors, *Human-Computer Interaction – INTERACT 2013*, pages 365–382, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[56] A. C. R. Paiva, J. C. P. Faria, and P. M. C. Mendes. Reverse Engineered Formal Models for GUI Testing. In *Proceedings of the 12th International Conference on Formal Methods for Industrial Critical Systems*, FMICS'07, pages 218–233, Berlin, Heidelberg, 2008. Springer-Verlag.

[57] F. Paternò, C. Santoro, and L. D. Spano. Improving Support for Visual Task Modelling. In *Human-Centered Software Engineering*, volume 7623, pages 299–306. Springer, 2012.

[58] F. Paternò and E. Zini. Applying Information Visualization Techniques to Visual Representations of Task Models. In *Proceedings of the 3rd Annual Conference on Task Models and Diagrams*, TAMODIA '04, pages 105–111, New York, NY, USA, 2004. ACM.

[59] A. M. Porrello. Death and Denial: The Failure of the THERAC-25, A Medical Linear Accelerator. Website. Retrieved August 8, 2018 from `http://users.csc.calpoly.edu/~jdalbey/SWE/Papers/THERAC25.html`.

[60] C. Romanyk, R. McCallum, and P. Salehi. A Model Based Approach to Web Application Design for Older Adults using MVC Design Pattern. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9752, pages 348–357. Springer Verlag, 2016.

[61] K. Sakib, Z. Tari, P. Bertok, and A. Mukherjee. *Verification of Communication Protocols in Web Services: Model-checking Service Compositions*, chapter 2, pages 15–26. Wiley Series on Parallel and Distributed Computing; 83. John Wiley & Sons, Incorporated, 2014.

[62] P. Salem. Practical Programming, Validation and Verification with Finite-state Machines: A Library and its Industrial Application. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, pages 51–60, New York, NY, USA, 2016. ACM.

[63] I. Schieferdecker. Model-Based Testing. *IEEE Software*, 29(1):14–18, January 2012.

[64] L. D. Spano and G. Fenu. IceTT: A Responsive Visualization for Task Models. In *Proceedings of the 2014 ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '14, pages 197–200, New York, NY, USA, 2014. ACM.

[65] R. Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1(2), June 1972.

[66] The Economist. When Code can Kill or Cure. Website, June 2012. Retrieved August 8, 2018 from `https://www.economist.com/node/21556098/all-comments`.

[67] H. Thimbleby. User Interface Design with Matrix Algebra. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 11(2):181–236, June 2004.

[68] H. 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, pages 221–230, New York, NY, USA, 2009. ACM.

[69] H. Thimbleby. Heedless Programming: Ignoring Detectable Error is a Widespread Hazard. *Software: Practice and Experience*, 42(11):1393–1407, November 2012.

[70] H. Thimbleby. Action Graphs and User Performance Analysis. *International Journal of Human - Computer Studies*, 71(3):276–302, March 2013.

[71] H. Thimbleby. Reasons to Question Seven Segment Displays. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '13, pages 1431–1440, New York, NY, USA, 2013. ACM.

[72] H. Thimbleby. Safer User Interfaces: A Case Study in Improving Number Entry. *IEEE Transactions on Software Engineering*, 41(7):711–729, July 2015.

[73] H. Thimbleby, A. Gimblett, and A. Cauchi. Buffer Automata: A UI Architecture Prioritising HCI Concerns for Interactive Devices. In *Proceedings of the 3rd ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '11, pages 73–78, New York, NY, USA, 2011. ACM.

[74] J. Turner, J. Bowen, and S. Reeves. Simulating Interaction Sequences. In *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '18, pages 8:1–8:7, New York, NY, USA, 2018. ACM.

[75] M. Utting and B. Legeard. *Practical Model-based Testing : A Tools Approach.* Morgan Kaufmann Publishers, San Francisco, CA, 2007.

[76] M. Utting, A. Pretschner, and B. Legeard. A Taxonomy of Model-based Testing Approaches. *Software Testing, Verification and Reliability*, 22(5):297–312, 2012.

[77] N. Walkinshaw, M. Roper, and M. Wood. The Java System Dependence Graph. In *Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation*, pages 55–64, September 2003.

[78] B. Weyers, J. Bowen, A. Dix, and P. Palanque. *The Handbook of Formal*

Methods in Human-Computer Interaction. Human-Computer Interaction Series. Springer Publishing Company, Incorporated, 2017.

[79] L. White and H. Almezen. Generating Test Cases for GUI Responsibilities Using Complete Interaction Sequences. In *Proceedings of the 11th International Symposium on Software Reliability Engineering*, ISSRE '00, pages 110–121, Washington, DC, USA, 2000. IEEE Computer Society.

[80] L. White, H. Almezen, and N. Alzeidi. User-Based Testing of GUI Sequences and their Interactions. In *Proceedings of the 12th International Symposium on Software Reliability Engineering*, ISSRE '01, pages 54–63, Washington, DC, USA, 2001. IEEE Computer Society.

[81] J. Wing, D. Jackson, and C. B. Jones. Formal Methods Light. *Computer*, 29:20–22, April 1996.

[82] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald. Formal Methods: Practice and Experience. *ACM Computing Surveys (CSUR)*, 41(4):1–36, October 2009.