The United Nations University

UNU-IIST

International Institute for Software Technology

Pre-event Proceedings of the 1st International Workshop on

Formal Methods for Interactive Systems

F M I S 2 0 0 6

Antonio Cerone and Paul Curzon (eds)

31 October 2006

UNU-IIST Report No. 347

UNU-IIST and UNU-IIST Reports

UNU-IIST (United Nations University International Institute for Software Technology) is a Research and Training Centre of the United Nations University (UNU). It is based in Macao, and was founded in 1991. It started operations in July 1992. UNU-IIST is jointly funded by the government of Macao and the governments of the People's Republic of China and Portugal through a contribution to the UNU Endowment Fund. As well as providing two-thirds of the endowment fund, the Macao authorities also supply UNU-IIST with its office premises and furniture and subsidise fellow accommodation.

The mission of UNU-IIST is to assist developing countries in the application and development of software technology.

UNU-IIST contributes through its programmatic activities:

1. Advanced development projects, in which software techniques supported by tools are applied,
2. Research projects, in which new techniques for software development are investigated,
3. Curriculum development projects, in which courses of software technology for universities in developing countries are developed,
4. University development projects, which complement the curriculum development projects by aiming to strengthen all aspects of computer science teaching in universities in developing countries,
5. Schools and Courses, which typically teach advanced software development techniques,
6. Events, in which conferences and workshops are organised or supported by UNU-IIST, and
7. Dissemination, in which UNU-IIST regularly distributes to developing countries information on international progress of software technology.

Fellows, who are young scientists and engineers from developing countries, are invited to actively participate in all these projects. By doing the projects they are trained.

At present, the technical focus of UNU-IIST is on formal methods for software development. UNU-IIST is an internationally recognised center in the area of formal methods. However, no software technique is universally applicable. We are prepared to choose complementary techniques for our projects, if necessary.

UNU-IIST produces a report series. Reports are either Research R, Technical T, Compendia C or Administrative A. They are records of UNU-IIST activities and research and development achievements. Many of the reports are also published in conference proceedings and journals.

Please write to UNU-IIST at P.O. Box 3058, Macao or visit UNU-IIST's home page: http://www.iist.unu.edu, if you would like to know more about UNU-IIST and its report series.

G. M. Reed, Director

The United Nations University

UNU-IIST

International Institute for Software Technology

P.O. Box 3058
Macao

Pre-event Proceedings of the 1st International Workshop on

Formal Methods for Interactive Systems

F M I S 2 0 0 6

Antonio Cerone and Paul Curzon (eds)

Abstract

Reducing the likelihood of human error in the use of interactive systems is increasingly important: the use of such systems is becoming widespread in applications that demand high reliability due to safety, security, financial or similar considerations. Consequently, the use of formal methods in verifying the correctness of interactive systems should also include analysis of human behaviour in interacting with the interface.

This report includes pre-event versions of all papers accepted for presentation at the *1st International Workshop on Formal Methods for Interactive Systems (FMIS 2006)* held on 31 December 2006 in Macau SAR China, as a satellite event of ICFEM 2006. FMIS 2006 is organized by UNU-IIST in collaboration with the Human Error Modeling (HUM) project sponsored by EPSRC on research grants GR/S67494 and GR/S67500.

The aim of this workshop is to bring together researchers in computer science and cognitive

psychology, from both academia and industry, who are interested in developing formal and semi-formal methodologies and tools for the evaluation and verification of interactive systems. The outcome is to establish a worldwide network of researchers interested in applying formal methods to HCI. The workshop focuses on general design and verification methodologies based on cognitive psychology as well as application areas such as mobile devices, embedded systems, safety-critical systems, high-reliability systems, shared control systems, digital libraries, eGovernment, pervasive systems, augmented reality.

All papers submitted to the workshop were reviewed by 3 referees followed by a meta-review process. From the 18 submissions the Program Committee selected 6 papers for long presentation (40 minutes) and 2 papers for short presentation (20 minutes). Revised versions of the 6 papers accepted for long presentations will be published after the workshop by Elsevier B. V. in the *Electronic Notes in Theoretical Computer Science* Series. The authors of the most relevant papers presented at the workshop will be invited to submit extended versions of their papers to be considered for publication in a special issue of the journal *Theoretical Computer Science*.

In addition to contributed papers, the workshop programme also includes a keynote talk by Harold Thimbleby from the University of Wales, Swansea. His presentation is entitled "Building dependable interactive systems" and uses medical devices as a case study to discuss the problems of making interactive systems dependable. It introduces suggestions for improving them, particularly by using formal methods.

Antonio Cerone Antonio Cerone joined UNU-IIST as Research Fellow in 2004. His research interests includes formal methods and their application to interactive systems, information security, asynchronous hardware, concurrent and real-time systems, and safety-critical systems. E-mail: antonio@iist.unu.edu.

Paul Curzon Paul Curzon is a Reader in Computer Science at Queen Mary, University of London. His main research interest concerns formal user modeling and the verification of interactive systems. He also has a wider interest in both usability evaluation methods and the use of machine-assisted proof for hardware and software verification as well as for the verification of verification systems themselves. E-mail: pc@dcs.qmul.ac.uk

Contents

Workshop Programme

09:00 REGISTRATION
09:30-09:40 WELCOME

09:40-11:10 SESSION 1: SECURITY AND USABILITY - PART 1 (Chair: Harold Thimbleby)
09:40-10:20 Model-checking Driven Design of Interactive Systems (Long Presentation)
by Antonio Cerone, Norzima Elbegbayan
10:20-11:00 Detecting Cognitive Causes of Confidentiality Leaks (Long Presentation)
by Rimvydas Ruksenas, Paul Curzon, Ann Blandford
11:10-11:30 COFFEE BREAK

11:30-12:30 SESSION 2: SECURITY AND USABILITY - PART 2 (Chair: Antonio Cerone)
11:30-12:10 Refinement: a constructive approach to formal software design for a secure e-voting interface (Long Presentation)
by J Paul Gibson, Dominique Cansell, Dominique Mry
12:10-12:30 Guaranteeing Consistency in Text-Based Human-Computer-Interaction (Short Presentation)
by Bernhard Beckert, Gerd Beuster
12:30-14:00 LUNCH

14:00-15:20 SESSION 3: BRIDGING GAPS BETWEEN APPROACHES (Chair: Siraj Shaikh)
14:00-14:40 Formal Models for Informal GUI Designs (Long Presentation)
by Judy Bowen, Steve Reeves
14:40-15:20 Towards a Common Semantic Foundation for Use Cases and Task Models (Long Presentation)
by Daniel Sinnig, Patrice Chalin, Ferhat Khendek
15:20-15:50 COFFEE BREAK

15:50-16:50 KEYNOTE SPEAKER: Harold Thimbleby (Chair: Paul Curzon)
Building dependable interactive systems
by Harold Thimbleby

16:50-17:50 SESSION 4: MODELLING INTERACTIVE SYSTEMS (Chair: Adegboyega Ojo)
16:50-17:30 Towards a Coordination Model for Interactive Systems (Long Presentation)
by Marco Antonio Barbosa, Lus Soares Barbosa, Jos Creissac Campos

17:30-17:50 Some Issues in Modeling the Performance of Soft Keyboards with Scanning (Short Presentation)
by Samit Bhattacharya, Anupam Basu, Debasis Samanta, Souvin Bhattacherjee, Animesh Srivastava

17:50-18:15 DISCUSSION AND CLOSING SESSION

Program Committee

Ann Blandford
UCL Interaction Center, UK

Ralph Back
Åbo Akademi, Finland

Howard Bowman
University of Kent, UK

George Buchanan
University of Wales Swansea, UK

Antonio Cerone ***(Program Co-chair)***
United Nations University, Macau SAR China

Paul Cairns
UCL Interaction Center, UK

Josè Creissac Campos
University of Minho, Portugal

Paul Curzon ***(Program Co-chair)***
Queen Mary, University of London, UK

Gavin Doherty
Trinity College, University of Dublin, Ireland

Michael Harrison
University of Newcastle upon Tyne, UK

C. Michael Holloway
NASA Langley Research Center, USA

Chris Johnson
University of Glasgow, UK

Alan Dix
Lancaster University, UK

Li Siu Pan
Macao Polytechnic Institute, Macau SAR China

Peter Lindsay
The University of Queensland, Australia

Adegboyega Ojo
UNI-IIST, Macau SAR China

Philippe Palanque
University of Toulouse III, France

Fabio Paternò
CNR-ISTI, Italy

Rimvydas Ruksenas
Queen Mary, University of London, UK

Model-checking Driven Design of Interactive Systems

Antonio Cerone [1] and Norzima Elbegbayan [2]

International Institute for Software Technology
United Nations University
Macau SAR China

Abstract

This paper describes a model-checking based methodology to detect systematic errors commonly made by non-expert users. The human and computer components of the systems are modelled separately. The human component consists of a general model of the user's cognitively plausible behaviour, which can be then refined into specific instances of behaviour that reflect relevant aspects of users' personalities and skills. We consider, as a case study, a formal model of an online interactive tool that enables conference attendees to share thoughts and reactions and select matching attendees to start communication with. Starting from the initial system design, a model-checking technique is used to highlight system vulnerabilities that arise from interactions with non-expert users and may lead to security violations. The results of the analysis are exploited to improve the design by introducing safeguards that reduce or even prevent security violations.

Key words: formal verification, human behaviour, usability, user error, social computing, process calculi, model-checking.

1 Introduction

The widespread use of computers in safety-critical and security systems increases the need for human-computer interaction to be designed in a way that reduces the likelihood of human errors.

Human Reliability Assessment (HRA), which mostly emerged in the 1980's, have been widely used in the analysis of safety-critical systems but have shown little success when applied to the safety assessment of user interface design [11]. In the 1990's increasing use of formal methods has yielded more objective analysis techniques [6], which, however, mainly addressed safety-critical

[1] Email: antonio@iist.unu.edu
[2] Email: norzima@iist.unu.edu

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

aspects of Interactive Systems (IS) where the human component is represented by operators with expected expertise and skills. Moreover, the operator behaviour was often modelled as defined by the interface requirements. In reality, however, the user interacting with the system does not necessarily behave as it was expected while designing the interface, and errors are actually the very result of an unexpected user behaviour that emerge through the interaction. To best capture such an emergent behaviour, a model of the operator must specify the *cognitively plausible behaviour*, that is all the possible behaviours that can occur, and that involve different cognitive processes [1]. The model must take into account all relationship between user's actions, user's goals and the environment.

A number of researchers have explored the use of formal models to understand how cognitive errors can affect user performance. Rushby [12] models the behaviour of a forgetful operator who follows a warning display light or a non-forgetful operator without warning lights and checks for an emergent *mode confusion*. Curzon and Blandford [5] focus on goal-based interactions and model the behaviour of a user who assumes all tasks completed when the goal is achieved, but forgets to complete some important subsidiary tasks (*post-completion error)*.

In this paper we focus on IS intended for use among large groups of people, with communication, collaboration, information exchange and interest matching as the main goals [8,9]. In such a context there is no concern about safety, but the system must be easy to use regarding learnability and efficiency as well as guarantee confidentiality and other security properties, as required by legal and community policies. The complexity of the interface should be acceptable for any level of user's skill, and the system should not discourage users with its deficient or inconvenient operability. Moreover, there is little *a priori* knowledge about the user's behaviour and experience in using the system, and about more general user's skills in dealing with computers and the Internet. It is important, therefore, to set the most pessimistic scenario, in which the user is *non-expert*, with minimal skills, and also explore alternative user behaviours corresponding to a variety of attitudes and personalities. Mode confusion and post-completion errors must be considered very likely to occur. On the other hand, the system must be designed to guide user's actions and decisions by offering appropriate options in stages of interaction.

We use modelling techniques developed in previous work [2], which are based on the CSP process algebra [10] and on temporal logic, first to define the user goals and the interface as separate processes, and then to compose such processes in parallel and analyse the emergent interaction, looking for security vulnerabilities. Analysis is carried out by specifiying the properties in temporal logic and using the Concurrency Workbench of the New Century (CWB–NC) [4], a model-checker developed by SRI, to verify properties against the model. We illustrate the approach on a simple case study based on a web-based online interactive tool that enables conference attendees to communicate

and share their thoughts and reactions to a shared event [7].

Section 2 introduces a scenario that may occur during a workshop or conference, in which conference attendees share thought and opinion through a web interface motivated by various goals (Section 2.1). We also assume a basic structure of the web interface accessible through a login mechanism (Section 2.2).

Section 3 briefly introduces the CSP notation used throughout the paper and describes the model of the user behaviour in terms of three possible goals that may motivate and lead the user in interacting with the system: expressing own ideas in the forum (Section 3.1), establishing contact with a matching user (Section 3.2) and gathering information about other users (Section 3.3).

Section 4 presents an initial design of the system as the parallel composition of two CSP processes (Section 4.3), one modelling the user privileges (Section 4.1) and the other modelling the web pages (Section 4.2).

Section 5 introduces assumptions on the expertise and forgetfulness of a typical user (Section 5.1), shows how to analyse the system design under the given assumptions with respect to a property which aims to prevent security violations (Section 5.2), and describes possible improvement of the design to reduce, or even completely overcome, the vunerabilities of the initial design.

2 Case Study: A Conference System

This tool consists of a web-based interface which could be a part of a bigger system, that features a simple discussion forum and a member list. Through web pages users gather information on a conference (or some other events) and find/contact other users who are likely to match their interests. The tool, however, does not feature a *dating service* [9]. *Matching decisions* are instead explicitly made by the user.

2.1 Scenario

We start considering a common scenario that may occur at a workshop or conference. We use the word *user* to identify the main subject of our scenario. After a lecture or speech a user would like to meet other attendees to discuss impressions or reactions to the attended presentation. Such attendees might either be working on similar projects as our user or have similar thoughts about the topic of the presentation. A lecture usually involves a large number of attendees and every single attendee could have a different opinion about the topic. In series of lectures attendees do not have many chances to communicate with one another and ask opinions. Therefore it is important to allow the user to search and initiate communications before attending the conference and in order to make in advance plans and appointments for meetings to be held while being at the conference.

A conference has a website dedicated to sharing and discussing ideas and

reflections about talks and seminars. On such a website users can set up their own profiles and browse other users' profiles in order to decide whether to contact them. The purpose of the system is to help people to meet and share their opinions and reactions to lectures or other events that will be (or were) held during the conference.

The conference web site contains all information about the conference, including lists of lectures and events. We assume that users have already started sharing their thoughts and opinions well before the beginning of the conference. In a typical situation, the user accesses the web site and scans through the lists of events and lectures, and reads the lecture notes and abstracts of papers and presentations. After **logging in** the user can **set up** a profile, which includes the choice of some keywords to represent the user's professional and research interests as well as ideas and thoughts. The user can also:

read messages posted by other users to get a general view of other people's reaction to a recent event/presentation as well as to look for users with matching thoughts and opinions;

post messages to share thoughts and opinions on a recent event/presentation;

reply to messages posted by other users to support or try to confute their thoughts and opinions;

read profiles of other users to understand whether they have matching interests, ideas and thoughts;

contact users who are believed to represent good matches;

logout from the system.

We assume that every post and reply has a link where all related posts and replies are listed.

The user may have different motivations to use the conference system. These motivations, in general, depend on the user's personality, social skills, familiarity with the topics, research and professional interests, as well as practical issues such as availability of time.

Motivations determine the users' goals in using the conference system. Depending on the specific goals, the user may exploit different services provided by the system. In this paper we analyse the behaviour of the user in relation to three specific goals:

gathering information by just browsing through posted messages and user profiles;

establishing contacts with users who represent good matches;

expressing ideas by just posting messages and replying to messages (after reading them).

2.2 Web Interface

The informal description of the user's interaction with the system presented in Section 2.1 highlights three basic user statuses:

- the user has **not logged in**;
- the user has **logged in**, but **not set up a profile**;
- the user has **set up a profile**.

We assume that

(1) only logged-in users can set up their profiles and read a user profile and a message;

(2) only users who have set up a profile can post or reply to a message and contact other users.

Therefore, the user status changes when the user logs in, sets up a profile and logs out.

The system interface consists of three main web pages: a **Home** page, a page to set the **User Profile**, and a **Forum** page that allows the user to read and post messages and to access other users' profiles.

The Forum Page is linked to two additional pages. The first page allows the user to analyse **profiles** looking for matching ones and, when matchings are found, contact the corresponding users. This page is also directly accessible from the User Profile page. The second page allows the user to analyse **messages** looking for matching ones and, when matchings are found, reply to them. These two pages are mutually linked because every message has an author who must have set up a profile, due to assumption (2) above. Moreover, messages are linked to the author's profile. Similarly, a profile may be linked to message(s), if the corresponding user has already posted (or replied to) any.

3 Modelling User Behaviour

The notation that we are going to use throughout the paper is based upon Hoare's CSP notation for describing Communicating Sequential Processes [10]. We use the CWB–NC [4] syntax for CSP:

- "`a -> X`" means that action `a` occurs and then process `X` starts;
- "`X [] Y`" is the choice between processes `X` and `Y`;
- "`X [| S |] Y`" is the parallel composition of processes `X` and `Y` with synchronisation set `S`.

The *synchronisation set* defines the set of actions that must synchronise within the parallel composition.

Our model of the user behaviour focusses on the three user goals introduced in Section 2.1. The specific goal the user has in mind will drive the

choice of the appropriate actions, among those allowed by the web interface. For example, if the goal is *gathering information*, the user will just need to browse through posted messages and other user profiles, whereas if the goal is *establishing contact*, the user will eventually need to explicitly contact another user. To achieve any goal the user always needs to login in the system (assumptions (1) and (2) in Section 2.2).

```
proc Goals = ( gather_info -> login -> GatherInfo )
          [] ( establish_contact -> login -> EstablishContact )
          [] ( express_ideas -> login -> ExpressIdeas )
```

After achieving the goal, the user can either logout or choose a new goal and continue the interaction session with the system. In principle, a *cognitively plausible behaviour* [1] must include the situation in which the user may leave the interaction session unattended at any time, independently of whether the goal is achieved or not. However, such a situation is unlikely to occur when the user focusses on achieveing the goal, but it is much more plausible after the goal is achieved. It is actually common that the user assumes all tasks completed when the goal is achieved, but forgets to complete some important subsidiary tasks (*post-completion error)* [5], such as logging out of the system. Therefore we assume that

(3) the user will not logout and will not leave the interaction session unattended while trying to achieve a goal, unless failing to perform an action needed to achieve the goal;

(4) after achieving the goal, the user may forget to logout and leave the interaction session unattended.

```
proc GoalAchieved = ( gather_info -> GatherInfo )
                 [] ( establish_contact -> EstablishContact )
                 [] ( express_ideas -> ExpressIdeas )
                 [] ( logout -> Goals)
                 [] ( leave -> ( ( short_delay -> Unattended )
                                 [] ( long_delay -> Unattended ) ) )
proc Unattended = ( contact -> Unattended )
               [] ( reply -> Unattended )
               [] ( post_a_message -> Unattended )
               [] ( read_a_message -> Unattended )
               [] ( read_a_profile -> Unattended )
               [] ( logout -> Goals)
               [] ( failure -> Unattended )
               [] ( short_delay -> Unattended )
               [] ( long_delay -> Unattended )
```

State `Unattended` models the situation in which the interaction session is left unattended by an authorised user and an unauthorised user may exploit this open session by performing any action. We use actions `short_delay` and `long_delay` to characterise respectively the short or long interval elapsed between the time when the authorised user leaves the interaction session unattended (action `leave`) and an unauthorised user starts exploiting the situation

(in state `Unattended`).

3.1 Expressing Ideas

Posting messages in the forum and replying to already posted messages are ways of expressing one's ideas.

In general, the goal can be achieved (action `goal_achieved`) by either posting a message (action `post_a_message`) or replying (action `reply`) to a message, possibly after reading such message (action `read_a_message`). Note that, in general, even if we assume that the user has the intention to read a message before replying to it, we cannot assume that such intention will be always implemented in the right sequence of actions. It may happen that the user reads several messages before replying to any of them and then, intending to reply to some of them, may erroneuosly reply to a message which was not previously read.

```
proc ExpressIdeas = ( post_a_message -> goal_achieved -> GoalAchieved )
                 [] ( read_a_message -> ExpressIdeas )
                 [] ( reply -> goal_achieved -> GoalAchieved )
                 [] ( failure -> ( ExpressIdeas
                        [] ( leave -> ( ( short_delay -> Unattended )
                                     [] ( long_delay -> Unattended ) ) )
                        [] ( logout -> Goals ) ) )
```

At any stage of the interaction, the user may fail (action `failure`) to perform an action. There are four possible ways in which the user may react to such a failure:

(i) try to repeat the failed action (remaining in state `ExpressIdeas` and repeating the same action);

(ii) try an alternative action (remaining in state `ExpressIdeas` and performing a different action);

(iii) leave the interaction session unattended (action `leave`);

(iv) log out from the system (action `logout`).

In general the choice of reaction depends on the user's personality and familiarity with computer systems, as well as time availability.

3.2 Establishing contacts

In order to establish contact with a matching user it is necessary to explicitly contact that user. In general, the user who wishes to establish contact may have already collected outside the system all necessary information to select a matching user and use the system just to contact such a matching user.

```
proc EstablishContact = ( read_a_profile -> EstablishContact )
                     [] ( read_a_message -> EstablishContact )
                     [] ( contact -> goal_achieved -> GoalAchieved )
                     [] ( failure -> ( EstablishContact
                           [] ( leave -> ( ( short_delay -> Unattended )
                                        [] ( long_delay -> Unattended ) ) )
```

```
                                    [] ( logout -> Goals ) ) )
```

The system provides two ways for gathering information to help selecting a matching user: reading profiles (action `read_a_profile`) and reading messages (action `read_a_message`). The user who wishes to establish contact will keep reading profiles and messages (remaining in state `EstablishContact`) until a matching user is found, before contacting such a matching user (action `contact`). However, action `contact` may be immediately performed without any iteration in the information gathering loop, if the information gathering process has been performed outside the system. Obviously, the `goal_achieved` action must be preceeded by the `contact` action.

As for the previous goal, at any stage of the interaction, the user may fail (action `failure`) to perform an action.

3.3 Gathering Information

Gathering information about other users may not only be a means to select a matching user but also be the final goal to achieve. If gathering information is the actual user's goal, then each of the `read_a_profile` and `read_a_message` actions may either be an iteration of the information gathering loop (in state `GatherInfo`) or lead to the `goal_achieved` action, which is performed when the user has collected all needed information.

```
proc GatherInfo = ( read_a_profile -> ( GatherInfo
                                    [] ( goal_achieved -> GoalAchieved ) ) )
            [] ( read_a_message -> ( GatherInfo
                                    [] ( goal_achieved -> GoalAchieved ) ) )
            [] ( failure -> ( GatherInfo
                          [] ( leave -> ( ( short_delay -> Unattended )
                                      [] ( long_delay -> Unattended ) ) )
                          [] ( logout -> Goals ) ) )
```

4 Initial System Design

4.1 Model of User Privileges

The three basic user statuses highlighted in Section 2.2 can be formalised by three CSP processes as follows.

```
proc OutUser = ( login -> (( noprofile -> enter -> InUser )
                        [] ( profile -> enter -> Member ) ) )
proc InUser = ( setup -> Member )
           [] ( read_a_profile -> InUser )
           [] ( read_a_message -> InUser )
           [] ( logout -> OutUser )
proc Member = ( read_a_profile -> Member )
           [] ( read_a_message -> Member )
           [] ( post_a_message -> Member )
           [] ( reply -> Member )
           [] ( contact -> Member )
```

```
        [] ( logout -> OutUser )
```

Process `OutUser` defines the initial state, in which the user has not logged in yet. After the user logs in (action `login`), the system checks whether the user has already set a profile (action `profile`) or not (action `noprofile`). If the user has not set a profile yet, the state changes to `InUser`, otherwise it changes to `Member`. These two states define the two user privileges that correspond to assumptions (1) and (2) in Section 2.2. The purpose of action `enter` is to move to the state corresponding to the appropriate user privilege and to activate the web interface described in Section 4.2. User privileges can be changed by executing action `setup` (from `InUser` to `Member`). Action `logout` leads back to the initial state (`OutUser`).

4.2 Model of the Web Interface

The model of the web interface consists of six states.

```
proc Entry1 = ( enter -> Home1 )

proc Home1 = ( users -> UserProfiles1 )
          [] ( forum -> Forum1 )
          [] ( setup -> Home1 )
          [] ( logout -> Entry1 )

proc UserProfiles1 = ( forum -> Forum1 )
                  [] ( read_a_profile -> AProfile1 )
                  [] ( home -> Home1 )

proc AProfile1 = ( back_to_users -> UserProfiles1 )
              [] ( read_a_message -> AMessage1 )
              [] ( contact -> AProfile1 )

proc Forum1 = ( read_a_message -> AMessage1 )
           [] ( users -> UserProfiles1 )
           [] ( post_a_message -> Forum1 )
           [] ( home -> Home1 )

proc AMessage1 = ( read_a_profile -> AProfile1 )
             [] ( back_to_forum -> Forum1 )
             [] ( reply -> AMessage1 )
```

In the initial state (`Entry1`) the home page of the web interface is activated by action `enter` (and the subsequent change to state `Home1`), which ends the procedure to check the user privileges described in Section 4.1.

The other states model the five web pages described in Section 2.2. Actions `users`, `forum`, `home`, `back_to_users` and `back_to_forum` allow the user to freely navigate through the five web pages. Note that action `logout` is only possible from state `Home`. This means that the user has always to go back to the home page in order to be able to logout.

4.3 Overall System Model

The overall system is expressed by process `SYSTEM1` given by the parallel composition of the user privileges (initially in state `OutUser`), the web interface (initially in state `Entry1`) and the user goals (process `Goal`)

```
proc SYSTEM1 = ( (OutUser [| {enter, read_a_profile, read_a_message,
   post_a_message, reply, contact, setup, logout} |] Entry1)
   [| {login, read_a_message, read_a_profile, contact, logout,
   post_a_message, reply, failure} |] Goals)
```

The `OutUser` and `Entry1` processes must synchronise on all actions that can be eventually performed by the `OutUser` process apart from the `profile` and `noprofile`, which define the checking procedure modelled by the `OutUser` process (they are internal actions of `OutUser`).

The process originated by the parallel composition of `OutUser` and `Entry1` is then composed with the `Goals` process. In this second parallel composition, the synchronisation must include all user actions that define interactions with the interface. Note that we have also included the `failure` action, which does not occur in the `OutUser` and `Entry1` processes (and therefore neither in their parallel composition) in the sychronisation set. This prevents the overall system from performing the `failure` action, so modelling the following assumption

(5) user never fails to perform an intended action that is immediately available on the current web page.

The purpose of this assumption is to show that the design weaknesses captured by the model-checking analysis presented in Section 5 are independent of the skill of the user in performing a single interaction with the interface.

5 Improving the System Design

In this section we use CWB–NC to analyse the interaction between the user behaviour model defined in Section 3 and various versions of the web interface defined in Section 4. The analysis of the original web interface defined in Section 4 highlights security vulnerabilities, which are then partly or entirely overcome in the next versions.

5.1 Constraining the User Behaviour

According to assumptions (3) and (4) in Section 3 the user will not logout and will not leave the interface unattended before achieving the goal. However, after achieving the goal, the user may forget to logout and leave the interface unattended without coming back to use it. Such a situation may lead to *security violations*. The interface is supposed to be designed with the aim to minimise the likelihood that an unattended session is exploited by a non-authorised user to access profiles (*privacy violation*) or pretend to be the

logged-in user (*masquerading*).

Ideally, we would like an unattended session to automatically logout *on time* to prevent security violations. However, in practice, we can just introduce *safeguards* that minimise the likelihood of security violations, in a way that does not introduce much degradation in the quality and performance of the services provided to the user. In order to find the right balance between security and the quality and performance services, it is important to analyse the user attitudes and behaviours while interacting with the system. Specific attitudes and behaviours may actually reduce the likelihood of some threats and increase the likelihood of others. For example, panicking when the planned action does not appear immediately available on the current web page is an attitude that may lead to the behaviour of leaving the session unattended, so causing a security threat. On the other hand, the attitude of always checking that all tasks have been completed after achieving a goal reduces the likelihood of forgetting to logout before leaving the session.

It is therefore sensible to introduce safeguards only to prevent the most likely threats, that is, those threats that are more likely to occur given specific assumptions about user attitudes and behaviours. The users of our system are not supposed to be expert in using interactive systems. In fact, some of them might have very low familiarity with computers. The user behavior model defined in section 3 is a very general one and needs to be restricted to capture specific attitudes and behaviours of non-expert users.

A typical behaviour of a non-expert user after achieving a goal is to try to logout but, if such attempted logout fails, to eventually leave the session unattended. Such a behaviour may be enforced by a process defined as follows.

```
proc NonExpert = ( home -> NonExpert )
              [] ( users -> NonExpert )
              [] ( forum -> NonExpert )
              [] ( back_to_users -> NonExpert )
              [] ( back_to_forum -> NonExpert )
              [] ( leave -> NonExpert )
              [] ( logout -> NonExpert )
              [] ( goal_achieved ->
                   ( ( logout -> NonExpert )
                  [] ( leave -> NonExpert )
                   )
                 )
```

This process has then to be composed in parallel with the system as follows.

```
proc SYSTEM1N = ( SYSTEM1 [| {home, users, forum, back_to_users,
        back_to_forum, goal_achieved, leave, logout} |] NonExpert )
```

Apart from `leave` and `logout`, any other action on which the two processes synchronise cannot occur after the `goal_achieved` action, consistently with the behaviour of a non-expert user described above.

The user defined by the `SYSTEM1N` process may, however, forget to logout and leave the session unattended even if there is a logout mechanism

(e.g. a logout button) promptly available on the current web page. This occurs when actions `leave` and `logout` are both available but `leave` is chosen. A non-forgetful user, who will always choose a `logout` action when available after `goal_achieved`, rather than leaving the session unattended (action `leave`), may be defined by appropriately synchronising the system with the `NonForgetful` process defined as follows.

```
proc NonForgetful =  ( goal_achieved ->  logout -> NonForgetful )
                  [] ( logout -> NonForgetful )
```

The appropriate synchronisation is achieved by composing this process, which works as a *constraint*, in parallel with `SYSTEM1N` as follows.

```
proc SYSTEM1NR = ( SYSTEM1N [| {goal_achieved, leave, logout} |]
                   NonForgetful )
```

Since the two components must also synchronise on `leave`, and this does not occur in the behaviour of the `NonForgetful` process, the `leave` action can never occur in the composite process.

5.2 Analysis of the Initial Design

An important requirement that our system should meet is that an open session is never left unattended. Meeting such a requirement would definitely prevent security violation from occurring.

According to assumptions (3) and (4) in Section 3 an open session may be left unattended only if the logged-in user, after achieving a goal, does not pursue a new goal and does not logout. Using CWB–NC syntax [4], a property asserting that every time a goal is achieved there will eventually be either a logout or a new goal pursued can be formalised as follows.

```
prop eventually_logout = A G ( {goal_achieved} ->
   F ( {gather_info} \/ {establish_contact} \/ {express_ideas} \/ {logout} ) )
```

This property is not stating that the session is never left unattended. It would be anyway useless to state a property that depend only on the *free will* of the user rather than on the way the interface constrains the user behaviour. Action `leave` may actually occur after action `leave` and before persuing a new goal (`gather_info` or `establish_contact` or `express_ideas`) or logging-out (`logout`).

Property `eventually_logout` prevents any action in $\mathcal{P}$ = { `contact`, `reply`, `post_a_message`, `read_a_message`, `read_a_profile` } from occuring between achieving a goal and persuing a new goal or logging-out. In fact, if such an occurrence were possible, then also an infinite sequence containing only actions in $\mathcal{P}$ would be possible after achieving the goal, which contraddict property `eventually_logout`.

Therefore property `eventually_logout` guarantees that if the the session is left unattended (`leave`) after achieving a goal (`goal_achieved`), then no unauthorised user can take over the session and perform actions in $\mathcal{P}$, possibly

leading to a security violation, before logging-out (pursuing a goal must always follow a `logout` action).

We should note that, in principle, the authorised user might also indefinitely browse web pages with no real goal and never logout, whereas this behaviour is also ruled out by property `eventually_logout`. However, this is not a cognitively plausible behaviour, not only because the user will not sit at the session for an infinite time, but also because we assume that all our users are strongly motivated in achieving goals.

Using the CWB–NC we can verify that `eventually_logout` does not hold for the `SYSTEM1NR` system. Therefore, the web interface needs to be improved to address non-expert users. The problem is that the logout is not available on each web page, but just on the Home page. The users have to properly navigate back to the Home page from the page where the goal has been achieved. This might be quite challenging for a non-expert user. In addition, the presence of a logout button on each web page would be a reminder for the user to logout, so addressing also expert but forgetful users.

We therefore modify the interface (renaming `Entry1` to `Entry2`, and analogously for the other states) by inserting in every state apart from `Entry2` (and `Home2`, which already has it) the choice

```
[] ( logout -> Entry2 )
```

The processes that define the composite system are defined as follows.

```
proc SYSTEM2 = ( (OutUser [| {enter, read_a_profile, read_a_message,
   post_a_message, reply, contact, setup, logout} |] Entry2)
   [| {login, read_a_message, read_a_profile, contact, logout,
   post_a_message, reply, failure} |] Goals)

proc SYSTEM2N = ( SYSTEM2 [| {home, users, forum, back_to_users,
        back_to_forum, goal_achieved, leave, logout} |] NonExpert )

proc SYSTEM2NR = ( SYSTEM2N [| {goal_achieved, leave, logout} |]
     NonForgetful )
```

We can now verify, using CWB–NC, that `eventually_logout` holds for the `SYSTEM2NR` system.

5.3 Introducing a timeout

A problem with the interface defined by `SYSTEM2` is the lack of any protection for forgetful users. Although adding a direct logout mechanism to each web page may work as a reminder to the user to logout, users might still forget to logout. Property `eventually_logout` does not actually hold for `SYSTEM2N`.

A way of improving the situation is the introduction of a *timeout* in the interface to force the system to automatically logout if there is no action by the on-line user, within a given time. We may modify the interface (renaming `Entry2` to `Entry3`, and analogously for the other states) by inserting in every state, apart from the initial one (`Entry3`), both choice

```
                [] ( long_delay -> timeout -> logout -> Entry3 )
```

and the choice of action `short_delay` with no change of state. For example, state `UserProfiles3` is defined as follows.

```
proc UserProfiles3 = ( forum -> Forum3 )
                   [] ( read_a_profile -> AProfile3 )
                   [] ( home -> Home3 )
                   [] ( logout -> Entry3 )
                   [] ( short_delay -> UserProfiles3 )
                   [] ( long_delay -> timeout -> logout -> Entry3 )
```

The timeout will occur only after some time (`long_delay`), which is long enough not to disrupt the short idling periods that users normally have during sessions. In general, this safeguard does not fully solve the problem of unauthorised accesses. In fact, an unauthorised user can still enter an unattended session before the timeout expires. Let us consider the composite system defined as follows.

```
proc SYSTEM3 = ( (OutUser [| {enter, read_a_profile, read_a_message,
   post_a_message, reply, contact, setup, logout} |] Entry3)
   [| {login, read_a_message, read_a_profile, contact, logout,
   short_delay, long_delay, post_a_message, reply, failure} |] Goals)

proc SYSTEM3N = ( SYSTEM3 [| {home, users, forum, back_to_users,
        back_to_forum, goal_achieved, leave, logout} |] NonExpert )
```

We can actually verify using CWB–NC that `eventually_logout` does not hold for the `SYSTEM3N` system. However, we can assume that

(6) no authorised user may enter an unattended session within a time period shorter (`short_delay`) than the delay (`long_delay`) that triggers the timeout.

Such an assumption may be modelled as follows.

```
proc QuickTimeOut = ( home -> QuickTimeOut )
           [] ( users -> QuickTimeOut )
           [] ( forum -> QuickTimeOut )
           [] ( back_to_users -> QuickTimeOut )
           [] ( back_to_forum -> QuickTimeOut )
           [] ( logout -> QuickTimeOut )
           [] ( leave -> long_delay -> timeout -> logout -> QuickTimeOut )
```

The process that incorporates this assumption is defined as follows.

```
proc SYSTEM3NQ = ( SYSTEM3N [| {timeout, home, users, forum, back_to_users,
   back_to_forum, leave, logout, short_delay, long_delay} |] QuickTimeOut  )
```

We can verify that `eventually_logout` holds for the `SYSTEM3NQ` system.

5.4 Introducing authentication

All safeguards introduced in previous sections contribute to reduce the likelihood of security violations, but they do not guarantee the absence of such

violations. Those safeguards aim actually to reduce the likelihood that an unauthorised user enter an unattended open session, but there are no explicit mechanisms in the system to prevent an unauthorised user, who has actually entered the session, from performing interactions with the systems.

We could modify the web interface by introducing a protection mechanism that requires the user to provide authentication (i.e. to input a password) before performing a specific critical action. For example, we would like to avoid *masquerading threats* by requiring authentication to users who wish to contact other users.

We may modify the interface (renaming `Entry3` to `Entry4`, and analogously for the other states) by replacing the `AProfiles3` with the following:

```
proc AProfile4 = ( back_to_users -> UserProfiles4 )
              [] ( read_a_message -> AMessage4 )
              [] ( request_contact ->
                         ( authenticated -> contact -> AProfile4 )
                      [] ( short_delay -> failure -> AProfile4 ) )
              [] ( logout -> Entry4 )
              [] ( short_delay -> AProfile4 )
              [] ( long_delay -> timeout -> logout -> Entry4 )
```

An analogous change must be done to state `Forum3`, in which messages can be posted, by introducing action `request_post`. These are the only two states in which an unauthorised user may perform a masquerading attack.

Posting a message and contacting another user are now allowed only upon successful authentication. For example action `contact` may occur only after action `authenticated`. If action `failure` occurs instead (the authentication fails), then action `contact` cannot occur (no contact can be established).

We assume that

(7) only authorised users can be authenticated.

This is a reasonable assumption, which in password-based authentication corresponds to the assumption that the password is kept secret. Assumption (7) may be defined by a constraint as follows.

```
proc Authorised = ( failure -> Authorised )
               [] ( authenticated -> Authorised )
               [] ( leave -> UnAuthorised )
proc UnAuthorised = ( failure -> UnAuthorised )
                 []  ( logout -> Authorised )
```

This process models the presence of an authorised user until action `leave` occurs, and then the presence of an unauthorised user, with no `authenticated` action allowed, until action `logout` occurs. Note that this constraint is also consistent with assumption (5) in Section 4.3.

The processes that define the composite system are defined as follows.

```
proc SYSTEM4 = ( (OutUser [| {enter, read_a_profile, read_a_message,
   post_a_message, reply, contact, setup, logout} |] Entry4)
   [| {login, read_a_message, read_a_profile, contact,
```

```
   logout, short_delay, long_delay, request_post, post_a_message,
   request_reply, reply, failure} |] Goals)

proc SYSTEM4A = ( SYSTEM4 [| {failure, authenticated, leave, logout} |]
                  Authorised )
```

We would like to prove that if an open session is left unattended (action `leave`), then no contact is established (action `contact`), no message is posted (action `post_a_message`), no reply is sent (action `reply`), until at some time when the session was unattended a logout (action `logout`) occurs. Such a property can be formalised as follows.

```
prop never_masquerading =  A G ( {leave} ->
   ( ( ~{contact,post_a_message,reply} ) W {logout} ) )
```

We can verify that `prop never_masquerading` holds for the `SYSTEM4A` system and does not hold for any of the previous versions of the overall system (with the user behaviour component modified with the choice regarding `request_contact`, `request_post` and `request_reply` described above).

The web interface might be modified in a similar way to obtain a system protected from confidentiality threats. In this case the request for authentication would be associated with actions `read_a_profile` and `read_a_message` and the property to be verified would be

```
prop confidentiality =  A G ( {leave} ->
   ( ( ~{read_a_profile,read_a_message} ) W {logout} ) )
```

Finally, the web interface could be modified to protect from both confidentiality violation and masquerading. Although these mechanisms guarantee perfect security under assumption (7), it is unlikely that they would all be implemented in a real system similar to the case study we have analysed. However, if it is likely that some sensitive information may be posted on the website, privacy policies might require confidentiality just for that particular information. Then the solution above would guarantee that no unauthorised user can read confidential profiles or messages of other users. Although masquerading is still possible and some users might be driven by the attacker to post confidential information, such information could not be read by the attacker.

6 Conclusion

We have used modelling techniques developed in previous work [2] to separately model the user goals and web interface components of an interactive tool for communication, information exchange and interest matching [7].

The composition step has been carried out using the *constraint-based modelling style* [13,3], in which the behavior of a process is restricted by composing it with a special process that works as a contraint. This approach has been exploited to model alternative aspects of the behaviour of non-expert users (processes `NonExpert`, `NonForgetful`) as well as assumptions that are needed

to set the contexts in which security properties can be expressed (processes `QuickTimeOut`, `Authorised`).

Analysis has been carried out by specifying properties in temporal logic and using several iterations of model-checking to discover security vulnerabilities of the initial design and verify the correctness of the safeguards introduced.

References

[1] R. Butterworth, Ann E. Blandford, and D. Duke. Demonstrating the cognitive plausability of interactive systems. *Formal Aspects of Computing*, 12:237–259, 2000.

[2] Antonio Cerone, Peter Lindsay, and Simon Connelly. Formal analysis of human-computer interaction using model-checking. In Bernhard Aichernig and Bernhard Beckert, editors, *Proceedings of the 3rd IEEE International Conference on Software Engineering and Formal Methods*, pages 352–361. IEEE Comp. Soc., 2005.

[3] Antonio Cerone and George Milne. Property verification of asynchronous systems. *Innovations in System and Software Engineering*, 1(1):25–40, April 2005.

[4] Rance Cleaveland, Tan Li, and Steve Sims. The concurrency workbench of the new century. User's manual, SUNY at Stony Brook, Stony Brooke, NY, USA, 2000. URL: `http://www.cs.sunysb.edu/~cwb`.

[5] Paul Curzon and Ann E. Blandford. Formally justifying user-centred design rules: a case study on post-completion errors. In E.A. Boiten, J. Derrick, and G. Smith, editors, *Integrated Formal Methods*, volume 2999 of *Lecture Notes in Computer Science*, pages 461–480. Springer-Verlag, Berlin, Germany, 2004.

[6] Alan John Dix. *Formal Methods for Interactive Systems*. Academic Press, 1991.

[7] Norzima Elbegbayan. Shared reflection — case study on conference support website. Technical Report 342, UNU-IIST, 2006.
URL: `http://www.iist.unu.edu/`.

[8] Joseph F. McCarthy *et al.* Digital backchannels in shared physical spaces: attention, intention and contention. In *CSCW '04: Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work*, pages 550–553, New York, NY, USA, 2004. ACM Press.

[9] Andrew Fiore and Judith S. Donath. Online personals: An overview. In *ACM Computer-Human Interaction*, Vienna, 2004.
URL: `http://smg.media.mit.edu/papers/atf/chi2004_personals_short.pdf`.

[10] C.A.R Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice Hall, 1985.

[11] B. Kirwan. Human error identification in human reliability assessment. Part 1: Overview of approaches. *Applied Ergonomics*, 25(5):299–318, 1992.

[12] John Rushby. Using model-checking to help discover mode confusions and other automation surprises. *Reliability Engineering and System Safety*, 75(2):167–177, February 2002.

[13] C. A. Vissers, G. Scollo, M. van Sinderen, and E. Brinksma. Specification styles in distributed systems design and verification. *Theoretical Computer Science*, 89:179–206, 1991.

Detecting Cognitive Causes of Confidentiality Leaks ⋆

R. Rukšėnas [a,1], P. Curzon [a], A. Blandford [b]

[a] *Dept. of Computer Science, Queen Mary, University of London, London, UK*
[b] *University College London Interaction Centre, London, UK*

Abstract

Most security research focuses on the technical aspects of systems. We consider security from a user-centred point of view. We focus on cognitive processes that influence security of information flow from the user to the computer system. For this, we extend our framework developed for the verification of usability properties. Finally, we consider small examples to illustrate the ideas and approach, and show how some confidentiality leaks, caused by a combination of an inappropriate design and certain aspects of human cognition, can be detected within our framework.

Key words: human error, security, cognitive architecture, formal verification, SAL.

1 Introduction

There has been much research on security (confidentiality) of information flow (see Sabelfeld and Myers' overview [19]). The starting point is the assumption that computation uses confidential inputs. The goal is to ensure a *noninterference policy* [10], which essentially means that no difference in outputs can be observed between two computations that are different only in their confidential inputs. Various approaches to this problem, such as access control [3] and static information-flow control [9], have been proposed, and formalisms and mechanisms developed, e.g. security-type systems [20] and type-checkers [15].

All this research focuses on the technical aspects of software systems. It aims at ensuring that the implementation of a system does not leak confidential information. However, technology is only one aspect of security. Within interactive systems, there is another actor besides a computer system – its

⋆ This research is funded by EPSRC grants GR/S67494/01 and GR/S67500/01.
[1] Email: rimvydas@dcs.qmul.ac.uk

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

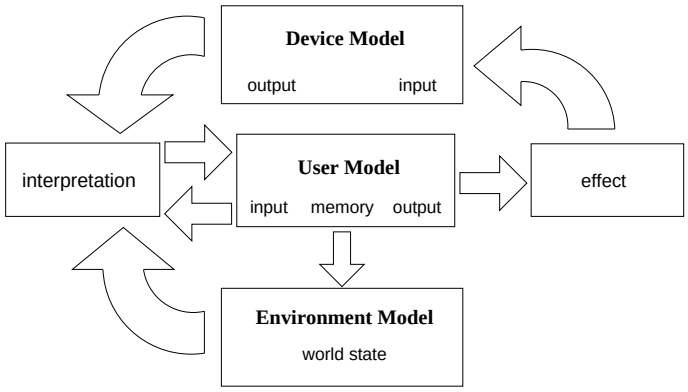

Fig. 1. The cycle of interaction

human user. Even perfectly designed and implemented systems cannot prevent users from unwittingly compromising confidential information they have. Users can breach security for many reasons. Nevertheless, research in human-computer interaction [1,13] reveals systematic causes of such violations, including cognitive overload, lack of security knowledge, and mismatches between the behaviour of computer systems and the mental model that their users have. Even in the absence of software errors security can be breached when the functionally correct behaviour is inconsistent with user expectations [13].

The relationship between users and security mechanisms is addressed by *user-centred security* which provides "security models, mechanisms, systems, and software that have usability as a primary motivation or goal" [21]. Much of this work takes social dimensions, considering problems like user motivation and understanding of security mechanisms, work practices, the relationships between system users, including authorities and communities of users, and threats to security exploiting social engineering techniques.

Our work lies between the technical aspects of information-flow security and the social aspects of user-centred security. More specifically, we are interested in information flow; however, the locus of this flow is now not within a computer system but within the inputs provided to it by its user. We are not considering the social aspects of human-computer interaction and security. Instead, the focus of our attention is cognitive processes that influence information flow from the human user into the computer system.

We build upon the generic user model (cognitive architecture) we developed in our work on usability [8]. It was developed from abstract cognitive principles, such as a user entering an interaction with knowledge of the task and its subsidiary goals. The cognitive architecture was later extended [17] to include an abstract specification, *interpretation*, of the pathways from device signals and environment objects to the user decision of what they mean (see Fig. 1). Incorporating such models of user behaviour into models of security is advocated by user-centred security [21]: e.g. Ka-Ping derives the guidelines (design rules) for secure interaction design from an informal user model [13].

Our cognitive architecture has proved of use for detecting various types of

systematic user errors in the context of usability and task completion [8,17]. Here our aim is to show that the behaviours emerging from this architecture also expose security problems and so facilitate the improvement of security aspects in user interaction design. To demonstrate this, we first informally discuss, from a security viewpoint, several examples of user error dealt with in our earlier work [8]. Then we consider an example of using the model checking tool SAL [14] to detect some confidentiality leaks emerging from our cognitive architecture and conditioned by the user interpretation of system prompts. More specifically, we consider security problems that may arise from the combination of user habits and (relative) positioning of input fields in authentication interfaces. The examples are small and intended to illustrate an approach and ideas that we believe are more generally applicable.

Summarising, the main contribution of this paper is the following:

- An investigation into the formal modelling of cognitive aspects of confidentiality leaks in interactive systems.
- An extension of our framework, developed for usability verification, to deal with the security problems in user interaction.
- An illustrative example of confidentiality leaks, caused by cognitive interpretation and detectable by model checking using our cognitive architecture.

Related work

Whilst conducted independently and in parallel, Beckert and Beuster's work [2] takes a similar approach to ours. They also develop a formal user model, and combine it with specifications of the application and the user's assumptions about that application to verify security properties of interactive systems. Their user modelling is based on the formalisation of an established methodology, GOMS [12], which is the core of their work. The modelling of user's assumptions partially coincides with our user interpretation. However, their "assumptions" model user choice between multiple plausible options, whereas our "interpretation" deals, in addition, with the user perception of interface objects depending on their shape, position, etc. Beckert and Beuster informally define three HCI security requirements, however, only one is formalised, whereas correctness properties in our framework also address the remaining two. It is also unclear whether they provide tool support for automatic verification. On the other hand, their methodology supports hierarchical models: an advantage when dealing with larger systems.

In the related area of safety-critical systems, Rushby *et al* [18] focus on mode errors and the ability of pilots to track mode changes. They formalise plausible mental models of systems and analyse them using the Murϕ verification tool. The mental models though are essentially abstracted system models; they do not rely upon structure provided by cognitive principles. Neither do they model user interpretation. Cerone *et al's* [7] CSP model of an air traffic control system includes controller behaviour. A model checker was used

to look for new behavioural patterns, missed by the analysis of experimental data. The classification stage in their model is similar to user interpretation.

Ka-Ping [13] gives a list of design rules, justified by an informal user model and tailored to increase security of interactive systems. As the rules are informal (many are probably too abstract to be formalised), there is no tool support for verifying whether designs obey them.

2 The Cognitive Architecture in SAL

Our cognitive architecture is a higher-order logic formalisation of abstract principles of cognition and specifies cognitively plausible behaviour [5]. The architecture specifies possible user behaviour (traces of actions) that can be justified in terms of specific results from the cognitive sciences. Real users can act outside this behaviour, about which the architecture says nothing. Its predictive power is bounded by the situations where people act according to the principles specified. The architecture allows one to investigate what happens if a person acts in such plausible ways. The behaviour defined is neither "correct" nor "incorrect". It could be either depending on the environment and task in question. We do not attempt to model the underlying neural architecture nor the higher level cognitive architecture such as information processing. Instead our model is an abstract specification, intended for ease of reasoning.

We rely upon cognitive principles that give a *knowledge level* description in the terms of Newell [16]. Their focus is on the goals and knowledge of a user. Our formalisation of the principles is based on the SAL model checking environment [14]. It provides a higher-order specification language and tools for analysing state machines specified as parametrised modules and composed either synchronously or asynchronously. The SAL notation we use is given in Table 1. We also use the usual notation for the conjunction, disjunction and set membership operators. A slightly simplified version of the SAL specification of a transition relation that defines our user model is given in Fig. 2, where predicates in italic are shorthands explained later on. Below we discuss the cognitive principles and the way they are reflected in the SAL specification (module `User`).

Non-determinism. In any situation, any one of several cognitively plausible behaviours might be taken. It cannot be assumed that any specific plausible behaviour will be the one that a person will follow. The SAL specification is a transition system. Non-determinism is represented by the non-deterministic choice, `[]`, between the named guarded commands (i.e. transitions). For example, *GoalCommit* in Fig. 2 is the name of a family of transitions indexed by `i`. Each guarded command in the specification describes an action that a user *could* plausibly make.

Mental versus physical actions. A user commits to taking an action in a way that cannot be revoked after a certain point. Once a signal has been sent

Table 1
A fragment of the SAL language

`x:T`	`x` has type `T`
`λ(x:T):e`	a function of `x` with the value `e`
`x′ = e`	an update: the new value of `x` is that of the expression `e`
`{x:T | p(x)}`	a subset of `T` such that the predicate `p(x)` holds
`a[i]`	the `i`-th element of the array `a`
`r.x`	the field `x` of the record `r`
`r WITH .x := e`	the record `r` with the field `x` replaced by the value of `e`
`g →` upd	if `g` is true then update according to `upd`
`c [] d`	non-deterministic choice between `c` and `d`
`[](i:T): c_i`	non-deterministic choice between the c_i with `i` in range `T`

from the brain to the motor system to take an action, it cannot be stopped even if the person becomes aware that it is wrong before the action is taken. Therefore, we model both *physical* and *mental* actions. Each physical action modelled is associated with an internal mental action that commits to taking it. In the specification, this is reflected by the pairings of guarded commands: *GoalCommit* – *GoalTrans*, and *ReactCommit* – *ReactTrans*. The first of the pair models committing to an action, the second actually doing it (see below).

User goals. A user enters an interaction with knowledge of the task and, in particular, task dependent sub-goals that must be discharged. These sub-goals might concern information that must be communicated to the device or items (such as bank cards) that must be inserted into the device. Given the opportunity, people may attempt to discharge such goals, even when the device is prompting for a different action. We model such knowledge as user goals which represent a pre-determined partial plan that has arisen from knowledge of the task in hand, independent of the environment in which that task is performed. No fixed order is assumed over how user goals will be discharged.

To see how this is modelled in SAL consider the guarded command *GoalTrans* for doing a user action that has been previously committed to:

$$\texttt{gcommit[i]} = \texttt{committed} \quad \rightarrow \quad \begin{array}{l} \texttt{gcommit}'\texttt{[i]} = \texttt{done}; \\ \texttt{gcomm}' = \texttt{FALSE}; \\ \mathit{GoalTransition}(i) \end{array}$$

The left-hand side of → is the guard of this command. It says that the rule will only activate if the associated action has already been committed to, as indicated by the `i`-th element of the local variable array `gcommit` holding value `committed`. If the rule is then non-deterministically chosen to fire, this value is changed to `done` and the boolean variable `gcomm` is set to false to indicate there are now no commitments to physical actions outstanding and the user model can select another goal. $\mathit{GoalTransition}(i)$ represents the state updates associated with this particular action i.

User goals are modelled as an array, `goals`, which is a parameter of the `User` module. The state space of the user model consists of three parts: input

```
TRANSITION
  [](i:GoalRange):  GoalCommit:
     gcommit[i] = ready ∧
     NOT(gcomm ∨ rcomm) ∧          →   gcommit'[i] = committed;
     finished = notf ∧                 gcomm' = TRUE
     goals[i].grd(in, mem, env)
[]
  [](i:ReactRange):  ReactCommit:
     rcommit[i] = ready ∧
     NOT(gcomm ∨ rcomm) ∧          →   rcommit'[i] = committed;
     finished = notf ∧                 rcomm' = TRUE
     react[i].grd(in, mem, env)
[]
  [](i:GoalRange):  GoalTrans:
                                       gcommit'[i] = done;
     gcommit[i] = committed    →       gcomm' = FALSE
                                       GoalTransition(i)
[]
  [](i:ReactRange):  ReactTrans:
                                       rcommit'[i] = ready;
     rcommit[i] = committed    →       rcomm' = FALSE
                                       ReactTransition(i)
[]
  Exit:
     PerceivedGoal(in, mem) ∧
     NOT(gcomm ∨ rcomm) ∧          →   finished' = ok
     finished = notf
[]
  Abort:
     NOT(EnabledGoals(in, mem, env)) ∧
     NOT(EnabledReact(in, mem, env)) ∧      finished' = IF Wait(in, mem)
     NOT(PerceivedGoal(in, mem)) ∧     →                THEN notf
     NOT(gcomm ∨ rcomm) ∧                               ELSE abort ENDIF
     finished = notf
[]
  Idle:
     finished = notf   →
```

Fig. 2. User model in SAL (simplified)

variable `in`, output variable `out`, and global variable (memory) `mem`; the environment is modelled by a global variable, `env`. All of these are specified using type variables and are instantiated for each concrete interactive system. Each goal is specified by a record with the fields `grd`, `tout`, `tmem` and `tenv`. The `grd` field is discussed below. The remaining fields are relations from old to new states that describe how two components of the user model state (outputs `out` and memory `mem`) and environment `env` are updated by discharging this

goal. These relations, provided when the generic user model is instantiated, are used to specify $GoalTransition(i)$ as follows:

```
out′ ∈ {x:Out | goals[i].tout(in,out,mem)(x)};
mem′ ∈ {x:Memory | goals[i].tmem(in,mem,out′)(x)};
env′ ∈ {x:Env | goals[i].tenv(in,mem,env)(x) ∧ possessions}
```

Since we are modelling the cognitive aspects of user actions, all three updates depend on the initial values of inputs (perceptions) and memory. In addition, each update depends on the old value of the component updated. The memory update also depends on the new value (`out′`) of the outputs, since we usually assume the user remembers the actions just taken. The update of `env` must also satisfy a generic relation, *possessions*. It specifies universal physical constraints on possessions and their value, linking the events of taking and giving up a possession item with the corresponding increase or decrease in the number (counter) of items possessed. For example, it specifies that if an item is not given up then the user still has it. The counters of possession items are modelled as environment components. We omit further details since, in this paper, possession properties are not used in any way.

If the guarded command for *committing* to a user goal, *GoalCommit*, fires, it switches the commit flag for goal `i` to `committed` thus enabling the *GoalTrans* command. The predicate `grd`, extracted from the `goals` parameter, specifies when there are opportunities to discharge this user goal. Because we assign `done` to the corresponding element of the array `gcommit` in the *GoalTrans* command, once fired the command below will not execute again. If the user model discharges a goal, it will not do so again without an additional reason such as a device prompt.

Reactive behaviour. Users may react to an external stimulus, doing the action suggested by the stimulus. For example, if a flashing light comes on a user might, if the light is noticed, react by inserting coins in an adjacent slot. Reactive actions are modelled by the pairing *ReactCommit* – *ReactTrans* in the same way as user goals but on different variables, e.g. parameter `react` of the `User` module rather than `goals`. $ReactTransition(i)$ is specified in the same way as $GoalTransition(i)$. The array element `rcommit[i]` is reassigned `ready` rather than `done`, once action `i` has been executed, since reactive actions, if prompted, *may* be repeated.

Goal based task completion. Users intermittently, but persistently, terminate interactions as soon as their main goal has been achieved [6], even if subsidiary tasks generated in achieving the main goal have not been completed. A cash-point example is a person walking away with the cash but leaving the card. In the SAL specification, a condition that the user perceives as the main goal of the interaction is represented by a parameter `PerceivedGoal` of the `User` module. Goal based completion is then modelled as the guarded command *Exit*, which simply states that, once the predicate `PerceivedGoal` becomes true and there are no commitments to user goals and/or reactive actions, the

user may complete the interaction. This action may still not be taken because the choice between enabled guarded commands is non-deterministic. Task completion is modelled by setting the local variable `finished` to `ok`, whereas the value `notf` means that the task is unfinished.

No-option based task termination. If there is no apparent action that a person can take that will help complete the task then the person may terminate the interaction. For example, if, on a ticket machine, the user wishes to buy a weekly season ticket, but the options presented include nothing about season tickets, then the person might give up, assuming the goal is not achievable.

In the guarded command *Abort*, the no-option condition is expressed as the negation of predicates `EnabledGoals` and `EnabledReact`. Note that, in such a case, a possible action that a person could take is to wait. However, they will only do so given some cognitively plausible reason such as a displayed "please wait" message. The waiting conditions are represented in the specification by predicate parameter `Wait`. If `Wait` is false, `finished` is set to `abort` to model a user giving up and terminating the task.

3 Verification of Security Aspects in User Interaction

In this section, we discuss examples of user error, focussing on the security aspects of interaction. We first introduce the properties to verify.

3.1 Correctness properties: usability and security

Previously, our approach dealt with two kinds of usability properties. First, we want to be sure that, in any possible system behaviour, the user's main goal of interaction (as they perceive it) is eventually achieved. Given our model's state space, this is written as the SAL assertion

$$\texttt{F}(\texttt{PerceivedGoal}(\texttt{in}, \texttt{mem})) \tag{1}$$

where `F` means 'eventually'. Second, in achieving a goal, subsidiary tasks are often generated that the user must complete to complete the task associated with their main goal. If the completion of the subsidiary tasks is represented as a predicate, `SecondaryGoal`, the required condition is (where `G` means 'always'):

$$\texttt{G}(\texttt{PerceivedGoal}(\texttt{in}, \texttt{mem}) \Rightarrow \texttt{F}(\texttt{SecondaryGoal}(\texttt{in}, \texttt{mem}, \texttt{env}))) \tag{2}$$

This states that the secondary goal is always eventually achieved once the perceived goal has been. Often secondary goals can be expressed as interaction invariants [8] which state that some property of the system state, that was perturbed to achieve the main goal, is restored. Previously, we viewed property (2) in terms of pure usability, applying it to, e.g. user possessions.

The verification of (2) can, however, also be used to detect security problems. Moreover, we will introduce a third kind of correctness property, relevant

to confidentiality leaks in user input. Intuitively, one would like to prevent such leaks in all system states, so we are aiming at a safety property. In terms of information-flow security [19], let us have, for simplicity, two confidentiality levels of user inputs, "high" and "low". A safety property that addresses some security aspects is that in no states do high inputs appear on a low channel. A boolean, `SecurityBreach`, represents system states that breach this. The property, stating that it is always true there is no security breach, is then:

(3) `G(NOT(SecurityBreach))`

We discuss how `SecurityBreach` is set to true, indicating breaches, in Section 4.

Note that neither of the first two correctness properties capture confidentiality leaks modelled as `SecurityBreach`. Property (1) is a usability property; the essential condition is a user achieving the main goal. The fact that this goal might be achieved by first making a mistake then undoing the erroneous action is irrelevant. However, with respect to security, undo is not good enough [13]: an erroneous action could already have leaked confidential information. Though checking property (2) can reveal some security problems related to, e.g. post-completion errors (see below), it is still a liveness property. As such, it does not require a system to satisfy the condition `SecondaryGoal` in all states, only at some point after the main goal has been achieved.

3.2 User error and security

Erroneous actions are the proximate cause of failure, since it was a particular action that caused the problem: e.g. a user entering data in the wrong field. To eliminate the problem, however, one must consider the ultimate causes of an error. In our framework, we consider situations where the ultimate causes are aspects (limitations) of human cognition that have not been addressed in the interface. An example is that a person enters data in a particular field because the interface design suggests it as appropriate for that data. In Hollnagel's terms [11] which distinguish between human error *phenotypes* (classes of erroneous actions) and *genotypes* (the underlying, e.g. psychological, cause), our cognitive architecture deals with genotypes. Since there is no evidence that security errors are conditioned by different cognitive causes to usability errors, our cognitive architecture can exhibit behaviours leading to security problems, even though it was developed without security concerns in mind. Some of these errors have the same cognitive causes as the usability errors we dealt with in our earlier work [8]. Next we discuss several types of user error, related to security but still detectable within the usability based approach represented by properties (1) and (2).

A persistent user error that emerges from the cognitive architecture is the post-completion error [6], where a user terminates an interaction with completion of subsidiary tasks outstanding. People have been found to make such errors even in lab conditions [6]. An example of this error, which is also a security breach, is when, with old cash machines, users persistently took cash

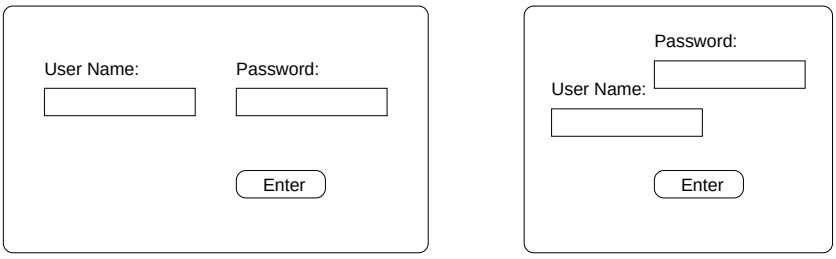

Fig. 3. Two layouts of an authentication interface

but left their bank card. Within our cognitive architecture, such behaviour emerges because of an action (*Exit*) that allows a user to stop once the goal has been achieved. Using our verification framework, this is detected by checking property (2). For this, `SecondaryGoal` would state that the total value of user possessions (bank cards included) in a state is the same as it was before the interaction. The formal verification of a similar example is described in [8].

Blandford and Rugg [4] give an example of an extant security breach caused by users forgetting to log out when moving away from an industrial printer, leaving it vulnerable to sabotage – e.g. by unauthorised users changing the printed message, etc. Being a case of the post-completion error, it can be detected by verifying property (2) with the appropriately chosen `SecondaryGoal`.

Previously [17] we also considered user error due to the shape-induced confusion over the meaning of interface prompts. The example was that of a user attempting to top-up a phone card using an ATM. We showed how model checking, based on our cognitive architecture, can identify user confusion as to which of two numbers, phone number or top-up card number, is requested. The property checked was of type (1), i.e. whether the user achieves the main goal. User confusion in a similar situation can also result in confidentiality leaks. For example, asked to enter a card number, a person might be confused whether the number requested is that of a bank card or a phone card. If a bank card number is entered when the interface prompts for a top-up card number, the input might be displayed which is a security breach. This problem would also be detected by analysing why the user could not achieve the main goal.

4 A Case Study: Authentication Interface

In this section, we extend our previous work and investigate how other security problems, not considered in that work, can be detected using our cognitive architecture formalised in SAL. In particular, we show how user habits in combination with some designs, can lead to the incorrect interpretation of interface prompts, resulting in the leakage of confidential information. To determine whether such leakages are possible, we introduce into our framework a new entity, generic module `tester`. This module is instantiated by providing a collection of channels and a high security value. The instantiated module then checks whether this value can appear on one of the low security channels.

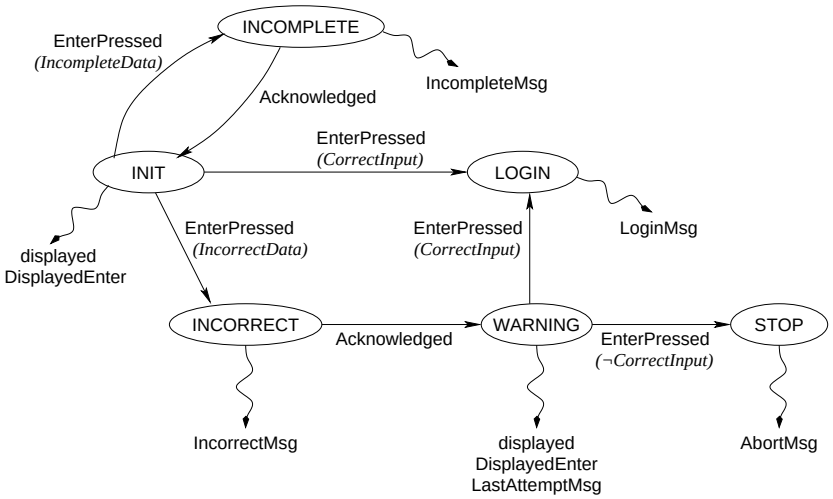

Fig. 4. Authentication procedure

4.1 An Authentication Interface

Our example considers a common problem concerning an authentication step present in various everyday interactive systems, e.g. internet banking. Before any transaction, users must establish their identities by providing a user name and a login password. The system checks whether the provided password is the same as the one associated with the provided user name and stored in the system's database. On the surface, one could expect the design of an authentication interface to be simple, e.g. like the one in Fig. 3(left). In reality, the situation is more complicated. The sizes of interface windows in internet banking systems are not fixed; users might change them at any time. This means that the layout of input fields is determined by an algorithm. Depending on this algorithm, the layout shown in Fig. 3(right) is possible when the window size is reduced. We will argue that the two interfaces are not equally secure and will show how confidentiality leaks in the second one can be detected using our verification framework.

We assume that a high security channel is associated with login passwords and a low security channel with user names. This could mean, e.g., that the text entered into the name box is echoed on the screen whereas an entry into the password box is hidden. The data is sent whenever the users press the `Enter` button. The operation of the authentication mechanism is illustrated by a finite state machine in Fig. 4 (false outputs are omitted). We distinguish two cases of incorrect input represented by the transitions *IncompleteData* and *IncorrectData*. The authentication procedure moves into the `INCOMPLETE` state when `Enter` is pressed and either a user name or a password is missing from the input boxes. An appropriate error message is displayed by the interface, and no other options for the user are given until the message is acknowledged. Once the user acknowledges it, the authentication procedure returns to the `INIT` state. The transition *IncorrectData* represents the case when both a user name and a password are provided but some of this data is incorrect. Upon

acknowledgment, the procedure moves into the `WARNING` state in this case. The idea is that, for security reasons, a single authentication attempt with incorrect data is allowed before the authentication procedure aborts the interaction (`STOP` state). Finally, authentication succeeds if the user provides correct data, represented by the `LOGIN` state reachable from either the `INIT` or `WARNING` state.

The SAL specification of the authentication procedure is a direct translation of the diagram in Fig. 4. The two input boxes are modelled as the type `Inbox = {A, B}`. Each box has a number of attributes: position, security level, "visibility", label, text entered and text displayed, modelled as arrays with the range `Inbox`. Thus, `position[j]` is a record with the coordinate fields `x` and `y`, denoting the top-left corner of box `j`. Its width and height are represented by the constants `dx` and `dy`. The security level `level[j]` is either `Low` or `High`; `displayed[j]` indicates whether `j` is displayed (visible) or not. The label `label[j]` is a value of type `{NameLabel, PasswordLabel}`. Finally, `value[j]` and `display[j]` represent, respectively, text entered and text displayed, which can differ when the entered text is hidden. The array `value` and booleans `EnterPressed` and `Acknowledged` are the inputs of the authentication procedure, whereas `position`, `displayed`, `label`, `display` with booleans `DisplayedEnter`, `IncompleteMsg`, `IncorrectMsg`, `LastAttemptMsg`, `LoginMsg`, and `AbortMsg` are its outputs.

4.2 A User Model

Now we instantiate the generic module `User` for the authentication task. We start by specifying the state space of the concrete user model. For each input box `j`, we assume that a person either sees it or not, and perceives its label and the text displayed, represented by `seen[j]`, `label[j]` and `value[j]`, respectively. The perception of whether the `Enter` button is active is denoted by `EnterActive`. The person also perceives whether an error, warning or authentication message is given, denoted by `ErrorMsg`, `WarningMsg` and `LoginMsg`. Variables `InputName` and `InputPass` denote the perception of which of the two boxes prompts for the user name and which for the password. Finally, `name` and `password` denote the values the person perceives as a user name and password. All these components form a record type, `In`, which is used to instantiate the corresponding type variable in `User`.

Next, we specify variables related to the actions users might take. The text typed into box `j` is represented by `value[j]`. The booleans `EnterPressed` and `Acknowledged` denote whether the `Enter` button and a button to acknowledge messages are pressed. These components form a record type, `Out`. We assume users remember their user name, `name`, and login password, `password`. They also remember whether they already typed information into box `j`, denoted `entered[j]` (reset to false when an error message is acknowledged), and keep track of whether there was a failure to authenticate, denoted `failed`. These form a record type, `Mem`, which also records, in a component of the type `Out`,

the actions taken in the previous step. The reality surrounding our system is given by a record type, `Env`. It includes the user name, `name`, and the correct password, `password`.

We assume that user knowledge of authentication includes the need to communicate (1) user name and (2) login password. This knowledge is specified as user goals (elements of array `goals`) instantiated by giving the action guard and the updates to the output component. For the goal of communicating the user name, the guard is that an input box, regarded as the name box, is seen. The output action is to enter the name as the user perceives it:

```
grd := λ(in,mem,env):  in.seen[in.InputName]
tout := λ(in,out0,mem):λ(out):  out = Default(out0.value)
                                WITH .value[in.InputName] := in.name
```

where `Default(x)` is a record with the field `value` set to `x` and all other fields set to false thus asserting that nothing else is done. The memory update (omitted) simply records the action taken. As an example, we will specify the most complicated memory update below. The action of communicating the login password is specified similarly. Since the environment updates change nothing (in all the actions), they are omitted here.

We assume that the user can *react* to the active `Enter` button by pressing it. For this to happen, the user must not have the recollection of a failure to authenticate. Alternatively, if there was such a failure, we expect the user to be more careful and press `Enter` only when both input boxes were filled in:

```
grd := λ(in,mem,env):  in.EnterActive ∧ (NOT(mem.failed) ∨
          (mem.entered[in.InputName] ∧ mem.entered[in.InputPass]))
tout := λ(in,out0,mem):λ(out):
           out = Default(out0.value) WITH .EnterPressed := TRUE
```

We also assume that the user can acknowledge error messages. This only happens when the message is interpreted as an error signal. The acknowledgment must also not have occurred, as indicated by the memory, in the previous step. By acknowledging the error message, the user records in the memory the fact of a failed authentication attempt, and "forgets" previously typing data into the input boxes (since the data was rejected), formally specified as:

```
grd := λ(in,mem,env):  in.ErrorMsg ∧ NOT(mem.out.Acknowledged)
tout := λ(in,out0,mem):λ(out):
           out = Default(out0.value) WITH .Acknowledged := TRUE
tmem := λ(in,mem0,out):λ(mem):  mem = mem0 WITH .failed := TRUE
           WITH .entered := [[j:Inbox] FALSE] WITH .out := out
```

As discussed earlier, the need to communicate the name and password is modelled as user goals. However, it is plausible that the user makes an error when trying to achieve those goals, e.g., enters a wrong password or presses `Enter` when some box is empty. Errors can also occur due to user habits; relying on previous experience, the user might expect the input box for the name to precede that for the password. In such cases, once the error

message has been acknowledged, the system prompts for a new authentication attempt. We assume that the user will respond to this prompt. The response is modelled as two reactive actions. In the case of the password, the action guard is that an input box is seen (as for the corresponding user goal) and the password was not entered, as indicated by the memory, in the previous step. The output action is the same as for the corresponding user goal. Finally, the memory update records the fact of entering the password:

```
grd := λ(in,mem,env):  in.seen[in.InputPass] ∧
                       NOT(mem.entered[in.InputPass]) ∧ mem.failed
tout := λ(in,out0,mem):λ(out):  out = Default(out0.value)
                          WITH .value[in.InputPass] := in.password
tmem := λ(in,mem0,out):λ(mem):  mem = mem0
          WITH .entered[in.InputPass] := TRUE WITH .out := out
```

The reactive action for entering the name is analogous to the one above.

Goal and wait predicates are the last parameters used to instantiate the `User` module. The display of `LoginMsg` confirms authentication which is the main goal. We also assume that there are no signals that a user would perceive as a suggestion to wait. These predicates are specified as follows:

```
PerceivedGoal = λ(in,mem): in.LoginMsg
Wait = λ(in,mem): FALSE
```

Finally, the user model for the authentication task, `UserAuthenticate`, is defined by instantiating the generic user model with the parameters (goals, reactive actions, perceived goal and wait condition) just defined.

4.3 User Interpretation

So far we have specified an authentication interface and have developed a formal model of its user. As in reality, the state spaces of the two specifications are distinct. The changing interface state is first attended to then interpreted by the user. Next we specify this interpretation, thus connecting distinct state spaces. The specification is given as a new SAL module, `interpretation`. The module, being a connector, has input variables that are the output variables of the interface, and an output variable that is the input (perception) component of the `UserAuthenticate` module (record `in`).

In the authentication task, the crucial aspect of user interpretation is the perception of the meaning (function) of the two input boxes. Their function is indicated by labels, however, we assume that people may not pay sufficient attention to the labels. Instead, the user might assume the name box comes first. The perception of precedence depends on the layout (coordinates) of boxes in the interface window. Formally, we define the condition when the input box `i` precedes `j` as follows (`pos` is an array of coordinates):

```
precedes(i,j,pos) = (pos[i].x + dx < pos[j].x ∧ pos[i].y ≤ pos[j].y)
                  ∨ (pos[i].x ≤ pos[j].x ∧ pos[i].y + dy < pos[j].y)
```

Intuitively, this means that `j` is placed to the right and to the bottom of `i`. Thus, the name box in the left interface in Fig. 3 precedes the password box, whereas neither of the boxes in the interface on the right precedes the other.

Now we formally define the user interpretation of the function of input boxes, depending on their layout and labelling. We distinguish three cases. First, the user might judge the function of input boxes from their labels:

```
ByLabel(l,x) = ∃(i,j):  l[i] = NameLabel ∧ l[j] = PasswordLabel ∧
                        x.InputName = i ∧ x.InputPass = j
```

Second, if `i` precedes `j` then `i` is perceived as a name and `j` as password box:

```
ByPrecedence(pos,x) = ∃(i,j):  precedes(i,j,pos) ∧
                               x.InputName = i ∧ x.InputPass = j
```

Finally, the user might get confused. This is possible when neither of the input boxes precedes the other or their labels are the same; the judgment about the function of the boxes is random in this case:

```
Random(pos,l,x) = (l[A] = l[B] ∨ ∀(i,j):NOT(precedes(i,j,pos))) ∧
                  x.InputName ≠ x.InputPass
```

User interpretation is modelled as a SAL definition which allows one to describe system invariants. Intuitively, this means that the left-hand side of an equation is updated whenever the value of the right-hand side changes. We assume that, once the user makes a mental commitment to a goal or reactive action, the interpretation of the interface outputs does not change until the associated physical action is performed. If there is no commitment, the user directly perceives the `Enter` button, the displayed input boxes with their labels and displayed text, and the interface messages. Hence the first seven conjuncts in Fig. 5 simply rename the interface variables to the corresponding fields of the record `in`.

For the user name and password, the user relies on the memory unless a warning message is displayed. If so, we expect the user to be careful enough to provide the correct values. For simplicity, here we do not consider how this is actually achieved (perhaps they are taken from a notebook), assuming that the values from the environment specification are used.

As explained earlier, the perception of which of the two boxes is for the names and which for the passwords is more complicated; the results of this perception are assigned to `InputName` and `InputLabel`, respectively. We assume that, upon receiving a warning message, the user becomes more careful and interprets the input boxes by their labels. Otherwise, if there are major changes in the layout of the boxes, the interpretation is an arbitrary choice between the three cases defined above. If there are no major changes, the interpretation of the boxes is the same as in the previous step. The auxiliary variables `s`, `p` and `l` are not intended to represent aspects of cognition. Intuitively, they, and the related `TRANSITION` section, are used to store the

```
DEFINITION in ∈ { x:In |
  IF NOT(gcomm ∨ rcomm) THEN
    x.EnterActive = DisplayedEnter ∧
    x.seen = displayed ∧ x.label = label ∧ x.value = display ∧
    x.ErrorMsg = (IncompleteMsg ∨ IncorrectMsg) ∧
    x.WarningMsg = LastAttemptMsg ∧ x.LoginMsg = LoginMsg ∧
    x.name = IF x.WarningMsg THEN env.name ELSIF mem.name ENDIF ∧
    x.password = IF x.WarningMsg THEN env.password
                 ELSIF mem.password ENDIF ∧
    IF x.WarningMsg THEN ByLabel(label,x)
    ELSIF MajorChanges(p,position,l,label) THEN ByLabel(label,x)
             ∨ ByPrecedence(position,x) ∨ Random(position,label,x)
    ELSE x.InputName = s.InputName ∧ x.InputPass = s.InputPass ENDIF
  ELSE x = s ENDIF }
TRANSITION
    s′ = in; p′ = position; l′ = label
```

Fig. 5. User interpretation in SAL

previous interpretation which allows specifying that user interpretation does not change. Finally, "major" changes mean changes in the precedence of input boxes or any label change:

```
MajorChanges(pos0,pos,l0,l) =
     ∃(i,j):  precedes(i,j,pos0) ≠ precedes(i,j,pos) ∨ l0[i] ≠ l[i]
```

Admittedly, this attempt to formally specify how the user perceives input boxes already hints at potential problems, even before the actual verification. However, we aim at developing a generic model of interpretation which would turn the specification process into a simple instantiation of the generic model.

4.4 Verification

We have specified an authentication interface and its user model. Now the correctness properties of this interactive system can be analysed. We start from the interface with no constraints on the layout of the input boxes (other than that they do not intersect). The usability property (1), the user eventually achieving the perceived goal, is satisfied by the interactive system. Next we proceed with the analysis of security aspects of the system.

Even though security properties for each concrete system can be specified separately, we prefer to take a generic approach as with the user model itself. We thus introduce a generic module, `tester`. The idea is that the module, composed with an interactive system, monitors the communication between the device and the user. When security is breached, it sets the variable `SecurityBreach` to true. What security aspects are monitored is determined by the instantiation of the module. It has three parameters. The type variable `Chan` represents the communication channels. The predicate `filter` specifies which of the channels are monitored. Finally, `test` denotes security sensitive

data. When this data appears on a monitored channel, `SecurityBreach` is set to true. The transitions of the module are the following family of commands:

```
[](j:Chan):  filter(j) ∧ value[j] = test  →  SecurityBreach′ = TRUE
```

For our authentication task, `Chan` is instantiated to the input boxes (type `Inbox`). The security sensitive data is the actual user password `env.password`. Finally, the channels monitored are low security channels:

```
filter(j) = (level[j] = Low)
```

With this instantiation of `tester`, we check property (3); the verification fails. The counterexample produced by SAL indicates that the user enters the password into the name box. The analysis of the specifications reveals that this counterexample occurs because neither of the boxes precedes the other which confuses the user.

Why was this confusion not detected when verifying property (1)? The answer is that it does not prevent the user from achieving the main goal, authenticating their identity. Our user model is "smart" enough to recover from the mistake made due to this confusion and, after receiving a warning message, to provide the required information according to the labels of the input boxes. Such a recovery, however, is not good enough from the security point of view, since the mistake could have already breached security and undoing or redoing a wrong action cannot undo the consequences of this breach in most cases, which is detected by the failure to establish property (3).

How can we avoid this security breach? Since the confusion leading to it is caused by the layout of the input boxes, the solution is to display them as in Fig. 3(left). However, one must be careful even with such a layout. If the password box preceded the name box, people might enter their password into the name box, due to their habits rather than confusion. Again, this security breach was detected by verifying (3). The latter holds only when the layout of the input boxes is such that `precedes(InputName,InputPass,position)` is true.

5 Conclusion

In previous work, we assumed that usability verification is enough to establish user-centred correctness of interactive systems. This is not true within security- or mission-critical contexts where it is possible to achieve the main goal while exposing ourselves to various security or safety risks. Here we addressed one aspect of interactive systems security – information-flow security. We discussed how security breaches can be detected using our earlier framework. The main focus, however, was an extension of that framework to address security aspects more directly and link them to the usability properties. For this, we introduced into our framework a generic tester module which monitors information flow between the user and the device and registers security breaches in it. Using the tester, we added to our verification approach a cor-

rectness property which captures some security aspects of interactive systems.

To illustrate these extensions, we considered a simple authentication interface. We showed how the layout of input fields combined with user habits can influence the user (mis)interpretation of interface prompts, possibly leading to confidentiality leaks (see our SAL specs at http://www.dcs.qmul.ac.uk/~rimvydas/usermodel/fmis06.zip). We demonstrated how these leaks are detected using the SAL verification tools, and how the analysis of the SAL counterexamples can help in eradicating cognitively susceptible interface designs. The SAL environment was primarily chosen because of its support for higher-order specifications. This is necessary for developing a generic cognitive architecture as ours.

The user interpretation stage was introduced into our framework from general considerations. We previously showed how modelling user interpretation allows us to detect usability problems due to the shape-induced confusion over device prompts [17]. Here we showed that similar ideas apply in the context of security properties and their dependence on the layout of input fields. Finally, we also considered user habits, which we had not dealt with before.

Human factors in the security context have been considered before [1,13]. The novelty of our approach is dealing with the cognitive aspects of security in a completely formal way, making them amenable to automatic verification. Moreover, our cognitive architecture could be used to prove generic results on, e.g., design rules for security, using its formalisation within the HOL prover.

References

[1] Adams, A., and M. A. Sasse, *Users are not the enemy*, CACM **42**(12) (1999), 41–46.

[2] Beckert, B., and G. Beuster, *A method for formalizing, analyzing, and verifying secure user interfaces*, in press: Proc. ICFEM 2006, LNCS, Springer, 2006.

[3] Bell, D. E., and L. J. La Padula, *Secure computer system: Unified exposition and Multics interpretation*, Tech. Rep. MTR-2997, MITRE Corp., MA, 1976.

[4] Blandford, A., and G. Rugg, *A case study on integrating contextual information with usability evaluation*, Int. J. Human-Computer Studies **57**(1) (2002), 75–99.

[5] Butterworth, R., A. Blandford, and D. Duke, *Demonstrating the cognitive plausibility of interactive systems*, Form. Asp. Computing **12** (2000), 237–259.

[6] Byrne, M. D., and S. Bovair, *A working memory model of a common procedural error*, Cognitive Science **21**(1) (1997), 31–61.

[7] Cerone, A., P. A. Lindsay, and S. Connelly, *Formal analysis of human-computer interaction using model-checking*, in: Proc. SEFM 2005, IEEE Press, 352–362.

[8] Curzon, P., and A. E. Blandford, *Detecting multiple classes of user errors*, in: R. Little, and L. Nigay, eds., Proc. EHCI 2001, vol. 2254 of LNCS, Springer, 2001, 57–71.

[9] Denning, D. E., and P. J. Denning, *Certification of programs for secure information flow*, CACM **20**(7) (1977), 504–513.

[10] Goguen, J. A, and J. Meseguer, *Security policies and security models*, in: Proc. IEEE Symp. on Security and Privacy, Apr. 1982, IEEE Press, 1982, 11–20.

[11] Hollnagel, E., "Cognitive Reliability and Error Analysis Method," Elsevier, 1998.

[12] John, B. E., and D. E. Kieras, *The GOMS family of user interface analysis techniques: Comparison and contrast*, ACM Trans. CHI **3**(4) (1996), 320–351.

[13] Ka-Ping, Y., *User interaction design for secure systems*, in: R. Deng, et al., eds., Proc. ICICS 2002, vol. 2513 of LNCS, Springer-Verlag, 2002, 278–290.

[14] de Moura, L., S. Owre, H. Ruess, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari, *SAL 2*, in: R. Alur, and D.A. Peled, eds., Computer Aided Verification: CAV 2004, vol. 3114 of LNCS, Springer-Verlag, 2004, 496–500.

[15] Myers, A. C., *JFlow: Practical mostly-static information flow control*, in: Proc. of ACM Symposium on Principles of Programming Languages, 1999, 228–241.

[16] Newell, A., "Unified Theories of Cognition," Harvard University Press, 1990.

[17] Rukšėnas, R., P. Curzon, J. Back, and A. Blandford, *Formal Modelling of Cognitive Interpretation*, in press: Proc. DSVIS 2006, LNCS, Springer, 2006.

[18] Rushby, J., *Analyzing cockpit interfaces using formal methods*, Electronic Notes in Theoretical Computer Science **43** (2001).

[19] Sabelfeld, A., and A. C. Myers, *Language-based information-flow security*, IEEE Journal on Selected Areas in Communications **21**(1) (2003), 1–15.

[20] Volpano, D., G. Smith, and C. Irvine, *A sound type system for secure flow analysis*, Journal of Computer Security **4**(3) (1996), 167–187.

[21] Zurko, M. E., *User-centered security: Stepping up to the grand challenge*, in: Proc. ACSAC 2005, IEEE Press, 2005, 187–202.

Refinement: a constructive approach to formal software design for a secure e-voting interface

Dominique Cansell [2]

Université de Metz,
France

J Paul Gibson [1,3]

Computer Science Department,
NUI Maynooth,
Ireland

Dominique Méry [4]

Université Henri Poincaré,
Nancy,
France

Abstract

Electronic voting machines have complex requirements. These machines should be developed following best practice with regards to the engineering of critical systems. The correctness and security of these systems is critical because an insecure system could be open to attack, potentially leading to an election returning an incorrect result or an election not being able to return any result. In the worst case scenario an incorrect result is returned — perhaps due to malicious intent — and this is not detected. We demonstrate that an incorrect interface is a major security threat and show the use of the formal method B in guaranteeing simple safety properties of the voting interface of a voting machine implementing a common variation of the single transferable vote (STV) election process. The interface properties we examine are concerned with the collection of only *valid* votes. Using the B-method, we apply an incremental refinement approach to verifying a sequence of designs of the interface for the collection and storage of votes, which we prove to be correct with respect to the simple requirement that only valid votes can be collected.

Key words: Formal Verification, refinement, formal specification, interface design, e-government

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

1 Introduction

The problem of electronic vote counting (or tabulation) involves a wide range of issues, requiring expertise in science, engineering and technology. As such, it provides a good challenge for the application of formal methods.

1.1 E-voting: background and motivation

Applying state-of-the-art computer and information technology to "modernise" the voting process has the potential to make improvements over the existing paper (or mechanical) systems; but it also introduces new concerns with respect to secrecy, accuracy and security [8]. The debate over the advantages and disadvantages of e-voting is not a new one; and recent use of such systems in actual elections has led to their analysis from a number of viewpoints: usability [9], trustworthiness and safety criticality [12], and risks and threats [15].

Despite ungoing uncertainty over the trustworthiness of these systems — which is the major disadvantage associated with them — many countries have chosen to adopt e-voting.

In a recent paper, Kocher and Schneier [10] conclude by stating: "The threats are real, making openness and verifiability critical to election security." The formal methods community is experienced in chasing technological change in software engineering: and this paper proposes that, in general, already existing formal techniques can help to alleviate many [5] of the verification problems that the adoption of new e-voting technologies can introduce. For the specific modelling and verification in our study we chose to use the B method.

1.2 The B Method

B is a method [1] for specifying, designing and coding software systems. The concept of refinement [3] is the key notion for developing B models of (software) systems in an incremental way. B models are accompanied by mathematical proofs that justify them. We start from an abstract model and each subsequent model is a refinement of the previous one. Proofs of properties of B models help to convince the user (designer or specifier) that the (software) system is correct, since they demonstrate that the behavior of the last, and most

[1] Thanks to the NUIM and CNRS who supported Dr Gibson's Sabbatical leave.
[2] Email: Dominique.Cansell@loria.fr
[3] Email: pgibson@cs.nuim.ie
[4] Email: Dominique.Mery@loria.fr
[5] We do not expect that all of the problems can be addressed completely by the use of formal methods. For example, problems of human error and those posed by malicious intent are very different in nature, but can both arise from simple design flaws. Our focus is on verification of design steps.

concrete, system (software) respects the behaviour of the first, most abstract model (which we assume has already been validated).

1.3 E-voting: formal methods, correctness and security

We propose to construct a formal, mathematical model of the e-voting problem in B. We argue that no e-voting system can be considered "safe" until the requirements of these systems are better defined and introduce refinement as a method for supporting the correct design and implementation of "safe e-voting systems". In this paper we concentrate on the front-end of the e-voting system (the user interface for the voters).

Our view is that the collection and tabulation must be developed formally: human error in the development of the collection and tabulation software might have considerably more serious consequences than such errors in the manual system. We argue that even a minor design flaw in the way in which votes are collected and passed on for tabulation can lead to security weaknesses that could be exploited by an election saboteur.

2 Critical system development: formality and security

To motivate the use of formal methods, we argue that e-voting is at least *mission critical* and may, in some circumstances, be considered as *safety critical* [12]. For our argument we consider a "worst case scenario", where the system fails to elect the correct candidates without the failure being identified. There is no meaningful way of equating this with a financial cost but its potential negative impact on the well-being of individuals and society is great. Thus, we must consider e-voting systems to be (at least) *mission critical* and we advocate the use of formal methods in their development as their application should ensure that the likelihood of failures due to modelling errors during design is reduced.

¿From a technological viewpoint we know that system (and interface) design has an important role in security assurance. Mercuri [13] addresses the theme of quality in the process of engineering security: "By encouraging artistry and applying craftsmanship to our security problems, viable solutions will emerge. One way of starting this process is by defining computer security with respect to need." This supports our view that one must start with a simple model of the security needs and refine that model, during design, towards a correct implementation. For this reason we chose a simple security requirement — that only *valid votes* can be found *in the system* — and start our formal development from there.

3 Valid Votes: a STV case study

The Single Transferable Vote (STV) model on which we build this case study is regarded as a good democratic election process. However, it incorporates a complex, not necessarily determinisitic, tabulation (counting) procedure. In 2003, Farrell and McAllister [6] reported on how a subtle change in the implementation of the STV rules can lead to major changes in the results returned.

In this paper, the counting algorithm is not developed formally. However, a brief overview of the tabulation process will help us to develop our claim that it is critical that there is a formal verification of the property that only valid votes are counted.

3.1 Overview of the typical counting algorithm

During an election, a candidate is elected if the number of votes they have is greater or equal to the *quota*. This quota, Q say, is a function of: the number of valid votes, V say, and the number of seats available for election, S say. It is usually defined by the equation: $Q = 1 + \frac{V}{S+1}$.

We note that the quota cannot be calculated without knowing the number of *valid* votes, and so its correct calculation is dependent on the notion of *validity* being correctly implemented. This notion is non-trivial as the STV election process allows voters to register support for more than one single candidate, by placing candidates in a preferred order. Thus on each vote, a candidate may or may not have an associated preference.

Informally, a vote is considered valid if and only if it shows a unique first preference. The means of specifying this property depends, of course, on a notation for representing a vote. Consider a constituency where there are three candidates: `A`, `B` and `C`, say. We could choose to represent a vote as a string of characters taken from the alphabet $\{A, B, C\}$. In such a string, we naturally interpret a character *ch* at index i in the vote string as stating that the ith preference of that particular vote is the candidate *ch*. Now, we can define a valid vote in this constituency by explicitly identifying the set of valid vote strings, for example:

$$\{A, B, C, AB, AC, BA, BC, CA, CB, ABC, ACB, BAC, BCA, CAB, CBA\}.$$

In such a definition we preclude, for example, the following strings from being considered as representations of valid votes:

- The empty string — correctly excluded, we would argue, as there is no first preference.
- The string AA — excluded, perhaps, because the candidate with the first preference has another associated preference. (We note that this was not explicit in the original informal definition, and it would be normal for this interpretation to be validated through additional discussion of this require-

ment with the customer.)

- The string $ABBC$ — excluded, perhaps, because the number of preferences cannot be more than the number of candidates. (We note, again, that this was not explicit in the original informal definition.)

Of course, it is better [6] to define the notion of validity as a generic boolean function that takes any string of characters and decides on its validity, without having to explicitly construct all the members of the valid vote set. An even better approach, as taken with our abstract B model is to specify the set of all valid votes and to use refinement to be sure that a vote belongs to this set without having to actually construct it (as it exists in the abstraction).

3.2 Requirements for valid votes

Within any voting system, changing the definition of a valid vote can have major consequences for the election process and the results returned. In any STV system, for example, there is a major difference between *allowing* and *requiring* a voter to place all candidates in a preference order.

In our model, it is required that a vote is considered valid if and only if it shows a unique first preference. This is a simple requirement, but one which can cause problems if it is not treated formally.

Clearly, if invalid votes manage to get passed to the tabulation process (to be counted) then there is a risk that this could break the counting process. For example, it would not be unreasonable to suggest that some of the tabulation methods make the assumption that the votes being counted are valid. However, without some degree of formal verification it is also likely that an invalid vote could — by accident — be counted and that this could lead to an incorrect result, a run-time error, or non-termination. Consequently, this weakness could also be exploited by an attacker to deliberately manipulate the election process.

Such a potential attack is similar to those mentioned in [14] where the security of votes stored in memory is addressed. In particular, the use of Trojan code to exploit vote data that has been tampered with is shown to be a real threat that requires elaborate schemes for the secure storage of votes. In most e-voting systems, there is a clear interface between the storage of votes and the input of votes. We argued that the same degree of care must be taken in designing the vote interface to ensure that Trojan code cannot be used to exploit the input of invalid votes.

[6] Simple, naive, construction (listing) of the complete valid vote set is infeasible for elections involving even a moderate number of candidates.

3.3 Validating votes in a typical implementation architecture

Typically, a voting machine has a simple, classic, layered architecture. We propose a generic, abstract model [7] in order to illustrate the need for formality in the processing of votes electronically:

- An *interface* facilitates the voter to input their preferences.
- A *store* records the preferences of all votes that have been input.
- A *tabulator* takes all the votes from the *store* and calculates the result.

We argue that an invalid vote in the tabulation process can compromise the security of the whole election. Thus, for an e-voting system to be considered secure, it is necessary that the tabulator does not process invalid votes; and so the store cannot transfer invalid votes into the tabulator. Consequently, it may be necessary to show that the store cannot receive invalid votes from the interface. There are clearly a number of design decisions that need to be taken with respect to the implementation of this simple architecture; and it is the responsibility of the designers to verify that their designs are correct with respect to the validity requirement.

4 Informal software design for vote validity

In this section we comment on typical design decisions that arise during implementation of our generic architecture. We focus, for simplicity, on the first layer of our architecture: the *interface*. Simple analysis of the requirement for *only valid votes in the tabulation process* could lead one to designing a machine where the interface layer takes responsibility for guaranteeing that voters record only valid votes and rely on a secure store. In all the following designs we refer to buttons that the user can press and the information that is displayed to the voter. We abstract away from details of how these buttons and displays are implemented.

4.1 The rapid prototyping approach

We wish to verify that a given interface design implements the requirement that no invalid vote can be sent to the store. A standard technique for carrying out such a verification is to incrementally add rigour to the process, structuring the verification process in a number of layers. A typical 3-layer approach, which we used for the purpose of our study, is:

- Run some voting scenarios through the natural language description and check, by hand, that there are no obvious examples of a scenario that leads to the requirement not being met.
- Prototype an executable model of the design and test this executable model as thoroughly as possible.

[7] We deliberately choose not to model a specific machine.

- Formalise the prototype model so that more formal model-checking or theorem-proving can be used to show that the design is *correct*.

In practice, with simple designs such as a simple vote interface, developers often see no need to progress to layer 3. In the designs that follow in the remainder of this section, we follow a rapid-prototyping approach that never progresses to the 3rd layer of rigour. In Section 5 we show how the B method and tools can be used to fully support layer 3 in this verification process.

In this section we concentrate on the validity of a single vote. We note that the validity of a complete set of votes will require that each individual vote is valid. The validity of individual votes will be a necessary, but perhaps not sufficient condition for the validity of all votes.

Before we consider verifying the property that the system contains only valid votes, it is necessary to formulate what we mean by a valid vote in a way that the design verification process can be automated. This means that, in our chosen modelling language, we have to choose a means of specifying the property that needs to be checked.

Consider the Java code[8] which models the `Vote` (of a single voter) as an array of "Candidates".

```
public class Vote{
int numberofCandidates;
int preferences [];
// Vote - construct empty Vote with no preferences
public Vote(int numCs){
    numberofCandidates = numCs;
    preferences = new int [numberofCandidates];
    for (int i = 0; i<numberofCandidates; i++) preferences[i] = 0;
}//END Vote Constructor
// ...
}// END CLASS Vote
```

We now formally specify a safety[9] property that defines a valid vote as a `boolean` method of the `Vote` class.

Of course, this invariant property could be modelled using *design-by-contract* [4] language and tool support. However, in this initial case study we choose to use only fundamental concepts that are part of the core programming language: we believe that little, if any, current voting software incorporates more rigorous design by contract methods.

[8] Starting with simple Java models helps us to identify the main issues with respect to interface evolution and refinement.

[9] The use of the word 'safety' here does not mean that the property is "safety critical" in the classic sense that lives may be at risk, for example. The term is taken from the formal methods community where it is used to refer to an invariant property of the system that must be true all the time otherwise system behaviour cannot guaranteed to be correct. It may be a safety property of the system but that does not necessarily make the system safety critical.

```
// isValid ****************************************************
// The safety property to ensure the validity of a Vote
// A vote is valid iff there is a unique 1st preference recorded
public boolean isValid(){
    int numberof1s =0;
    for (int i = 0; i<numberofCandidates; i++)
      if (preferences[i]==1) numberof1s++;
    return (numberof1s==1);
}// END Vote.isValid
```

Given the formal (constructive) specification of a valid vote (in Java) we can now proceed to the design of the first component of our generic architecture: the *interface*. The goal is to offer alternative interface designs and to verify the correctness, or otherwise, of each.

4.2 Design1: the simplest interface

We propose the following design for verification:

> "Every candidate is associated with a *candidate button*. Beside each candidate button, information is displayed to show the preference that is associated with that candidate. Initially, at the beginning of each vote, no preferences are displayed. When a voter presses a candidate button (for the 1st time) then it is assumed that the voter selected that candidate to be their first preference, and the number 1 is displayed beside that candidate's button. When a voter presses (for the second time) another candidate button then this is taken to represent their second preference, etc.... If they press a candidate's button that already has a preference associated with it then that press is ignored. When a voter is happy that they have recorded all their preferences, they press a *validate vote* button and the vote is sent to be stored. The *validate vote* button will only send the vote if at least one candidate button has been pressed."

We now model the design by a Java method (`pressCandButton`) which implements the behaviour of the system in response to the pressing of a candidate's button. We note that the choice of underlying data representation, with a vote as an array of candidate's preferences, maps closely the structure of the voting data as presented to the voter.

```
// Design1
// pressCandButton ****************************************
// Record next press as a preference, provided button is enabled
// Calls lastPref() to correctly assign the next preference value
// Calls buttonEnabled() to check if preference already allocated
public void pressCandButton(int button){
    if (!buttonEnabled(button-1)) return;
      else preferences[button-1] =lastPref()+1;
} // END Vote.pressCandButton
```

This `pressCandButton` method is dependent on two other methods: `buttonEnabled` for checking if the button being pressed is actually enabled (has not been pressed before), and `lastPref` for finding out the value of the last preference selected. The `buttonEnabled` behaviour is straightforward to implement, and left as an exercise for the reader. The `lastPref` method, as implemented below, examines all the preferences and deduces that the largest current preference value must have been the last preference made. We note, for future reference, that this is a correct (but inefficient) implementation of the design.

```
// lastPref ******************************************************
// Called by pressCandButton()
protected int lastPref(){
    int largest =0;
    for (int i = 0; i<numberofCandidates; i++)
      if (preferences[i]> largest) largest = preferences[i];
    return largest;
} //END Vote.lastPref
```

It is not easy to ensure the correctness of even this simple interface design without explicitly specifying and correctly implementing the `isValid()` method, as we have done. Without an explicit stament of what is valid, it is possible that the notion of validity could be interpreted ambiguously and thus the engineers could believe that they have built a correct interface when such an interface allows votes to be cast that the users do not consider to be valid.

4.3 Poor design may lead to security risks

A reasonable approach to rapidly prototyping this "simple" first design is to realise that a vote is valid as soon as one of the candidate buttons is pressed. This was hinted at in the initial statement of the design:

> "The *validate vote* button will only send the vote if at least one candidate button has been pressed."

It is a much quicker and simpler solution to hardcode this as an `enabled` boolean variable. In fact, this solution is correct but it is a poor design because it is not robust to changes in requirements. Consider a scenario where the interface requirements are extended to allow voters to reset their vote (in order, perhaps, to facilitate them in correcting an input error, or in changing their minds). The design using boolean `enabled` could result in the developers forgetting (during the coding of the new design feature) that they need to set this value to `false` when a vote is reset, and consequently lead to their designs allowing an *empty vote* — with no preferences recorded — to be sent to the store. Could this sort of thing happen in a real e-voting system? Judging by analysis regarding the quality of the code (and development methods) used in systems that have been examined in detail [11], one could not be sure that

professional developers would not make the same simple mistake.

It is difficult to judge the severity of such an interface design fault. Clearly, it has the potential for voters to have their voting intentions incorrectly recorded. This could lead to an election returning the wrong result. A second risk is that invalid empty votes in the store may break the tabulation process: if tabulation methods (functions) work on the assumption that all votes have at least one unique first preference (and may have been tested under that assumption) then it is possible that tabulation may not even terminate if this assumption is broken. A third risk is that some attacker has managed to introduce code into the system that can manipulate the tabulation process but will only be called when an invalid vote (or password-like sequence of invalid votes) is passed to the store. Of course, rigorous design procedures should find these types of design flaws without the need to resort to formal methods. However, in *critical system* design, "should" is not good enough.

4.4 Design2: a more sophisticated, incorrect, interface design

In the second design, we analyse how an extension to the requirements, in order to provide a more sophisticated interface, can pose specific problems. Imagine that we wish to provide a means of a voter changing their vote, without them having to reset all their preferences. In particular, we wish them to be able to cancel the last preference chosen (in the case that they accidentally pressed the wrong button). We propose the following design for verificiation:

"if the voter presses a candidate button again then that preference is erased".

This is a faulty design, but without the use of formal methods are we sure to identify the fault before more costly implementation takes place?

The `VoteExt1` class uses the inheritance mechanism in Java to add this extra interface feature to the already existing `Vote` class.

```
// Design2
public class VoteExt1 extends Vote {
VoteExt1 (int numCs){ super(numCs);}
// pressCandButton ********************************************
// The new feature requires over-riding of pressCandButton().
// Here is how it SHOULD NOT be done
public void pressCandButton(int button){
    if (!buttonEnabled(button-1)) preferences[button-1] = 0;
    else preferences[button-1] =lastPref()+1;
} // END VoteExt1.pressCandButton
} // END VoteExt1
```

In an ideal world, our development tool(s) should be able to automatically tell us that, either: the new functionality respects the safety property of the exisiting `Vote` class (that we have already formally verified) and provide the proof, or identify at least one scenario in which the new functionality breaks the safety property.

This code nearly works: it implements the requirements correctly provided one makes the assumption that a voter will only press a button a second time (to cancel a preference) *immediately* after pressing that button for a first time, with no other preferences being recorded in the mean time. We model this using B in Section 5; and demonstrate how the B tools automatically prove that the system is "broken" if this assumption about the voter's behaviour is invalid.

4.5 The risk of feature interactions in design

We note that making parallel changes to requirements and design models can lead to feature interactions similar to those well documented for telephone services [7]. Consider the interaction between two features that by themselves do not break the safety property of the system but when combined lead to invalid votes being recorded:

- The `lastPrefs` method was optimized by adding a counter value so that an iteration through the candidate list was not required each time a new preference was input.
- A `reset` button was added to allow all preferences to be deleted.

Individually, each of these refinements does not compromise the system by allowing the introduction of previously invalid votes. However, together they can result in an invalid vote with no first preference being recorded even though some preferences have been selected. (With candidates `A`, `B` and `C`, for example, the sequence of button presses `A-B-reset-C` leads, under the new combined feature functionality, to an invalid vote where only the 3rd preference for `C` is recorded.)

4.6 Design3: a more sophisticated, correct interface design

Consider the interface design whereby a voter can press a candidate button a second time and this will delete that candidate's preference value, as in Design2, above. However, it will also delete all the preference values lower than the preference just deleted.

Informally, one can verify its correctness with the following reasoning: "This will guarantee that if the 1st preference is deleted then all the preferences are deleted and that the empty vote is the only invalid vote that can be found in the interface." We see, in the next section, how we prove the correctness of the design in a more formal manner.

```
// Design3
public class VoteExt2 extends Vote {
VoteExt2 (int numCs){ super(numCs);}
// The extension over-rides the pressCandButton method.
// pressCandButton ********************************************
public void pressCandButton(int button){
if (!buttonEnabled(button-1)) {
    int deletefrom = preferences[button-1];
    for (int i = 0; i<numberofCandidates; i++){
      if (preferences[i] >= deletefrom) preferences[i] = 0;
      else preferences[button-1] =lastPref()+1;}
} // END VoteExt2.pressCandButton
} // END VoteExt2
```

Name	Syntax	Definition
Binary Relation	$s \leftrightarrow t$	$\mathcal{P}(s \times t)$
Domain	$\mathsf{dom}(r)$	$\{a \mid a \in s \wedge \exists b.(b \in t \wedge a \mapsto b \in r)\}$
Codomain	$\textsc{ran}\,(r)$	$\mathsf{dom}(r^{-1})$
Restriction	$s \lhd r$	$\mathbf{id}(s); r$
Co-restriction	$r \rhd t$	$r;\ \mathbf{id}(s)$
Anti-co-restriction	$r \rhd\!\!- t$	$r \rhd (ran(r) - t)$
Image	$r[w]$	$\textsc{ran}\,(w \lhd r)$
Partial Function	$s \nrightarrow t$	$\{r \mid r \in s \leftrightarrow t \ \wedge\ (r^{-1}; r) \subseteq \mathbf{id}(t)\}$
Total Function	$s \rightarrow t$	$\{f \mid f \in s \nrightarrow t \ \wedge\ \mathsf{dom}(f) = s\}$
Total injection	$s \rightarrowtail t$	$\{f \mid f \in s \rightarrow t \ \wedge\ f^{-1} \in t \nrightarrow s\}$

Fig. 1. B set notations

5 Formal software design for vote validity

5.1 Incremental development and refinement

The main idea in our refinement based-approach is to start with a very abstract model of the system under development. We then gradually add details to this first model by building a sequence of more concrete ones. The relationship between two successive models in this sequence is that of *refinement* [3,1]. This is controlled by means of a number of *proof obligations*, which guarantee the correctness of the development. Such proof obligations are proved by automatic (and interactive) proof procedures supported by a proof engine [5]. The essence of the refinement relationship is that it preserves *system properties*.

Figure 1 gives set-theoretical notations of the B data modelling language. A complete introduction to B can be found in [1].

The refinement of an event B model [**?**] allows one to enrich the model in a *step-by-step* manner. Refinement provides a way to construct stronger invari-

ants and also to add details to a model. It is also used to transform an abstract model into a more concrete version by modifying the state description. This is essentially done by extending the list of state variables (possibly suppressing some of them), by refining each abstract event into a corresponding concrete version, and by adding new events. The abstract state variables, x, and the concrete ones, y, are linked together by means of a *gluing invariant* $J(x, y)$. A number of proof obligations ensure that: (1) each abstract event is correctly refined by its corresponding concrete version, (2) each new event refines *skip*, (3) no new event takes control for ever, and (4) relative deadlock-freeness is preserved.

Following the traditional refinement process, the first model (`AllVotesAtOnce`) we propose is very high level: it abstracts away from individual votes and button presses. There is only one event — Voting — which models the votes of all electors who came to vote in *one shot*.

For readers unfamiliar with the B modelling language, we note that:

- $ELECTOR$ is the set of all electors and $CAND$ is the set of all candidates,
- *vote* is the variable which contains votes of all the electors, and
- *nbv* is the cardinal of the domain of *vote* representing the total number of votes.

Instead of modelling all the votes in a single *one shot*, as above, in our next step we choose to refine the most abstract specification so that each individual vote is modelled, using the event One_Vote in the model `EachVoteAtOnce`. Without such a refinement a realisitic implementation cannot be constructed.

In this more concrete `EachVoteAtOnce` model, $STATE$ is an enumeration set which contains two values: *voting* to represent a voting system that is *open*, and $finish$ to represent when the voting is *closed*. The variable *st* contains one of these values, *elector* is the subset of $ELECTOR$ recording the electors who have already voted, *vt* is the variable which contains these votes, and *cvt* is its cardinal. All votes in *vt* are valid. The event One_Vote models the vote of a single elector in *one shot* (as a single event).

The B models that follow correspond to the three Java designs that we saw in Section 4. We show how the designs can be formally verified when modelled in B.

5.2 Design1: the simplest interface

In this `Vote` model — which corresponds to the Java `Vote` class — an elector e (who has not already voted) votes for candidates by pressing on the corresponding buttons.

One_voting is an enumeration set which contains three values: *no_elec* when no electors are voting, *start* when the a new elector e starts to vote, and *valid* when the elector e pushes the button to validate their vote. The variable *sto* contains one of these values. Variable e contains the current elector, *vt* is

MODEL

$AllVotesAtOnce$

SETS

$ELECTOR; CAND$

CONSTANTS

nbc

PROPERTIES

$nbc = card(CAND)$

DEFINITIONS

$valid(v) \;\widehat{=}\; \exists n \cdot (n \in 1..nbc \;\wedge\; v^{-1} \in 1..n \rightarrowtail CAND))$

VARIABLES

$vote, nbv$

INVARIANT

$vote \in ELECTOR \leftrightarrow (CAND \times (1..nbc)) \;\wedge$

$\forall e \cdot (e \in \mathsf{dom}(vote) \;\Rightarrow\; valid(vote[\{e\}]) \;\wedge$

$nbv \in \mathbb{N} \;\wedge$

$nbv = card(\mathsf{dom}(vote))$

INITIALISATION

$vote := \emptyset || nbv := 0$

EVENTS

Voting $\widehat{=}$

begin

$$vote, nbv :| \left(\begin{array}{l} vote \in ELECTOR \leftrightarrow (CAND \times (1..nbc)) \;\wedge \\ \forall e \cdot (e \in \mathsf{dom}(vote) \;\Rightarrow\; valid(vote[\{e\}])) \;\wedge \\ nbv \in \mathbb{N} \;\wedge \\ nbv = card(\mathsf{dom}(vote)) \end{array} \right)$$

end

their current vote which is modified when a candidate button is pushed, and n is the preference number of the chosen candidate.

We remark that the guard of the event Button_valid requires that $n \neq 0$ and so we are sure that when an elector pushes this button then the partial vote v is not empty and so v is a valid vote. Remark also that we have no condition on n in the guard of the event Button_cand. When a candidate is not in the codomain of v we are sure[10] that $n < nbc$.

[10] We have proven it thanks to the invariant property and the refinement construct. This

MODEL

$EachVoteAtOnce$

REFINES

$AllVotesAtOnce$

SET

$STATE = \{voting, finish\}$

VARIABLES

$vote, nbv,$
$st, elector, vt, cvt$

INVARIANT

$elector \subseteq ELECTOR \ \wedge$
$vt \in elector \leftrightarrow (CAND \times (1..nbc)) \ \wedge$
$\mathsf{dom}(vt) = elector \ \wedge$
$\forall e \cdot \ (e \in \mathsf{dom}(vt) \ \Rightarrow \ valid(vt[\{e\}]) \ \wedge$
$cvt \in \mathbb{N} \ \wedge$
$cvt = card(elector) \ \wedge$
$st \in STATE \ \wedge$
$(st = finish \ \Rightarrow \ \mathsf{dom}(vote) = elector)$

INITIALISATION

$vote := \emptyset \ || \ nbv := 0 \ ||$
$st := voting \ || \ elector := \emptyset \ ||$
$cvt := 0 \ || \ vt := \emptyset$

EVENTS

One_Vote $\widehat{=}$
any e, v, n **where**
$st = voting$
$e \in ELECTOR - elector$
$n \in 1..nbc$
$v \in 1..n \rightarrowtail CAND$
then
$vt := vt \ \cup \ (\{e\} \times v^{-1}) \ ||$
$elector := elector \ \cup \ \{e\} \ ||$
$cvt := cvt + 1$
end;
Voting $\widehat{=}$
when
$st = voting$
then
$vote, nbv := vt, cvt \ ||$
$st := finish$
end

END

5.3 Enriching the interface: Design2 and Design3

In the previous model an elector cannot correct input mistakes: now we specify three: (1) delete that candidate's preference (corresponding to `VoteExt1` in the Java), or (2) delete that candidate's preference only if that was the last preference made (corresponding to the suggested "fix" that we wished to formally verify), or (3) delete that candidate's preference and all lower preferences (corresponding to `VoteExt2`).

In the first design, if we remove the corresponding vote and decrement our

proof, as all others referred to in the paper, is available from the authors on request.

MODEL

$Vote$

REFINES

$EachVoteAtOnce$

SET

$One\_Voting = \{start, valid, no\_elec\}$

VARIABLES

$vote, nbv,$

$st, elector, vt, cvt$

e, n, v, sto

INVARIANT

$e \in ELECTOR \ \wedge$

$v \in \mathbb{N} \leftrightarrow CAND \ \wedge$

$n \in 0..nbc \ \wedge$

$n = card(\mathsf{ran}(v)) \ \wedge$

$sto \in One\_Voting \ \wedge$

$(sto \neq no\_elec \ \Rightarrow$

$v \in 1..n \rightarrowtail CAND) \ \wedge$

$(sto \neq no\_elec \ \Rightarrow$

$e \in ELECTOR - elector) \ \wedge$

$(st = voting \ \wedge \ sto = valid \ \Rightarrow \ n \neq 0)$

INITIALISATION

$vote := \emptyset \ || \ nbv := 0 \ ||$

$st := voting \ || \ elector := \emptyset \ ||$

$cvt := 0 \ || \ vt := \emptyset \ ||$

$sto := no\_elec \ || \ e :\in ELECTOR \ ||$

$v := \emptyset \ || \ n := 0$

EVENTS

Start_vote $\widehat{=}$

any E **where**

$sto = no\_elec \ \wedge$

$E \in ELECTOR - elector$

then

$sto := start \ ||$

$e := E \ ||$

$n := 0 \ ||$

$v := \emptyset$

end;

Button_cand $\widehat{=}$

any c **where**

$sto = start \ \wedge$

$c \in CAND - \mathsf{ran}(v)$

then

$n := n + 1 \ ||$

$v := v \ \cup \{n + 1 \mapsto c\}$

end;

Button_valid $\widehat{=}$

when

$sto = start \ \wedge$

$n \neq 0$

then

$sto := valid$

end;

One_Vote $\widehat{=}$

when

$st = voting \ \wedge$

$sto = valid$

then

$vt := vt \ \cup \ (\{e\} \times v^{-1}) \ ||$

$elector := elector \ \cup \ \{e\} \ ||$

$cvt := cvt + 1 \ ||$

$sto := no\_elec$

end;

END

Button_cancel_incorrect_cand $\widehat{=}$
any c **where**
$sto = start \ \wedge$
$c \in \mathsf{ran}(v)$
then
$n := n - 1 \ ||$
$v := v \ \rhd \ \{c\}$
end;

Button_cancel_last_cand $\widehat{=}$
any c **where**
$sto = start \ \wedge$
$n \mapsto c \in v$
then
$n := n - 1 \ ||$
$v := v \ \rhd \ \{c\}$
end;

Button_cancel_cand $\widehat{=}$
any c **where**
$sto = start \ \wedge$
$c \in \mathsf{ran}(v)$
then
$n := v^{-1}(c) - 1 \ ||$
$v := \{n | n \in \mathbb{N} \wedge n < v^{-1}(c)\} \ \lhd \ v$
end;

$$(sto \neq no\_elec \ \Rightarrow \ v^{-1}(c) - 1 = card(\mathsf{ran}(\{n | n \in \mathbb{N} \wedge n < v^{-1}(c)\} \ \lhd \ v))$$

counter an unproved proof obligation is generated. This is formally treated by the *Button_cancel_incorrect_cand* event. In particular, the new event doesn't preserve the invariant which says that the partial vote v is an injection.

In Section 4, we suggested a "fix" to the previous design. Using B, this "fix" can be formally modelled (in event *Button_cancel_last_cand*) and proven correct. In fact, there are no difficulties to prove the invariant preservation. All new proof obligation are discharged automatically by the prover.

For this event there was only one difficulty in the proof process: we need to prove that $v^{-1}(c) - 1 = card(\mathsf{ran}(\{n | n \in \mathbb{N} \wedge n < v^{-1}(c)\} \ \lhd \ v))$. We have proved this assertion — for all c in the co-domain of v — by simple induction on the assertions clauses.

6 The semi-automated proof process

The complexity of the development is evaluated through the number of proof obligations generated for the validation of each model or refinement; among

generated proof obligations, a large number are automatically discharged by the tool [5]. In our simple case study, 44 proof obligations are automatically discharged, and 10 are interactively proved using the tool but with human help.

7 Conclusions and Future Work

We have demonstrated the use of the formal method B in guaranteeing a simple safety property of an interface to an e-voting machine. We demonstrated that guaranteeing *validity* of votes recorded not only helps in the formal verification of the voting process, but also has an important role to play in making the machine more secure. This is the first step in developing a generic framework for the design of secure interfaces which could be proved to satsify various safety-related properties. It is our goal to try and formulate such a framework as a set of formal design patterns much like those proposed by Abrial when using B to verify properties of control systems [2]

There are interesting alternative techniques for e-voting, like pollsterless systems [16] which could benefit from further formal verification using our approach. This is planned for future work. Furthermore, we are currently using B to prove the correctness of an actual storage mechanism which claims to offer tamper-evident, history-independent and subliminal free data structures [14]. After this we aim to use B to prove safety properties concerned with the tabulation of votes, and so verify all layers in a typical voting architecture.

References

[1] Abrial, J., "The B Book - Assigning Programs to Meanings," Cambridge University Press, 1996, iSBN 0-521-49619-5.

[2] Abrial, J.-R., *Formal methods in industry: achievements, problems, future*, in: *ICSE '06: Proceeding of the 2&8th international conference on Software engineering* (2006), pp. 761–768.

[3] Back, R. J. R., *On correct refinement of programs*, Journal of Computer and System Sciences **23** (1979), pp. 49–68.

[4] Bate, I., R. Hawkins and J. McDermid, *A contract-based approach to designing safe systems*, in: *CRPIT '33: Proceedings of the 8th Australian workshop on Safety critical systems and software* (2003), pp. 25–36.

[5] ClearSy, "Web site B4free set of tools for development of B models," (2004). URL http://www.b4free.com/index.php

[6] Farrell, D. and I. McAllister, *"The 1983 change in surplus vote transfer procedures for the australian senate and its consequences for the single transferable vote"*, Australian Journal of Political Science **38** (2003), pp. 479–491.

[7] Gibson, J. P., *Feature requirements models: Understanding interactions.*, in: P. Dini, R. Boutaba and L. Logrippo, editors, *FIW* (1997), pp. 46–60.

[8] Gritzalis, D., editor, "Secure Electronic Voting," Advances in Information Security **7**, Springer, 2003.

[9] Herrnson, P. S., B. B. Bederson, B. Lee, P. L. Francia, R. M. Sherman, F. G. Conrad, M. Traugott and R. G. Niemi, *Early appraisals of electronic voting*, Soc. Sci. Comput. Rev. **23** (2005), pp. 274–292.

[10] Kocher, P. and B. Schneier, *Insider risks in elections*, Commun. ACM **47** (2004), p. 104.

[11] Kohno, T., A. Stubblefield, A. D. Rubin and D. S. Wallach, *Analysis of an electronic voting system*, in: *IEEE Symposium on Security and Privacy (S&P 2004)* (2004), pp. 27–40.

[12] McGaley, M. and J. P. Gibson, *E-Voting: A Safety Critical System*, Technical Report NUIM-CS-TR-2003-02, NUI Maynooth, Comp. Sci. Dept. (2003).

[13] Mercuri, R. T., *Computer security: quality rather than quantity*, Commun. ACM **45** (2002), pp. 11–14.

[14] Molnar, D., T. Kohno, N. Sastry and D. Wagner, *Tamper-evident, history-independent, subliminal-free data structures on prom storage-or-how to store ballots on a voting machine (extended abstract)*, in: *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P'06)* (2006), pp. 365–370.

[15] Neumann, P. G., *Inside risks: risks in computerized elections*, Commun. ACM **33** (1990), p. 170.

[16] Storer, T. and I. Duncan, *Polsterless remote electronic voting*, Journal of E-Government **1** (2004).
URL http://www.dcs.st-and.ac.uk/research/publications/SD04a.php

Guaranteeing Consistency in Text-Based Human-Computer-Interaction

Bernhard Beckert [1]

Department of Computer Science
University Koblenz-Landau
Koblenz, Germany

Gerd Beuster [2,3]

Department of Computer Science
University Koblenz-Landau
Koblenz, Germany

Abstract

Wrong assumptions about the state of the computer system are a main source of error in human-computer interaction. We show how consistency requirements between the state of a computer system and the user's assumptions about the state can be formally defined. The definition of HCI consistency is used to show correctness of a methodology to ensure consistency for TTY-based applications.

Key words: Formal Methods, Security, HCI

1 Introduction

Security of interactive systems critically depends on correct display of the system's state. Only if the user's assumptions about the state of the system are correct, can he make informed decisions about the further course of action. Informally, a system is consistent if the user's assumptions about the system correspond to the actual system state whenever he interacts with the system. There are two main sources for wrong assumptions about system state leading to inconsistent systems:

[1] Email: beckert@uni-koblenz.de
[2] Email: gb@uni-koblenz.de
[3] This work was partially funded by the German Federal Ministry of Education, Science, Research and Technology (BMBF) in the framework of the Verisoft project under grant 01 IS C38. The responsibility for this article lies with the authors. See http://www.verisoft.de for more information about Verisoft.

This paper has been presented at the
1st International Workshop on Formal Methods for Interactive Systems
URL: fmis.iist.unu.edu

- Inconsistency during updates

  Human-Computer Interaction is inherently asynchronous. Execution of user commands and updates of the data displayed by the output device take time. Due to the inherently asynchronous character of Human-Computer Interaction, the user may err about the system state; either because commands have not been executed yet, or because the screen has not been updated.

- Insufficient information or wrong interpretation of data

  The system may not provide enough information to determine the system sate, or the user may interpret application output wrongly. A large part of the specification of interactive applications is concerned with the relation between user input and the information shown to the user. For example, when editing a text, the current (internal) state of the text should be shown to the user, and user input should cause changes to the text. Usually, the specification of user input and system output is rather informal. Specifications declare that something "is shown on the screen" and the user "enters a text." In most cases, this informal description is sufficient. However, in security-critical applications, a precise and formal definition is desirable.

The second point, wrong interpretation of data, is addressed in a number of works. Reeder and Maxion [21] analyzed the problem of representing NTFS file permissions on Windows XP systems and developed the design principle of "anchor-based subgoaling" in order to mitigate the problem.

In this work, we address the first source of errors, inconsistencies during updates. Most user interface security requirements are highly application specific. However, there are also some generic requirements. We show that for a large class of applications, it is possible to define generic requirement in a formal way. In this paper, we focus on one of these generic requirements: The user should always be aware of the system state when he issues a command. We show how consistency during updates can be guaranteed for text-based applications.

In Section 4 we develop a formal definition of consistency based on the generic computer security requirement of application Integrity. In Section 5, we show that the common approach to model interactive applications does not guarantee consistency. We provide an alternative model for which consistency can be guaranteed. In Sections 6 and 7, a generic specification template for interactive applications that guarantee consistency in HCI is presented. In this paper, we consider text-based user interfaces only, but our methods can be extended for handling graphical user interfaces. In the Verisoft project (http://www.verisoft.de), our methods are applied to specify and verify an email system.

2 Related Work

We build upon work on formal methods for developing computing systems, human-computer interaction (HCI) research, and secure system design.

Formal methods, human computer interaction, and security are established fields of computer science research. There is also work combining each two of these fields. Formal methods have been used to specify human-computer interaction. User interfaces have been designed and evaluated under security aspects. System security has been treated with formal methods. In this work, we combine all *three* fields.

Abowd et al. [1] and Jain [15] give a survey of formal languages for the description of user interfaces. More overviews are given in two (different) books called *Formal Methods in Human-Computer Interaction* [13,19]. An early contribution to formal methods for the description of user interaction is the PIE model, developed by Dix and Runciman [9]. In this model, system behavior is defined as a function from commands issued by the user to effects produced by the system. In case of a text-based user interface, the input is a sequence of keystrokes and the output are characters displayed on the screen [8].

Carr's Interaction Object Graphs (IOG) are an extension of statecharts for modeling elements of graphical user interfaces and their interactions [4]. It allows a description both on the pixel-level and on an aggregated level. IOGs focus on graphical user interfaces, and the language used to describe them is directly executable. The formalism of IOG allows basic reasoning tasks like testing for reachability of all states. Statecharts are also used by Degani et al. for specification of the interaction interfaces between human users and machines, addressing the question what information about a machine's state is required in order to operate it safely [6].

Sucrow [23] uses graph grammars to describe graphical user interface elements. Changes in the GUI are modeled by re-write rules. Palanque et al. [18] use hierarchical Petri nets to combine user models and a system models of interactive systems. Berstel et al. developed "Visual Event Grammars" (VEG), a formal method for the specification and validation of graphical user interfaces [2]. They describe complex graphical user interface as communicating automata.

PIE and similar formalisms put an emphasis on describing the I/O behavior of a computer system and are suitable for automated reasoning, e.g. with model checkers. Rushby uses model checking in order to detect potential discrepancies between system behavior and the mental models of system users [22].

Other approaches like Task Knowledge Structures (TKS) [12], (Extended) Task Action Grammar ((E)TAG) [5], and Goals Operators Methods Selection-rules (GOMS) [16] focus on providing cognitive models of the user. TKS provides an explicit representation of the cognitive model of the user. TAG

allows a precise formal description of the user actions, the user's knowledge and the user's internal representation of the system (what the user thinks about the system). GOMS is more oriented towards psychological analysis of user behavior and timed measurement of user activity. A major weakness of GOMS is that it is limited to sequential user plans, and that it does not provide means to generate application specifications from user models. ConcurTaskTrees [20], developed by Paterno et al., provides a richer formalism for the description of user behavior and generation of application specifications. Harrison et al. [27] use formal methods to derive requirements for human-error tolerance from task descriptions.

A general weakness of these formal HCI models is that they require detailed models of the user behavior in order to model the interaction between a computer system and a user. While computer systems can (and should) be formally specified, a formal user model is always based on assumptions about the user which may or may not be true. The approach presented in this we paper make no unnecessary assumptions about the user.

In [13], Dix and Harrison develop the concept of "State Display Conformance" which is closely related to the consistency requirements developed in this paper. It should be noted that Grudin's argument *against* user interface consistency requirements [11] does not apply to the work presented in this paper. He argues that consistency defined as having similar user interface elements for similar functionality can not be generalized, because similarity depends on context. Our work however does not address consistency within a user interface, but consistency between a user's mental representation of a system state and the actual system state.

In a number of works, formal specification methods like Z have been applied to user interface design. One of the first formal specifications of interactive components was the specification of a text editor in Z in Sufrin's paper *Formal specification of a display editor* [24]. Based on Sufrin's specification, Booth and Jones implemented an editor in the Miranda functional programming language [3]. Goldson [10] and Hussey/Carrington [14] provide more case studies in using Z for user interface specification.

3 Notation

We specify the abstract behavior of system components by Input Output Labeled Transition Systems (IOLTS) and Linear Temporal Logic (LTL). Below, we define these concepts and some related notions used throughout this paper.

Definition 3.1 A *Labeled Transition System* (LTS) is a tuple $L = (S, \Sigma, s_0, \rightarrow)$ where S is a set of *states*, $s_0 \in S$ is an *initial state*, Σ is a set of *labels*, and $\rightarrow \subseteq S \times \Sigma \times S$ is a *transition relation*. We use the notation $p \xrightarrow{\sigma} q$ for $(p, \sigma, q) \in \rightarrow$.

Definition 3.2 An *Input Output Labeled Transition System* (IOLTS) is an

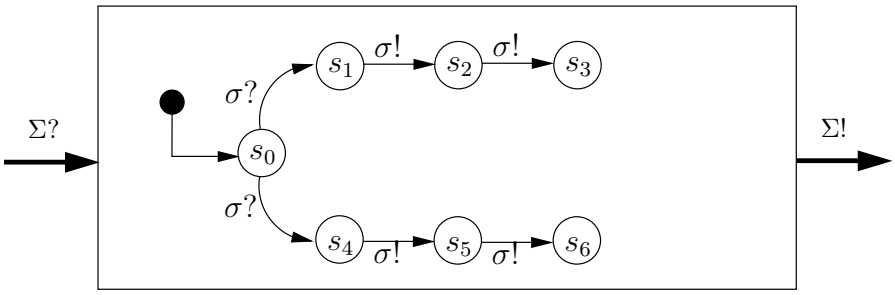

Fig. 1. State Transition Diagram representation of an IOLTS.

LTS $L = (S, \Sigma, s_0, \rightarrow)$ with $\Sigma = \Sigma? \cup \Sigma! \cup \Sigma I$. We call $\Sigma?$ the *input alphabet*, $\Sigma!$ the *output alphabet*, and ΣI the *internal alphabet*.

We use state transition diagrams to visualize IOLTS. An example is shown in Figure 1.

The combination of two IOLTSs L_a and L_b where the output alphabet of L_a is the input alphabet of L_b is called a *composition*:

Definition 3.3 Let $L_a = (S_a, \Sigma_a, s_{0a}, \rightarrow_a)$, $L_b = (S_b, \Sigma_b, s_{0b}, \rightarrow_b)$ be two IOLTS with $\Sigma!_a = \Sigma?_b$. The *composition* $(L_a||L_b) = (S, \Sigma, s_0, \rightarrow)$ of L_a and L_b is defined by:

$$
\begin{aligned}
S &= S_0 \times S_1 \\
\Sigma? &= \Sigma?_a \\
\Sigma! &= \Sigma!_b \\
\Sigma I &= \Sigma I_a \cup \Sigma I_b \cup \Sigma!_a \\
s_0 &= (s_{0a}, s_{0b}) \\
\rightarrow &= \{((s_a, s_b), \sigma, (s'_a, s_b)) \mid s_a \xrightarrow{\sigma}_a s'_a \text{ with } \sigma \in \Sigma?_a \cup \Sigma I_a\} \cup \\
&\quad \{((s_a, s_b), \sigma, (s_a, s'_b)) \mid s_b \xrightarrow{\sigma}_b s'_b \text{ with } \sigma \in \Sigma!_b \cup \Sigma I_b\} \cup \\
&\quad \{((s_a, s_b), \sigma, (s'_a, s'_b)) \mid s_a \xrightarrow{\sigma}_a s'_a \text{ and } s_b \xrightarrow{\sigma}_b s'_b \text{ with } \sigma \in \Sigma!_a = \Sigma?_b\}
\end{aligned}
$$

Often, components are combined by *mutual* composition. In mutual composition, the output of L_a serves as input for L_b, and the output of L_b serves as input of L_a (this is illustrated in Figure 2).

Definition 3.4 Let $L_a = (S_a, \Sigma_a, s_{0a}, \rightarrow_a)$ and $L_b = (S_b, \Sigma_b, s_{0b}, \rightarrow_b)$ be IOLTS.

We assume the input and output alphabets of L_a and L_b to consist of internal and external subsets, where the internal input is denoted with $\Sigma?I$, the external input with $\Sigma?I$, the internal output with $\Sigma!I$, and the external output with $\Sigma!E$. And we demand that these subsets are chosen such that $\Sigma!I_a = \Sigma?I_b$ and $\Sigma!I_b = \Sigma?I_a$.

Then, the *mutual composition* $(L_a||_m L_b) = (S, \Sigma, s_0, \rightarrow)$ of L_a and L_b is defined by:

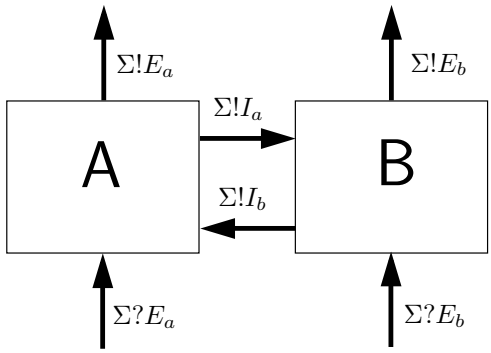

Fig. 2. Mutual composition of IOLTSs.

$$
\begin{aligned}
S &= S_0 \times S_1 \\
\Sigma? &= \Sigma?E_a \cup \Sigma?E_b \\
\Sigma! &= \Sigma!E_a \cup \Sigma!E_b \\
\Sigma I &= \Sigma I_a \cup \Sigma I_b \cup \Sigma!I_a \cup \Sigma!I_b \\
s_0 &= (s_{0a}, s_{0b}) \\
\rightarrow &= \{(s_a, s_b), \sigma, (s'_a, s_b)) \mid s_a \xrightarrow{\sigma}_a s'_a \text{ with } \sigma \in \Sigma?E_a \cup \Sigma!E_a \cup \Sigma I_a\} \cup \\
&\quad \{(s_a, s_b), \sigma, (s_a, s'_b)) \mid s_b \xrightarrow{\sigma}_b s'_b \text{ with } \sigma \in \Sigma?E_b \cup \Sigma!E_b \cup \Sigma I_b\} \cup \\
&\quad \{(s_a, s_b), \sigma, (s'_a, s'_b)) \mid s_a \xrightarrow{\sigma}_a s'_a \text{ and } s_b \xrightarrow{\sigma}_b s'_b \text{ with} \\
&\qquad\qquad \sigma \in \Sigma!I_a \cup \Sigma!I_b = \Sigma?I_b \cup \Sigma?I_a\}
\end{aligned}
$$

The input/output behavior of a component is described by *traces*, which are (possibly infinite) sequences of elements from the alphabet Σ, and *paths*, which are corresponding sequences of states.

Definition 3.5 Let $L = (S, \Sigma, s_0, \rightarrow)$ be an IOLTS. Then, a *path* is a sequence $\langle s_0, s_1, \ldots \rangle$ of states from S with $s_i \rightarrow s_{i+1}$ for all $i \geq 0$. A *trace* (of L) is a sequence $\langle \sigma_0, \sigma_1, \ldots \rangle$ of elements of Σ such that there is a path $\langle s_0, s_1, \ldots \rangle$ with $s_i \xrightarrow{\sigma_i} s_{i+1}$ $(i \geq 0)$.

We use Linear Temporal Logic (LTL) to describe properties of components. The syntax of LTL is defined as usual, i.e., given a set P of atomic propositions, LTL formulae ϕ are constructed inductively by:

$$\phi ::= p | \phi \vee \phi | \phi \wedge \phi | \neg\phi | X\phi | \phi U \phi | G\phi | F\phi \qquad (p \in P)$$

Now, we can use IOLTSs to interpret LTL formulas—in combination with valuations λ that map atomic propositions to the states in which they are true. The satisfaction relation is extended to more complex formulae as usual.

Definition 3.6 Given an IOLTS $L = (S, \Sigma, s_0, \rightarrow)$ and a set P of atomic propositions, a *valuation* λ is a mapping from P to S. An atom p is said to be true in $s \in S$ iff $s \in \lambda(P)$.

Given a path $c = \langle s_0, s_1, \ldots \rangle$, by c^i we denote the sub-path of c starting at s_i.

Whether an LTL formula ϕ is satisfied by a path c and a valuation λ, denoted by $L, \lambda, c \models \phi$, is inductively defined as follows:

- $L, \lambda, c \models \top$
- $L, \lambda, c \models \phi$ if $\phi \in P$ and $s_0 \in \lambda(\phi)$
- $L, \lambda, c \models \neg\phi$ if not $L, \lambda, c \models \phi$
- $L, \lambda, c \models \phi \wedge \psi$ if $L, \lambda, c \models \phi$ and $L, \lambda, c \models \psi$
- $L, \lambda, c \models \phi \vee \psi$ if $L, \lambda, c \models \phi$ or $L, \lambda, c \models \psi$
- $L, \lambda, c \models \mathbf{X}\phi$ if $L, \lambda, c^1 \models \phi$
- $L, \lambda, c \models \phi \mathbf{U} \psi$ if (a) $L, \lambda, c \models \psi$ or (b) there is some $i \geq 1$ s.t. $L, \lambda, c^i \models \psi$ and $L, \lambda, c^k \models \phi$ for all $0 \leq k < i$
- $L, \lambda, c \models \mathbf{G}\phi$ if $L, c^i \models \phi$ for all $i \geq 0$
- $L, \lambda, c \models \mathbf{F}\phi$ if $L, c^i \models \phi$ for some $i \geq 0$

An LTL formula ϕ is said to be satisfied by a valuation λ, denoted by $L, \lambda \models \phi$, iff $L, \lambda, c \models \phi$ for all paths c of L. And ϕ is said to be satisfied by L, denoted by $L \models \phi$ iff $L, \lambda \models \phi$ for all valuations λ.

4 Formal Definition of User Interface Integrity

The aim of computer security is to guarantee access to services and resources to authorized persons, while preventing access and manipulation by unauthorized parties. The basic security threats are *Data Leaking*, *Data Manipulation*, and *Program Manipulation* [7]. These are countered by the core security requirements, usually abbreviated as *CIA*:

Confidentiality: Information is available to authorized parties only.

Integrity: Both the assumptions of the user about the application, and the assumptions of the application about the user are correct.

Availability: Accessibility of services and data is guaranteed.

Adapting these concepts to user interface security is straightforward:

HCI Confidentiality: No secret information is leaked via the user interface.

HCI Integrity: There is a correspondence between the configuration of the application (defined by its internal state and data), and the user's assumption about the data and the state.

HCI Availability: The user interface must guarantee reachability of desirable states, and it must prevent user interactions that lead to transitions into undesirable states.

In the following, we concentrate formalizing the integrity requirement. Informally, we define HCI Integrity as follows:

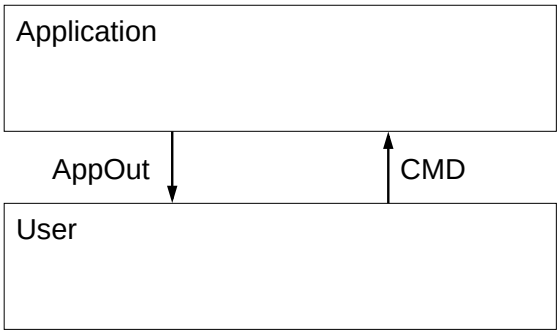

Fig. 3. Basic System (User + Application) Model

Definition 4.1 HCI Integrity: Whenever the user makes a critical decision, all critical properties are the same in the application and the user's assumption about the application.

In order to maintain a most general view on all possible applications and user models, we do not define *what* constitutes critical properties and their correct interpretation by the user. We only assume that there *are* critical properties, and that the user may or may not have correct assumptions about them. We assume that in all critical states atomic proposition *critical* holds, and that there are atomic propositions $a_0, \ldots, a_n$ representing critical properties of the application, and $u_0, \ldots, u_n$ representing the user's assumptions about these properties. With these definitions, HCI Integrity is defined by the LTL formula

$$\mathbf{G}(\mathit{critical} \rightarrow ((a_0 \leftrightarrow u_0) \wedge (a_1 \leftrightarrow u_1) \wedge \cdots \wedge (a_n \leftrightarrow u_n))) \quad (1)$$

The definition of $a_0, \ldots, a_n$ and $u_0, \ldots, u_n$ depends on the actual application and user models. For given user and application models, automated reasoning techniques (e.g., model checking) can be used to check if the HCI Integrity formula holds.

5 Guaranteeing Integrity

In the last Section, we developed a formal definition of integrity. In order to apply the definition, suitable user and application models and definitions of a valuation function λ must be provided. In this Section, we use the methodology to deduce required properties of a generic class of text-based user interface. Based on this, a specification of a main execution loop for this class of applications is developed in Section 6.

In a generic model of a TTY application, one user interacts with one application. A keyboard is used as the input device and a TTY screen as the output device. This model is depicted in Figure 3. Two types of messages are used to exchange information between the user and the application: *AppOut* is the data type for information shown on the screen. *CMD* is the data type for input given by the user. This model can be further structured without losing

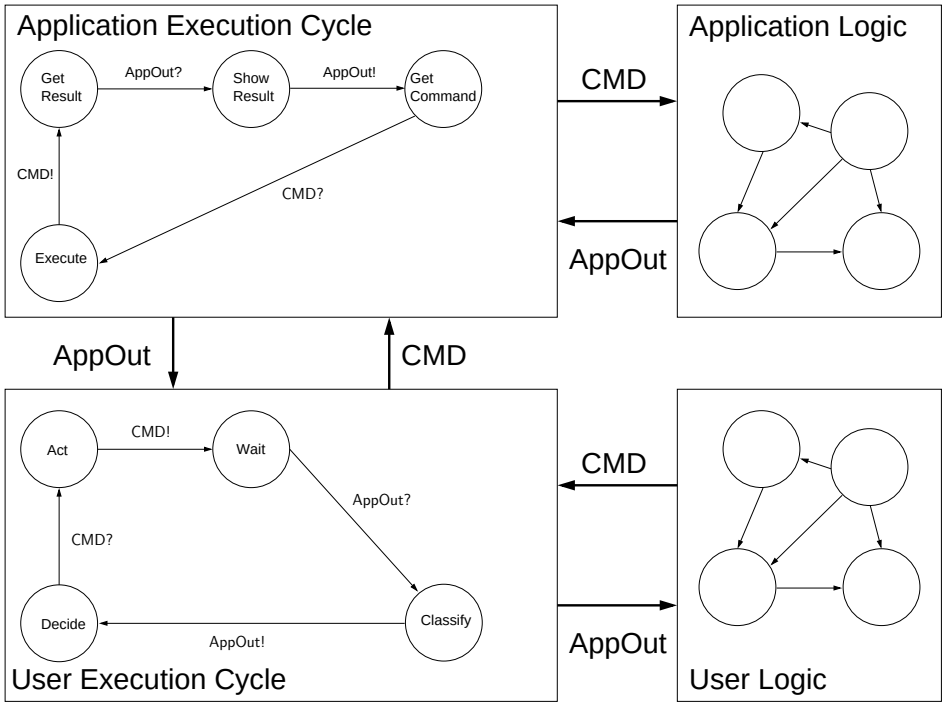

Fig. 4. Basic generic model of user and application.

generality. All well-designed applications (and all reasonable models of user behavior) split up the components into a generic execution loop, governing the general behavior of the application (or the user), and an application (task) specific component. The separation of a generic execution loop and a task specific component serves two purposes: It follows established system design practice and therefore allows realistic modeling of applications. Secondly, the separation in a generic and an application specific component allows to deduce properties that hold for all applications following this design, independent of the concrete application's task.

A basic model following this approach is shown in Figure 4. In this model, *AppOut* and *CMD* are variables representing all possible command input and application output. Question marks after variable names indicate reading of an input value, and exclamation marks indicate writing of an output value. Thus, in one cycle of application execution the following steps are taken: The application waits for the user to enter a command (GetCommand $\xrightarrow{\text{CMD?}}$ Execute). The command is processed by application logic (Execute $\xrightarrow{\text{CMD!}}$ GetResult, GetResult $\xrightarrow{\text{AppOut?}}$ ShowResult), and the result of the computation is forwarded to the output device (ShowResult $\xrightarrow{\text{AppOut!}}$ GetCommand). In the same way, the user reads application output, evaluates which command should be issued next, and enters the command into the input device.

This basic model already allows to deduce interesting properties in respect to Integrity constraints. A reasonable assumption is that all decisions made

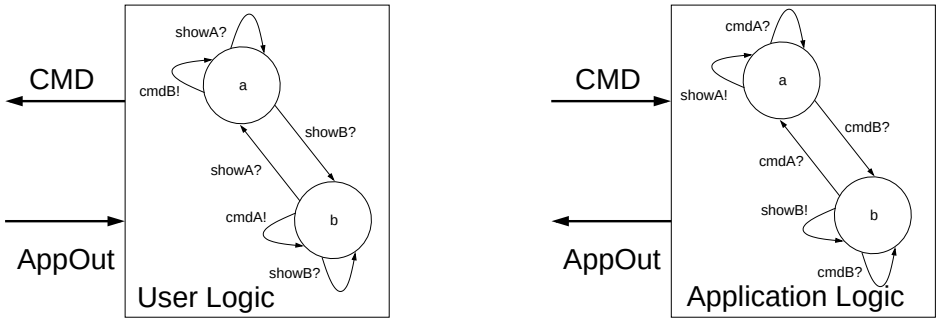

Fig. 5. Simple Logic Modules.

AppExec	**UserExec**	**AppLogic**	**UserLogic**	**AppOut**	**CMD**
ShowResult	Wait	a	b	-	-
GetCommand	Wait	a	b	showA	-
GetCommand	Classify	a	b	showA	-
GetCommand	Decide	a	a	showA	-
GetCommand	Act	a	a	showA	-
GetCommand	Wait	a	a	showA	cmdB
Execute	Wait	b	a	showA	-
Execute	Classify	b	a	showA	-
Execute	Decide	b	a	showA	-

Fig. 6. Refutation of naïve model (excerpt)

by the user are critical::

$$\lambda(\mathit{critical}) = \{\text{UserExeCyc.Decide}\}$$

Critical properties of an application, and user assumptions about critical properties, always depend on the user and application model. We provide most simple definitions of these components in order to show a general weakness of the naïve application and user execution cycle model. In this most simple component definition, there are only two commands, two application outputs, and two states in both the user and application logic model. These logic models are shown in Figure 5. As the security relevant property, we define the question whether the application is in state "a":

$\lambda(a_0) = \{\text{AppLogic.a}\}$
$\lambda(u_0) = \{\text{UserLogic.a}\}$

Integrity is not guaranteed for this model. The problem lies in the lack of consistency, as the trace given in Figure 6 shows: When the user decides about

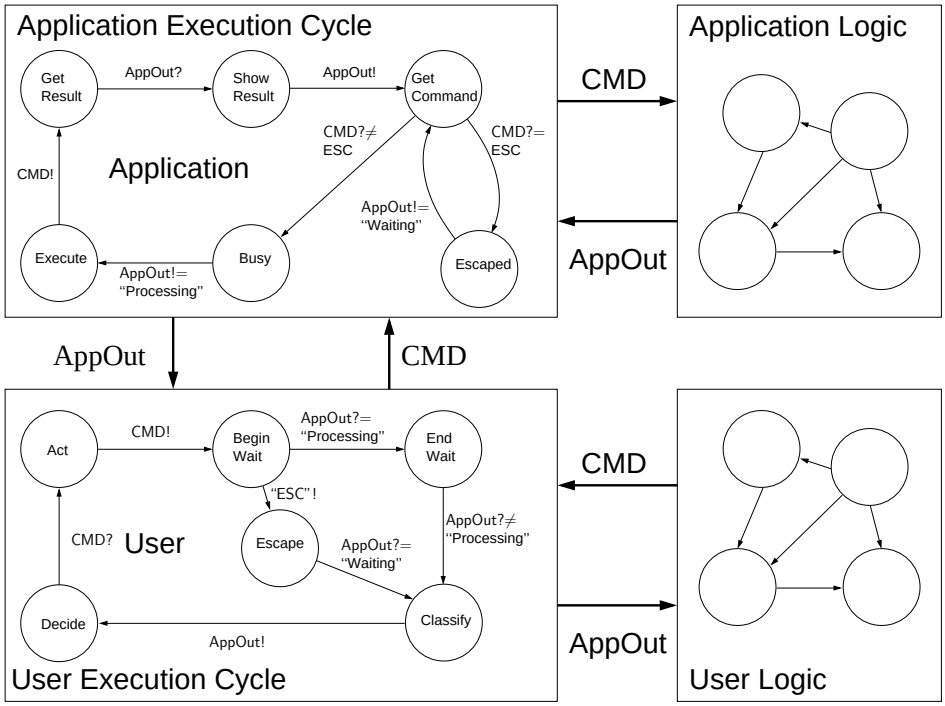

Fig. 7. Refined Model

the next command for the second time, he does not recognize that execution of the first command has not been completed.

The system model is not a model for Formula 1, because the application configuration may change while the user issues a command. This problem is well known from real-world computer systems: if the user does not know if a command has been executed, he may be tempted to re-issue the command, resulting in double execution of the command. In the worst case, this can lead to a security problem, for example when the user accidentally confirms a critical action twice. Next, we show a solution for the problem. In Section 6, we apply the solution to a real-world program specification.

The problem can be solved by introducing new states for synchronization, as shown in Figure 7. In this model, the system gives visual feedback indicating whether it is waiting for user input or processing user input. Once the application received a user command, it shows "processing" on the screen. When processing is finished, the new application status is shown. Just showing the message "processing" while executing user commands is not sufficient, however. Depending on execution speed, the user may not recognize the message "processing" at all (because it was shown for a very short amount of time), our it may take a long time before the message is shown (in case the system is slow). In order to give the user the ability to distinguish between the two cases, an escape-key is introduced. If the user pushes the escape-key, the message "waiting" is shown. This way, if the user does not know about the state of the input process, he can press "escape" and wait for the message "waiting" to show up. The model given in Figure 7 satisfies the integrity

constraint of Formula 1.

While it is perfectly fine to change the specification of the application, one may ask if it is acceptable to change the user model, i.e. our assumptions about the user. We do think this is acceptable. It is common practice to train the user on how to operate a system. For this, a formal user model allows to explicit state what a user has to know in order to operate the system.

6 Specification of Secure Interactive Applications

In Section 5 we showed that the naïve model of user and application interaction is not sufficient to guarantee consistency. While we showed that the refined model guarantees consistency for the given application logic and user logic components, we did not—and can not—show that the consistency constraint holds for all application and user logic components, because it does not solely rely on consistency between the user and the application model; the user must also have the right assumptions about the application model. He must have knowledge about the inner working of the application, and about the consequences of his actions. This knowledge is represented in the user logic module. Just like the user's and application's execution loops, the logic components of user and application are modeled as IOLTS. This requires a state-based representation of the application, and of the user's knowledge about the application.

In the last Section, basic application logic and user logic modules were used in the refutation of the naïve model. These example modules (given in Figure 5), had only two outputs: showA and showB. In actual applications, possible system configurations and outputs are much richer in detail. Even if considering TTY-based applications, we have screens with multiple rows and columns, where each cell can contain an alphanumeric character. Even on a moderately sized screen, the set of all possible combinations of output characters are too large to be modeled explicitly. Therefore, it is necessary to find a suitable abstraction of application states, application output and user assumptions about application states in order to make real-world applications suitable for automated model checking for consistency constraints. Of course, such security-relevant states are abstractions of the application's actual internal configuration, which is much richer in detail. Nevertheless, we assume that these states are the *right* abstraction in that the user has sufficient information about the internal configuration of the application if he or she knows the abstract state of the application.

In the following, our abstract model of an application assumes that an application can be in one of many states, and that the current state is represented in variable `applicConf`. User commands (usually corresponding to keystrokes entered by the user) trigger state transitions. Depending on the result of a command, the system transits into a new state. The actual specifications of command `execution` is application dependent. Pseudo code for

```
1: repeat
2:   {Show Result}
3:   updateScreen(confAsString(applicConf),applicConf)
4:   {Get Command}
5:   repeat
6:     cmd := getKeystroke()
7:     if cmd = ESC then
8:       {Escaped}
9:       updateScreen(confAsString(applicConf) +
                      ''Waiting'',applicConf)
10:    end if
11:  until cmd ≠ ESC
12:  {Busy}
13:  updateScreen(confAsString(applicConf) +
                  ''Processing'',applicConf)
14:  {Execute & Get Result}
15:  applicConf := execute(cmd,applicConf)
16: until cmd = QUIT
```

Algorithm 1. The main event loop

a main event loop implementing the Application Execution Cycle model from Figure 7 is given in Algorithm 1.

For the specification of the screen update function `updateScreen`, we use the following auxiliary functions:

- *confAsString*(`applicConf`) is a string that allows the user to identify the state of the application.
- *screenOutput*(`applicConf`) is a two-dimensional array of characters. It contains the correct screen output corresponding to `applicConf`. The actual definition of `screenOutput` is under the discretion of the application at hand.
- $stringAt(x, y)$ is the string shown on screen position (x, y).

We require that the current state of the application logic component plus optionally the additional information "waiting" or "processing" are shown in the first line of the screen. A specification for function `updateScreen` in OCL [4] is shown in Table 1.

It should be noted that we do not restrict ourselves to a certain application. The specification fits every applications requiring a secure, text-based user interface.

[4] The OCL specification should be understandable without deeper knowledge of OCL. See [25,26] for more information on OCL and [17] for the current language specification.

context	`updateScreen(status,conf)`
post	$stringAt(0,0) =$ `status` and
	$\forall k \in \{1,\ldots,screenHeight-1\}$:
	$stringAt(0,k) =$
	$screenOutput($`applicConf`$)[k-1]$

Table 1
Specification of the application's function for updating the screen contents

7 Verification

In order to verify that an implementation satisfies the consistency constraints, a number of assumptions about the user are necessary:

(i) The user observes the screen.

(ii) The user understands the output of *stateAsString*.

Under these assumptions, it is sufficient to show that the status string as provided by *stateAsString* is adequate, and that *updateScreen* is called as specified in the last Section. With these assumptions and the additional assumption that the operating system works correctly, verification of observability can be split into two parts:

(i) Proofs for the application's functions `execute` and `updateScreen`.

(ii) Proofs about the main event loop in respect to the application model.

The second part is generic, since the main event loop given in Algorithm 1 is applicable to all applications following our design methodology.

It should be noted that two calculi are integrated in our approach. The properties of the main execution loop need to be proven in some temporal calculus. Satisfaction of the requirements for the main execution loop can be proven by model checking. Proofs about the application's functionality can be executed in a calculus based on pre- and postconditions, e.g. Hoare logic. From the proofs about the application's functionality it follows that for each distinct system configuration, `updateScreen` produces a distinct and up-to-date screen representation. From the assumption that the user understands the chosen representation, it follows that observability is given immediately after every call of `updateScreen`.

In project Verisoft (`http://www.verisoft.de`), this approach is used to prove observability of an email client application in the context of a pervasively verified computer system.

8 Conclusions and Future Work

In this paper, we showed how formal methods can be used to guaranteed a fundamental requirement of user interface security.

In Section 4, we translated the generic security requirements of Confidentiality, Integrity, and Availability to human-computer interaction, and we gave a formal definition of Integrity for HCI: The user should always be aware of the current state of the system. In Chapter 5, we developed generic models for TTY-based application. We showed that a naïve approach leads to models that do not guarantee consistency. We provided a refined model that satisfies consistency constraints. In Section 6, we showed how the formal model can be transfered to actual applications, and what has to be shown about an application in order to ensure its security.

In project Verisoft (`http://www.verisoft.de`) our method is used to specify, implement and verify a secure email client. In Verisoft, both the operating system and the application program are formally verified based on that specification.

In the future, we plan to extend our work to other aspects of user interface security. Our goal is to create a systematic formal description of user interface security for interactive systems.

References

[1] G. D. Abowd, J. P. Bowen, A. J. Dix, M. D. Harrison, and R. Took. User interface languages: A survey of existing methods. Technical Report PRG-TR-5-89, Oxford University Computing Laboratory, October 1989.

[2] Jean Berstel, Stefano Crespi Reghizzi, Gilles Roussel, and Pierluigi San Pietro. A scalable formal method for design and automatic checking of user interfaces. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14(2):124–167, April 2005.

[3] Simon P. Booth and Simon B. Jones. A screen editor written in the miranda functional programming language. Technical Report TR-116, Department of Computing Science and Mathematics, University of Stirling, February 1994.

[4] David A. Carr. Interaction object graphs: an executable graphical notation for specifying user interfaces. In Philippe Palanque and Fabio Paternò, editors, *Formal methods in Human-Computer Interaction*, pages 141–155. Springer, 1997.

[5] Geert de Haan. *ETAG, A Formal Model of Competence Knowledge for User-Interface Design*. PhD thesis, Vrije Universiteit, Amsterdam, 2000.

[6] Asaf Degani, Michael Heymann, George Meyer, and Michael Shafto. Some formal aspects of human-automation interaction. Technical report, NASA, Moffett Field, CA: NASA Ames Research Center, 2000.

[7] Rüdiger Dierstein. Sicherheit in der Informationstechnik — der Begriff IT-Sicherheit. *Informatik Spektrum*, 27(4), August 2004.

[8] A. Dix and G. Abowd. Modelling status and event behaviour of interactive systems. *Software Engineering Journal*, 11(6):334–346, 1996.

[9] A. J. Dix and C. Runciman. Abstract models of interactive systems. In P. Johnson and S. Cook, editors, *HCI'85: People and Computers I: Designing the Interface*, pages 13–22. Cambridge: Cambridge University Press, 1985.

[10] Doug Goldson. Formal modelling of interactive systems. In *Proceedings of APAQS 2OOO, the First Asia-Pacific Conference on Quality Software*, IEEE Conference Proceedings. IEEE Computer Society Press, 2000.

[11] Jonathan Grudin. The case against user interface consistency. *Communications of the ACM*, 32(Issue 10):1164–1173, October 1989.

[12] F. Hamilton. Predictive evaluation using task knowledge structures, 1996.

[13] Michael Harrison and Harold Thimbleby, editors. *Formal methods in human-computer interaction.* Cambridge Univ. Press, Cambridge, Mass., 1990.

[14] A. Hussey and D. Carrington. Specifying a web browser interface using Object Z. In Philippe Palanque, editor, *Formal methods in human computer interaction*, chapter 8. Springer, 1998.

[15] Vipul Jain. User interface description formalisms. Technical report, McGill University School of Computer Science, Montréal, Canada, 1994.

[16] Bonnie E. John and David E. Kieras. The GOMS family of user interface analysis techniques: Comparison and contrast. *ACM Transactions on Computer-Human Interaction*, 3(Issue 4):320–351, December 1996.

[17] Object Modeling Group. *Unified Modelling Language Specification, version 1.5*, March 2003.

[18] Philippe Palanque, Remi Bastide, and Valerie Senges. Validating interactive system design through the verification of formal task and system models. In *Engineering for Human-Computer Interaction.* Chapman & Hall, August 1995.

[19] Philippe Palanque and Fabio Paternò, editors. *Formal methods in human computer interaction.* Springer, New York, London, 1998.

[20] Fabio Paternò. *Model-Based Design and Evaluation of Interactive Applications.* Springer, 1999.

[21] Robert W. Reeder and Roy A. Maxion. User interface dependability through goal-error prevention. In *DSN '05: Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 60–69, Washington, DC, USA, 2005. IEEE Computer Society.

[22] John Rushby. Using model checking to help discover mode confusions and other automation surprises. *Reliability Engineering and System Safety*, 75(2):167–177, February 2002.

[23] Bettina Sucrow. Formal specification of human-computer interaction by graph grammars under consideration of information resources. In *Automated Software Engineering*, pages 28–35, 1997.

[24] B. Sufrin. Formal specification of a display editor. *Science of Computer Programming*, pages 157–202, 1982.

[25] Jos Warmer and Anneke Kleppe. OCL: The constraint language of the UML. *Journal of Object-Oriented Programming*, 12(1):10–13,28, March 1999.

[26] Jos B. Warmer and Anneke G. Kleppe. *The Object Constraint Language: Precise Modeling With UML*. Addison-Wesley Professional, 1998.

[27] Peter Wright, Bob Fields, and Michael Harrison. Deriving human-error tolerance requirements from task analysis. In *IEEE International Conference on Requirements Engineering*, 1994.

Formal Models for Informal GUI Designs

Judy Bowen [1,2] and Steve Reeves [3]

Department of Computer Science
University of Waikato
Hamilton, New Zealand

Abstract

Many different methods exist for the design and implementation of software systems. These methods may be fully formal, such as the use of formal specification languages and refinement processes, or they may be totally informal, such as jotting design ideas down on paper prior to coding, or they may be somewhere in between these two extremes. Formal methods are naturally suited to underlying system behaviour while user-centred approaches to user interface design fit comfortably with more informal approaches. The challenge is to find ways of integrating user-centred design methods with formal methods so that the benefits of both are fully realised. This paper presents a way of capturing the intentions behind informal design artefacts within a formal environment and then shows several applications of this approach.

Key words: Formal methods, user-centred design, GUIs, refinement, informal design artefacts.

1 Introduction

When we are designing and building systems, particularly large and complex systems, it is not unusual to work in a modular fashion. Different parts of the system will be worked on at different times, perhaps by different groups of software engineers, designers and programmers.

Separation of the design and implementation of a graphical user interface (GUI) of a system from what we will refer to as the underlying system behaviour is a common and pragmatic approach for many applications. The development of user interface management systems (UIMS) based on the logical separation of system functionality and user interface (UI) is exemplified by the Seeheim model [20]. The separation allows us to not only focus on the

[1] Thanks to the anonymous reviewers for DSV-IS 2006
[2] Email: jab34@cs.waikato.ac.nz
[3] Email: stever@cs.waikato.ac.nz

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

different concerns which different parts of the system development present, but, more importantly, allows for different approaches and design techniques.

When we develop the underlying system functionality for an application we are often concerned with issues such as correctness, reliability, robustness and efficiency *etc.* which lend themselves to the techniques we call "formal". Such formal techniques include specifying requirements, validating and verifying specifications and refinement methods. When we develop UIs, however, our concerns are often more human-focussed (this is particularly true if we follow a user-centred design (UCD) approach). The design techniques we adopt reflect this and rely on more informal strategies such as prototyping, scenarios and storyboards, iteration based on user-feedback, usability testing *etc.*

Whilst we can see the benefits of this separation of concerns and design methods in terms of being able to adopt the most suitable development approach to different parts of the task, there are clearly some problems associated with it. If our aim is to use a formal process to develop provably correct software (which it is), then we must ensure that all parts of the system have been designed in a way which satisfies this.

This gap between the formal and informal has been identified and discussed many times, notably in 1990 by Thimbleby [24]. Several different approaches have been taken over recent years by different groups of researchers to try and bridge this gap. Much of the work that has been done falls into one of the following categories:

- Development of new formal methods for UI design. *E.g.* Modelling UIs using new formalisms [7];
- Development of hybrid methods from existing formal methods and/or informal design methods. *E.g.* Using temporal logic in conjunction with interactors [17];
- Use of existing formal methods to describe UIs and UI behaviour. *E.g.* Matrix algebra for UI design [25];
- Replacing existing human-centred techniques with formal model-based methods. *E.g.* Using UI descriptions in Object-Z [23] to assess usability [12].

Whilst much of this work is demonstrably a step forward in bringing together formal methods and UI design, the methods and techniques which have been developed have failed, in the most part, to become mainstream.

One of the reasons for this seeming reluctance for either group to adopt the new methods proposed is, of course, the reluctance of any group to change working practices which are meeting their individual needs. Persuading users of formal methods to adopt less formal, or new hybrid, methods has proved as unsuccessful as encouraging UI designers to abandon their human-centred approach in favour of more formal approaches.

Rather than trying to change the methods used by different groups of software developers, the approach we are taking with our research is to consider

the existing, diverse, methods being used and develop ways of formally linking them together. In particular, because our interests lie in both using formal methods and rigorous development techniques to develop our software, and UCD approaches to UI design, our intention is to find ways of interpreting the sorts of informal design artefacts produced in a UCD process within our formal framework.

In this paper we will introduce a way of formally describing informal design artefacts, called the *presentation model*. We will give some examples of the use of the presentation model within a formal design context and then show how we can extend this model with another formalism, finite state machines. We can then begin to explore both the static and dynamic meanings of the designs which form the basis of the model.

2 User-Centred Design Artefacts

The purpose of user-centred design is to ensure that the software we build, and in particular the interface to that software, meets the expectations of the intended users. To this end the processes used are designed to involve users from an early stage to find out about not only the tasks they need to perform with the software, but also things like the current working practices of the users, their experience with similar software, internal company working processes that will be affected by this new software, *etc.*

Techniques used early in a UCD process may include ethnographic studies which allow the designers to understand not only the users, but also their work environment and work processes. This may be followed by task analysis methods to examine the users' requirements of the system. Task analysis has received a lot of attention from formal practitioners over the years, and a number of models exist for this, as well as methods for developing UIs from such models, *e.g.* [6], [18]. UCD practitioners may use scenarios and personas to enhance the task analysis process and give details of specialised requirements and user behaviours.

The actual design of the UI may involve brainstorming sessions between designers and users which will lead to the development of prototypes. These prototypes are then tested by both users and design specialists and updated in an iterative process before a final design is reached. Even this final design is subject to amendment once the system has been implemented and subsequently undergone usability testing.

The key to UCD, therefore, is to ensure that the actual users of the system are involved at all stages of the design process. The sorts of artefacts that are generated during such processes reflect this collaborative way of working and will include things like white-board design sessions with post-it notes used to represent interface elements, textual narrative descriptions of things like domain information and scenarios, task analysis models, user descriptions and paper-based prototypes.

One of the problems we face when trying to capture UCD processes within a formal software engineering context is that the artefacts produced are intentionally informal. They aim to encourage users to feel able to participate and change the design, and lo-fidelity artefacts, such as paper prototypes for example, have been shown to be very successful for this purpose.

There have been several methods and tools developed which support prototyping or enable the use of tablet PCs [14], collaborative whiteboards [21] or desktop computers to generate prototypes in a manner similar to paper prototyping [8]. It may be that some, or all, of these tools could be adapted or extended to support the sort of work we are currently doing. However, as our focus is currently on existing commonly used design techniques and artefacts, we have deliberately chosen not to consider such tools here. Instead we focus on lo-fidelity artefacts like paper-based prototypes.

3 Formal Methods and Refinement

When we state that we wish to use formal methods as the basis for our system derivation we mean that we want to build models, at whatever level of abstractness/concreteness is most natural and useful to the developers of the system, which we can investigate with "mathematical" precision. So, typically, we want to build our models (write our specifications) in a language which has well-defined properties: syntax, semantics and logic. Without the first two properties we cannot (without a well-defined syntax) separate the specifications from all the other artefacts, or (without a well-defined semantics) know what a specification means even if we know, syntactically, that we have one.

The third requirement, that we have a logic, is also clearly necessary: being able to build a well-defined specification is a good start, but we also need to be able to precisely investigate that specification, see what its assumptions are, see what properties it has, see what implications for the system arise and so on. For all these necessary things we must have a logic.

So, our requirements are broad, not very onerous and leave developers open to choose whichever language they like to use (making decisions on grounds of familiarity, suitable for the task *etc.*) as long as it has our three properties.

3.1 Refinement

The idea behind refinement is very simple and goes back to Wirth [26]. It is based on the desire to be able to move between different models of a system without having any negative impact on a user's view or feel of the system in terms of its functionality or usability. As a simple example, we might move from a system that uses sets to one that uses arrays: here we move from abstract to concrete, from a convenient and useful idea (sets) to an implementation-oriented one (which probably includes too much detail).

This original idea behind refinement has been generalised so that we can think about not just differing implementations but differing levels of abstraction of model, from specification to implementation.

The basic intuition behind refinement is [9]:

> *Principle of Substitutivity*: it is acceptable to replace one program by another, *provided* it is impossible for a user of the programs to observe that the substitution has taken place. If a program can be acceptably substituted by another, then the second program is said to be a *refinement* of the first.

4 Integration of Techniques

Integration of different languages and models within formal methods is not unusual (indeed this activity has at least one whole conference devoted to it, namely IFM [13]). The central idea is to use the differing features and strengths of the different methods as appropriate. Sometimes it is enough to just use different formalisms to specify different parts or different properties of the system, but the best effect is seen when methods are fully integrated so there are formal links between them allowing for a fully rigorous development.

Our aim is to formally link the formal and informal processes so that we get all of the benefits of rigorous specifications and refinement, namely the ability to prove properties of a system and ensure formally that we meet requirements and end up with a correct implementation, while at the same time benefiting from the informal design methods of a UCD process which ensures we satisfy the user requirements and develop a usable interface.

Using formal methods in GUI design is not a new idea, and many different approaches to this have been taken. These may be along the lines of formalising particular parts of the design process, such as task analysis [19], or describing GUIs in a formal manner [10], or deriving implementations from formal models [11],[7]. However, what we are trying to do is to look at an existing design methodology, *i.e.* user-centred design, examine the types of processes and artefacts that are used and find ways of incorporating these into a formal process.

5 Presentation Model

The presentation model is used to formally capture the meaning of an informal design artefact such as a scenario, storyboard or prototype. It is a deliberately simple model because the informal artefacts it describes are themselves simple and easy to understand. This is important as it makes it easier to encourage others to adopt and use the model. When we talk about the *meaning* of a design artefact we are talking about what the UI described by the informal artefact is supposed to do, *i.e.* if it were transformed into an implementation what its behaviour would be. If we consider a paper-based prototype in isola-

tion its meaning may be ambiguous; it requires some supporting information or context to make clear what is intended.

When a designer shows a prototype to a user, there is a discussion about what the prototype will do when the parts shown are interacted with. This forms what we call the narrative of the prototype, the accompanying story which allows the user to understand how it will work and what the various parts do. This allows a simulated interaction to take place which enables the user and designer to evaluate the suitability of the proposed design. The presentation model is a formal model which describes an informal design artefact in terms of the widgets of the design and captures their meaning. It is deliberately abstract and high-level. The presentation model is not intended to replace the informal design artefact, rather it acts as a bridge between the meaning captured by the design and the formal design process being used for the system functionality. The syntax for presentation models is given next.

5.1 Syntax

$\langle pmodel \rangle ::= \langle declaration \rangle \langle definition \rangle$
$\langle declaration \rangle ::= PModel\{\langle ident \rangle\}^+,$
$\quad WidgetName\{\langle ident \rangle\}^+,$
$\quad Category\{\langle ident \rangle\}^+,$ [4]
$\quad Behaviour\{\langle ident \rangle\}^*,$
$\langle definition \rangle ::= \{\langle pname \rangle is \langle pexpr \rangle\}^+$
$\langle pexpr \rangle ::= \{\langle widgetdescr \rangle\}^+ \mid \langle pname \rangle : \langle pexpr \rangle \mid \langle pname \rangle$
$\langle pname \rangle ::= \langle ident \rangle$
$\langle widgetdescr \rangle ::= (\langle widgetname \rangle, \langle category \rangle , (\{\langle behaviour \rangle\}^*))$
$\langle widgetname \rangle ::= \langle ident \rangle$
$\langle category \rangle ::= \langle ident \rangle$
$\langle behaviour \rangle ::= \langle ident \rangle$

$\{Q\}^+$ *indicates one or more Qs*
$\{R\}^*$ *indicates zero or more Rs*

An example of a legal presentation model is then:

PModel	*p q r*
Widgetname	*aCtrl bCtrl cSel*
Category	*ActionControl SValSelector*
Behaviour	*dAction eAction fAction*

p is $(aCtrl, ActionControl, (eAction\ fAction))$
$(bCtrl, ActionControl, (dAction))$
q is $(cSel, SValSelector, (eAction\ fAction))$

[4] The categories used are based on the work in [2]

r is p : q

This model describes a UI with two components, p and q (where these may be different windows, or different states of the UI). The entire UI (*i.e.* the combination of p and q) is described by r and the : operator acts as a composition. p has two widgets, *aCtrol* and *bCtrl*, which are both *ActionControls*. The behaviours associated with *aCtrl* are *eAction* and *fAction* and for widget *bCtrl* the associated behaviour is *dAction*. q has one widget, *cSel*, which is a *SValSelector* with the behaviours *eAction* and *fAction*. Presentation model r, therefore, is the combination of all of the widgets of p and q and describes the total possible behaviours of the UI.

5.2 Semantics

We can now give the semantics of the model. Firstly, we can describe the complete model of a design as an environment *ENV*.

The environment is a mapping from the name (from the set *Ide* of identifiers) of some presentation model and its parts to their respective values:

$$ENV = Ide \to Value$$
$$Value = Const + \mathbb{P}(Const \times Const \times \mathbb{P}\, Const)$$
$$Const = \{\overline{v} \mid v \text{ is an identifier}\}$$

We use semantic functions to build up the contents of the environment and to describe its structure based on the given syntax.

$$[\![\_]\!] : \langle pmodel \rangle \to ENV$$
$$Dc : \langle declaration \rangle \to ENV$$
$$Df : \langle definition \rangle \to ENV \to ENV$$
$$Expr : \langle pexpr \rangle \to ENV \to ENV$$

$$[\![Decl\ Def]\!] = Df[\![Def]\!](Dc[\![Decl]\!])$$

$$Dc[\![PModel\ \pi_1 \,..\, \pi_{n1}\ WidgetName\ \alpha_1 \,..\, \alpha_{n2}\ Category\ \epsilon_1 \,..\, \epsilon_{n3}\ Behaviour\ \beta_1 \,..\, \beta_{n4}]\!] = \{\pi_i \mapsto \overline{\pi_i}\}_1^{n1} \cup \{\alpha_i \mapsto \overline{\alpha_i}\}_1^{n2} \cup \{\epsilon_i \mapsto \overline{\epsilon_i}\}_1^{n3} \cup \{\beta_i \mapsto \overline{\beta_i}\}_1^{n4}$$

where $\{e_i\}_1^k$ is shorthand for the set $\{e_1, e_2, .., e_k\}$

$$Df[\![D\ Ds]\!]\rho = Df[\![Ds]\!](Df[\![D]\!]\rho)$$
$$Df[\![P\ is\ \psi]\!]\rho = \rho \oplus \{P \mapsto Expr[\![\psi]\!]\rho\}$$

where ρ represents the current environment.

$$Expr[\![E\ Es]\!]\rho = Expr[\![E]\!]\rho \cup Expr[\![Es]\!]\rho$$
$$Expr[\![\psi : \phi]\!]\rho = Expr[\![\psi]\!]\rho \cup Expr[\![\phi]\!]\rho$$
$$Expr[\![(N\ C\ (b_1 \,..\, b_n))]\!]\rho = \{(\rho(N)\, \rho(C)\, \{\rho(b_1) \,..\, \rho(b_n)\})\}$$
$$Expr[\![I]\!]\rho = \rho(I)$$

Fig. 1. Design for Mobile Phone Application UI

Our presentation models consist of widgets with names, categories and behaviours. In our semantics we have shown how the syntax of the model creates mappings from identifiers to constants in the environment (which represents the design that the model is derived from). The presentation model semantics is a conservative extension of set theory, that is, everything which is provable about presentation models from the semantics is already provable in set theory using the definitions given in the semantic equations. This then allows us to rely on the existing sound logic of set theory to derive a necessarily sound logic for our presentation models.

Next we provide an example of a UI design and presentation model of that design which we will use to illustrate the uses and extensions for presentation models.

6 Example

The following example is an adaptation of an example given by Calvery *et al.* in [5] and [4]. The example involves a home heating control system which is accessible via several different devices, namely a home-based, wall-mounted control panel, a web-server running on a standard PC, a PDA and a WAP-enabled mobile phone. The control system supports the monitoring and control of temperatures in a number of different rooms as well as overall adherence to ambient temperature levels. For the purposes of our example we use an amended version of the mobile phone application UI which allows us to illustrate our particular points.

A proposed UI design for the mobile phone version of the system is given in Figure 1. This shows the four different screens which make up the UI for the application which we label $C1$, $C2$, $C3$ and $C4$ respectively. The presentation

model for the mobile phone UI design follows (some detail has been omitted for brevity):

PModel	*MPHeat MPMenu MPBed MPLounge MPBath*
Widgetname	*BathSelect LoungeSelect BedSelect QuitOpt IncBathOpt DecBathOpt IncLoungeOpt DecLoungeOpt IncBedOpt DecBedOpt AcceptOpt CancelOpt BathTempDisp BathRangeDisp LoungeTempDisp LoungeRangeDisp BedTempDisp BedRangeDisp*
Category	*ActCtrl SValSel SValRespdr*
Behaviour	*ShowBath ShowLounge ShowBed QuitApp IncBathTemp DecBathTemp IncLoungeTemp DecLoungeTemp IncBedTemp DecBedTemp StoreSettings ShowMenuPage DispBathTemp DispBathRange DispBedTemp DispBedRange DispLoungeTemp DispLoungeRange*
MPMenu is	*(BathSelect,ActCtrl,(ShowBath))*
	(LoungeSelect, ActCtrl,(ShowLounge))
	(BedSelect, ActCtrl,(ShowBed))
	(QuitOpt, ActCtrl,(QuitApp))
MPBed is	*(BedTempDisp, SValRespndr,(DispBedTemp))*
	(BedRangeDisp, SValRespndr,(DispBedRange))
	(IncBedOpt, SValSel,(IncBedTemp))
	(DecBedOpt, SValSel,(DecBedTemp))
	(AcceptOpt, ActCtrl,(StoreSettings))
	(CancelOpt, ActCtrl,(ShowMenuPage))
MPLounge is	... omitted
MPBath is	... omitted
MPHeat is	*MPMenu : MPBath : MPLounge : MPBed*

7 Using the Presentation Model

7.1 Presentation Models and Refinement

Our first use for the presentation model is to enable us to include the design of the UI in our formal refinement process. We have previously given a detailed account of this process [3] and it is not our intention to repeat these details here. However we will give an outline of the process and direct the interested reader to [3].

We have talked about a relationship between the activities of our formal and informal design processes. We start to define this at the first design activity for each method, *i.e.* requirements gathering for the formal specification and determining user requirements for the UI design. Rather than treating these two activities independently we need to ensure that the information

gathered from each is used together to produce a specification and UI requirements which are not only fully inclusive, but which share a vocabulary and compliment each other. We create a formal specification for a system and use naming conventions which indicate which of the given operations are user operations, that is, which of the operations upon the system state should be made available directly to the user via the UI.

If we were to create a formal specification for the heating application, based on the requirements given in [5] and using the specification language Z [1], we would follow this convention and name the operations we describe accordingly. For example, in order to describe the requirements of a user to be able to increase the temperature in any of the rooms, we would expect to see the following in our specification (using the Z idiom of *promotion*):

$$
\begin{array}{|l}
\hline
Room \\
CurrentTemp : TEMP \\
\hline
CurrentTemp \leq MaxTemp \\
CurrentTemp \geq MinTemp \\
\hline
\end{array}
$$

$$TempControlSystem \mathrel{\widehat=} [rooms : RID \pfun Room]$$

$$
\begin{array}{|l}
\hline
\Phi UpdateRoomTemp \\
\Delta TempControlSystem \\
\Delta Room \\
rid? : RID \\
\hline
rid? \in \operatorname{dom} rooms \\
\Theta Room = rooms\ rid? \\
rooms' = rooms \oplus \{rid? \mapsto \Theta Room'\} \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline
IncreaseRoomTemp \\
\Delta Room \\
newTemp? : TEMP \\
\hline
newTemp \geq MinTemp \\
newTemp \leq MaxTemp \\
CurrentTemp' = newTemp \\
\hline
\end{array}
$$

$$USER\_IncRoomTemp \mathrel{\widehat=} \exists \Delta Room \bullet \Phi UpdateRoomTemp \land IncreaseRoomTemp$$

As part of our refinement process we need to ensure that all user operations described in the specification have been described in the UI design. That is, we should ensure that the system informally described by the GUI design is a refinement of the specification. From the presentation model of the design we can produce a Z description (using the framework for describing widget categories in Z given in [2] as the basis). From the presentation model of the mobile phone heating application we can derive such a Z description. If we focus on the requirement to increase the temperature of the bedroom we can look at one part of the derived Z description which is:

IncBedTempOp

$\Delta Bedroom$
$iActValue? : TEMP$
$iAction? : ACTION$

$iAction? = IncBedTemp \Rightarrow$
$\quad bedTemp' = iActValue?$
$iAction? \neq IncBedTemp \Rightarrow$
$\quad bedTemp' = bedTemp$

IncBedTempSelCtrl

$iCState : CONTROLSTATE$
$iSelValue : TEMP$
$iAction! : ACTION$
$iActValue! : TEMP$

$iCState = Active \Rightarrow$
$\quad iAction! = IncBedTemp$
$iCState = NotActive \Rightarrow$
$\quad iAction! = NoAction$
$iActValue! = iSelValue$

$$ActiveIncBedTemp \mathrel{\widehat=} [IncBedTempSelCtrl \mid iCState = Active] \gg IncBedTempOp$$

The presentation model for this example describes a widget *BedTempDisplay* whose category is *SValResponder*. If we refer to [2] we see that to describe such a widget in Z we must provide a schema with observations on active state (which captures the notion of user interaction), selected values and behaviours, and then link this (using the Z piping notation) to an operation schema which describes the associated behaviour (in this case setting the temperature to a new value).

We can now use the standard simulation techniques (which are based on [27] and [9]) to show that a refinement holds between the UI design and the specification.

7.2 Presentation Models and Design Equivalence

Following on from the use of presentation models in refinement we have derived a notion of equivalence between designs, based again on the presentation model. The intention here is to be able to take different UI designs (for the same system) and using the presentation models of these designs determine if they can be considered in some way equivalent.

Design equivalence is important during a refinement process, where rapid iteration of designs means it may be more practical to require proof of refinement back to the specification only when the design changes significantly, *i.e.* when it is no longer functionally equivalent to the previous version of the design. We define this notion of functional equivalence next.

The functionality of a design is given by the set of behaviours of the presentation model of that design. So if we wish to compare two different UI designs to determine whether or not they have the same functionality, then we can simply compare the corresponding behaviour sets of their presentation models. Formally we state:

Definition 7.1 If *DOne* and *DTwo* are UI designs and *PMOne* and *PMTwo* are their corresponding presentation models then:

$$DOne \equiv_{func} DTwo =_{df} Behaviours[PMOne] = Behaviours[PMTwo]$$
$$Behaviours[P] =_{df} \{[\![P]\!]b \mid b \in act(P)\}$$

$act(P)$ is a syntactic function that returns all identifiers for behaviours in P.

Design equivalence is also important in cases where we are designing several interfaces for different versions of a system, as we are in the home heating system example. We want to be sure that the different versions of the system provide the user with the same functionality. Again, we could prove this by using refinement techniques from each of the different system designs back to the specification. However, design equivalence provides a weaker approach to refinement which allows us to ensure that the intended behaviour (both system functional behaviour and UI functional behaviour) is provided by all of the UIs.

As well as functional equivalence we have considered other types of equivalence which exist between designs, namely *Component Equivalence* and *Isomorphism*. We will not go into the details of these types of equivalence here as they are beyond the scope of this paper.

7.3 Presentation Models and Design Consistency

The third use of presentation models we present here is their use in ensuring consistency between designs. Consistency is an important principle of UI design. Shneiderman [22] includes consistency as one of his eight golden rules for interface design:

> Strive for consistency.
> Consistent sequences of actions should be required in similar situations; identical terminology should be used in prompts, menus, and help screens; and consistent commands should be employed throughout.

An application may consist of numerous different screens and dialogues, so maintaining consistency throughout is not a trivial task. One of the things we can ensure, using the presentation model, is that controls which have the same function have the same name (so the user does not have to remember that in one part of the interface they use *Quit* to exit the interface and in another they use *Close*). Conversely we can also check, again using the model, that controls with the same name have the same function and this ensures that the user always knows what to expect when they encounter such a control.

8 Limitations and Extensions

We have provided some examples of how we can use presentation models of informal designs to not only help with our aim of integration of informal

design artefacts into our formal process (via the refinement mechanism and equivalence), but also in dealing with design concerns such as consistency.

There is, however, more to say on the subject of refinement. While we may be able to prove (or disprove) that particular functionalities are included in a UI design (because they are a behaviour of one of the widgets) this is not enough to imply refinement.

As we have already stated, a UI may consist of many different windows and dialogues. The mechanics of moving between these different windows or dialogues are included in the UI-functionality of the design (these are the behaviours which do not correspond to underlying system functions, but instead are used to change the state of the UI). For a refinement to hold between a specification and a design we need to ensure that not only do the required system functions exist in some part of the UI design, but also that they are reachable via some UI function.

Proving the property of reachability in a UI is a common concern in much of the early work on using formal methods with UIs. Rather than trying to adapt and incorporate an existing technique for this into our process, we want to be able to use our presentation model for this purpose also.

The problem with trying to capture the idea of dynamic change of the UI via the presentation model is that that the model gives us a static view of the design. It describes a total environment given by the design (which we can consider to be all of the possibilities of that design), but the (deliberately) simple use of a triple for each widget does not hold enough information to extend its use to dynamic behaviour. One possible solution to this would be to extend the model with additional information. However, we want to avoid making it so complex that it becomes a burden upon designers or formal practitioners to learn and use. We have decided to use another common formalism, in conjunction with the presentation model, in order to be able to prove these more dynamic properties. The formalism that we have chosen is that of Finite State Machines (FSM).

FSM have been used previously for GUI modelling in both design (as early as the late 1960's [16]) and as a way of evaluating interfaces [15]. One of the drawbacks with using FSM in this way is the known problem of state explosion, where the number of states of the machine becomes intractably large. Given the complexity of modern UIs this is certainly a concern and potential problem whenever we try and use FSM to model GUIs or GUI behaviour. However, because we already have an abstraction of the UI (the presentation model) we can use this in conjunction with a FSM and in most cases we produce a FSM which require only a very small number of states. We produce a FSM which is at a high level of abstraction and decorate it with presentation models which provide the lower-level meaning.

Our FSM consists of: a finite set of states, Q; a finite set of input labels, $\sum$; a transition function, δ, which takes a state and an input label and returns a state ($q \rightarrow a \rightarrow q'$); a start state, q_0, one of the states in Q. The FSM is

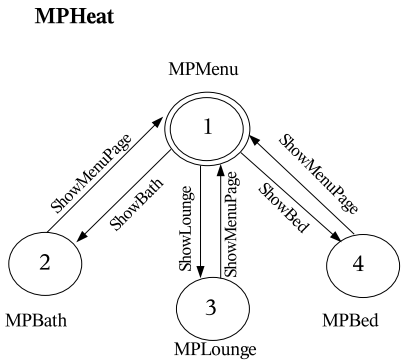

Fig. 2. PIM for MPHeat Presentation Model

then a four tuple $(Q, \sum, \delta, q_0)$.

Each of the states in Q is associated with the name of a presentation model in the overall model which the FSM describes. When the FSM is in a particular state then the presentation model associated with that state is the currently active one, *i.e.* the part of the UI described in that model is visible to the user and available for interaction. We can clearly extend the definition of FSM to a quintuple and add a mapping from state names to presentation model names to formally show this association. The input labels in $\sum$ are themselves the names of behaviours taken from the behaviour sets of the presentation models. In this way we can associate the UI functionality of parts of the design with the dynamic behaviour which makes available different parts of the interface to the user. We call the combination of presentation model and FSM in this way a presentation and interaction model (PIM).

We give a definition of well-formedness for our FSM as follows:

> A PIM of a presentation model is well-formed iff the labels on transitions out of any state are the names of behaviours which exist in the behaviour set of the presentation model which is associated with that state.

Using the notation for our FSM we can give this more formally as:

$$\forall (q, t, q') : \delta \bullet \exists\, b \in act(q_{PModel}) \bullet t = b$$

where q_{PModel} is the presentation model associated with state q.

If we return again to the design of the mobile phone-based application for the heating example, given in Figure 1, we can see that before we can consider the reachability of functions of this UI we need to capture the way in which we move from one part of the interface to another. It is common for prototypes to be annotated to include this sort of information (or in the case of storyboards this is implicit in the flow of the diagrams).

We capture this implied dynamic behaviour between parts of the UI using a FSM which we decorate with parts of the presentation model to give meaning. Figure 2 gives the PIM for the design of Figure 1.

Now, in order to show that a particular behaviour is reachable we first

need to show that the part of the UI it is in (*i.e.* the component presentation model which includes this behaviour in its set of behaviours) is itself reachable in the FSM, and this can be shown using standard FSM methods.

9 Conclusion

In this paper we have described how to aid the integration of formal methods with informal UI design methods. This approach involves creating a formal model of informal design artefacts in a way which allows us to then use them in formal processes.

We have described the presentation model, which formally captures an informal UI design, and discussed how we can use this to include designs in a formal refinement process as well as for design equivalence and consistency checking. The presentation model allows us to capture static properties of a UI design and we have subsequently shown how we can use this with another formalism, FSM, to capture dynamic UI behaviour based on UI functions which change the available functionality of the UI for a user, giving PIMs.

The main advantage we propose for the presentation model and the methods we have shown is that they work in conjunction with existing methods being used by formal practitioners and designers. We do not require that these groups abandon their existing methods and techniques, but rather enhance these with a relatively straight-forward formalism and set of techniques which work alongside, rather than replace, their existing methods.

This paper is designed to give an overview of our work and techniques, rather than going into detail about one particular part of it. We have described our general work in this area and where appropriate we have referred the reader to more detailed accounts in prior publications.

Considerably more work has been done in the area of formal methods and UI design than we are able to give account of in this paper. We have tried to reference appropriate works, and explain the difference in our approach where relevant, but these references should by no means be considered an exhaustive list.

References

[1] ISO/IEC 13568. *Information Technology—Z Formal Specification Notation—Syntax, Type System and Semantics.* Prentice-Hall International series in computer science. ISO/IEC, first edition, 2002.

[2] Judy Bowen. Formal specification of user interface design guidelines. Masters thesis, Computer Science Department, University of Waikato, 2005.

[3] Judy Bowen and Steve Reeves. Formal refinement of informal GUI design artefacts. In *Proceedings of the Australian Software Engineering Conference (ASWEC'06)*, pages 221–230. IEEE, 2006.

[4] G. Calvary, J. Coutaz, and D. Thevenin. Supporting context changes for plastic user interfaces: A process and a mechanism. In A. Blandford, J. Vanderdonckt, and P. Gray, editors, *Joint Proceedings of HCI'2001 and IHM'2001*, pages 349–363. Springer-Verlag, 2001.

[5] Gaelle Calvary, Joèlle Coutaz, and David Thevenin. A unifying reference framework for the development of plastic user interfaces. In *EHCI '01: Proceedings of the 8th IFIP International Conference on Engineering for Human-Computer Interaction*, pages 173–192, London, UK, 2001. Springer-Verlag.

[6] Francesco Correani, Giulio Mori, and Fabio Paternò. Supporting flexible development of multi-device interfaces. In *EHCI/DS-VIS*, pages 346–362, 2004.

[7] Antony Courtney. Functionally modeled user interfaces. In *Interactive Systems. Design, Specification, and Verification. 10th International Workshop DSV-IS 2003, Funchal, Madeira Island (Portugal) J. Joaquim, N. Jardim Nunes, J. Falcao e Cunha (ed.)*, pages 107–123. Springer Verlag Lecture Notes in Computer Science LNCS, 2003.

[8] Adrien Coyette, Stéphane Faulkner, Manuel Kolp, Quentin Limbourg, and Jean Vanderdonckt. Sketchixml: towards a multi-agent design tool for sketching user interfaces based on usixml. In *TAMODIA '04: Proceedings of the 3rd annual conference on Task models and diagrams*, pages 75–82, New York, NY, USA, 2004. ACM Press.

[9] John Derrick and Eerke Boiten. *Refinement in Z and Object-Z: Foundations and Advanced Applications*. Formal Approaches to Computing and Information Technology. Springer, May 2001.

[10] David J. Duke, Bob Fields, and Michael D. Harrison. A case study in the specification and analysis of design alternatives for a user interface. *Formal Asp. Comput.*, 11(2):107–131, 1999.

[11] D. F. Gieskens and J. D. Foley. Controlling user interface objects through pre- and postconditions. In *Proc. of CHI-92*, pages 189–194, Monterey, CA, 1992.

[12] A. Hussey, I. MacColl, and D. Carrington. Assessing usability from formal user-interface designs. Technical Report TR00-15, Software Verification Research Centre, The University of Queensland, 2000.

[13] IFM05. http://www.win.tue.nl/ifm/, 2005.

[14] J. Landay. Silk: Sketching interfaces like krazy. In *Human Factors in Computing Systems (Conference Companion), ACM CHI '96, Vancouver, Canada, April 13–18*, pages 398 – 399, 1996.

[15] A. Paiva, N. Tillmann, J. Faria, and R. Vidal. Modeling and testing hierarchical GUIs. In *D. Beauquier, E. Borger, and A. Slissenko, editors, ASM05.* Universite de Paris, 2005.

[16] David L. Parnas. On the use of transition diagrams in the design of a user interface for an interactive computer system. In *Proceedings of the 1969 24th national conference*, pages 379–385. ACM Press, 1969.

[17] F. M. Paternò, M.S. Sciacchitano, and J. Lowgren. A user interface evaluation mapping physical user actions to task-driven formal specification. In *Design, Specification and Verification of Interactive Systems*, pages 155—173. Springer Verlag, 1995.

[18] Fabio Paternò. Task models in interactive software systems. *Handbook of Software Engineering and Knowledge Engineering*, 2001.

[19] Fabio Paternò. Towards a UML for interactive systems. In *EHCI '01: Proceedings of the 8th IFIP International Conference on Engineering for Human-Computer Interaction*, pages 7–18, London, UK, 2001. Springer-Verlag.

[20] G. E. Pfaff. *User Interface Management Systems.* Springer-Verlag New York, Inc., 1985.

[21] Beryl Plimmer and Mark Apperley. Computer-aided sketching to capture preliminary design. In *CRPIT '02: Third Australasian conference on User interfaces*, pages 9–12, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc.

[22] Ben Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction.* Addison Wesley Longman Inc, 3rd edition, 1998.

[23] Graeme Smith. *The Object-Z Specification Language.* Kluwer Academic Publishers, 2000.

[24] H. Thimbleby. Design of interactive systems. *The Software Engineer's Reference Book*, 1990.

[25] Harold Thimbleby. User interface design with matrix algebra. *ACM Trans. Comput.-Hum. Interact.*, 11(2):181–236, 2004.

[26] Niklaus Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227, April 1971.

[27] J. Woodcock and J. Davies. *Using Z: Specification, Refinement and Proof.* Prentice Hall, 1996.

Towards a Common Semantic Foundation for Use Cases and Task Models

Daniel Sinnig [a,1,2] Patrice Chalin [a,3] Ferhat Khendek [b,4]

[a] *Department of Computer Science and Software Engineering*
Concordia University
Montreal, Canada

[b] *Department of Electrical and Computer Engineering*
Concordia University
Montreal, Canada

Abstract

Use cases are the notation of choice for functional requirements documentation, whereas task models are used as a starting point for user interface design. In this paper, we motivate the need for an integrated development methodology in order to narrow the conceptual gap that exists between software engineering and user interface design. A prerequisite is the definition of a common semantic framework. With respect to the definition of a suitable semantic domain, we discuss core requirements and review related work. A preliminary approach based on (sets of) partially ordered sets is presented. A mapping from CTT task models and use case graphs to the before-mentioned formalism is proposed.

Key words: Use Cases, Task Models, Scenarios, Semantics, Posets

1 Introduction

Unfortunately in current practice, functional requirements specification and UI design are neither harmonized nor coordinated. Instead of having a unique process, where UI design follows as a logical progression from functional requirements specification, both entities are treated rather independently. In particular, it has been noted that most UI design methods are not very well

[1] This work is partially supported by the National Sciences and Engineering Research Council of Canada.
[2] Email: d_sinnig@encs.concordia.ca
[3] Email: chalin@encs.concordia.ca
[4] Email: khendek@ece.concordia.ca

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

integrated with standard software engineering practices [23]. In fact, UI design and the engineering of functional requirements are often carried out by different people following different processes.

There exists a relatively large conceptual gap between software engineering and UI development with both disciplines having their own models, lifecycles and theories. The following two issues follow directly as a result of this lack of integration:

- Developing UI-related models and software engineering models independently neglects existing overlaps, gives rise to redundancies and increases the maintenance overhead.
- Deriving the implementation from UI-related models and software engineering models towards the end of the lifecycle is problematic as both processes do not commence from the same specification and thus may result in inconsistent designs.

In this paper, we present preliminary results from an ongoing research project which has as a main goal: bridging the conceptual gap between software engineering and UI design by formally integrating use cases and task models. While use cases are the method of choice for the purpose of functional requirements documentation [4], UI design typically starts with the identification of user tasks, and environmental requirements [20]. None-the-less, use cases and task models share many similarities. We demonstrate this (in part) by the presentation of a common semantic framework for both models.

The remainder of this paper is structured as follows. Section 2 gives an informal introduction to our framework. In Section 3 we review and compare use cases and task models. Section 4 discusses related work with respect to the definition of semantics of scenario-based notations. It is also in this section that we outline core requirements for a common semantic model and present our approach. We conclude and provide an outlook of future work in Section 5.

2 Overall Framework

Our overall research goal has been to define an integrated *methodology* for the development of use cases and task models within an overall software engineering process. **A key** objective of this initiative is the definition of a formal framework for handling use cases and task models at the requirements and design levels. The cornerstone for such a formal framework is a common semantic domain for both notations.

The common semantic domain is the essential basis for the formal definition of a *satisfiability* relation. Such a relation allows us to make (formal) semantic links between successive refinements of use cases and task models. Refinements, and proofs of *satisfiability*, would ideally be aided by tools, supporting the verification. Such tools typically follow two main approaches:

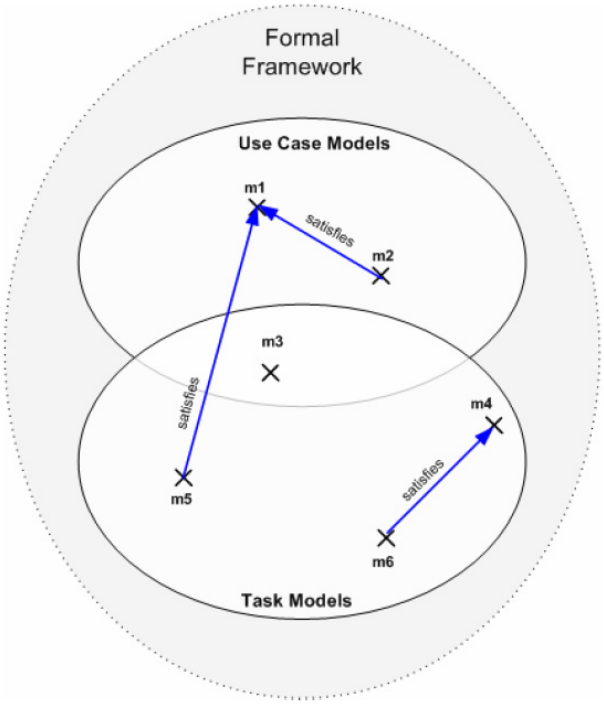

Fig. 1. Relating Use Cases and Task Models within a Formal Framework

Automatic verification (i.e. model-checking, automatic theorem proving) and manual verification (interactive theorem proving). But even without tools, an informal application of the *satisfiability* relation can serve as a rigorous basis for identifying simple traceability links (as is commonly done in software engineering) among the artifacts. Figure 1 visualizes our idea of having a general notion of *satisfiability* that applies equally well between artifacts of a similar nature (e.g. two use cases) as it does between use cases and task models.

3 Background

In this section we remind the reader of the key characteristics of use cases and task models. For each notation we provide definitions, and an illustrative example. Finally, both notations are compared and main commonalities and differences are contrasted.

3.1 Use Case Models

Use cases were introduced roughly 15 years ago by Jacobson. He defined a use case as a *"specific way of using the system by using some part of the functionality"* [8]. More recent popularization of use cases is often attributed by Cockburn [4]. Use case modeling is gradually making its way into mainstream practice which sees it as a key activity in its software development process (e.g. Rational Unified Process) and as a result, there is accumulating evidence of

Use Case: Order Product

Goal: Customer places an order for a specific product.
Level: User-goal
Primary Actor: Customer
Pre-conditions: The primary actor is logged into the system

Main Success Scenario:

1. Customer actor indicates that he/she wants to search for a specific product.
2. Customer selects the product category and optionally the desired brand and model.
3. System displays search results that match the customer supplied criteria.
4. Customer selects a specific product and then specifies the desired quantity.
5. System confirms availability of the product (in the requested quantity) and displays the purchase summary.
6. Customer selects the method of payment and enters the corresponding account information.
7. System interacts with the **payment authorization system** to carry out the payment.
8. System informs the Primary actor that the order has been confirmed.
9. Customer acknowledges. *{Use case ends.}*

Extension Points:

*a. **Customer indicates that he/she wishes to cancel the order**
*a1. *{Use case ends}*
4a. **Customer indicates that he/she wishes to do another product search:**
4a1. Use case *{Use case resumes at step 1}*.
5a. **The desired product is not available in sufficient quantities.**
5a1. System informs the customer that product is not available in desired quantity.
5a2. *{Use case ends.}*
8a. **The payment information is invalid:**
8a1. System informs the customer that payment information provided is invalid.
8a2. *{Use case resumes at step 6}*

Fig. 2. Example Use Case for "Order Product"

significant benefits to customers and developers [14].

A use case captures the interaction between actors and the system under development. It is organized as a collection of related success and failure scenarios that are all bound to the same goal of the primary actor [11]. Use cases are typically employed as a specification technique for capturing functional requirements. They document the majority of software and system requirements and as such, serve as a contract (of the envisioned system behavior) between stakeholders [4]. In current practice, use cases are promoted as structured textual constructs written in prose language. While the use of narrative languages makes use cases modeling an attractive tool to facilitate communication among stakeholders, prose language is well known to be prone to ambiguities and leaves little room for advanced tool support.

As a concrete example of a use case, Figure 2 presents a detailed user-goal level use case for "Ordering a Product". A use case starts with a "header" section containing various properties of the use case. The core part of a use case is its *main success scenario*, which follows immediately after the header. The main success scenario consists of a sequence of interaction steps between the user and the system. The interaction steps indicate the most common

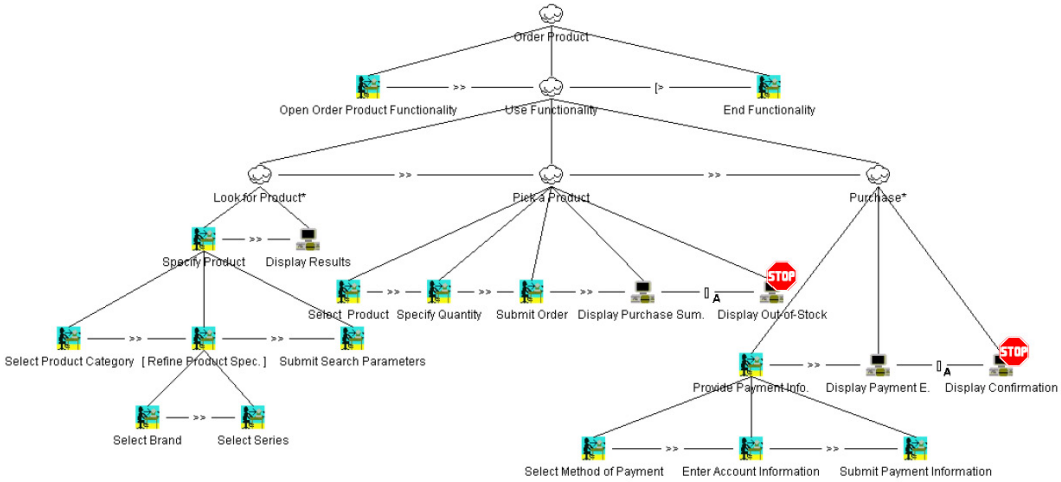

Fig. 3. "Order Product" Task Model

way in which the primary actor can reach his/her goal by using the system.

The use case is completed by specifying the use case *extensions*. These extensions constitute alternative scenarios which may or may not lead to the fulfillment of the use case goal. They represent alternative (and sometimes exceptional) behavior (relative to the main success scenario) and are indispensable to capturing full system behavior. Each extension starts with a condition (relative to one or more steps of the main success scenario), which makes the extension relevant and causes the main scenario to "branch" to the alternative scenario. The condition is followed by a sequence of action steps, which may lead to the fulfillment or the abandonment of the use case goal and/or further extensions. From a requirements point of view, exhaustive modeling of use case extensions is an effective requirement elicitation device.

3.2 Task Models

User task modeling is by now a well understood technique supporting user-centered UI design [18]. In most UI development approaches, the task set is the primary input to the UI design stage. User task models describe the tasks that users perform using the application, as well as how the tasks are related to each other. The origin of most task modeling approaches can be traced back to activity theory [10], where a human operator carries out activities to change part of the environment (artifacts) in order to achieve a certain goal [5].

Like use cases, task models describe the user's interaction with the system. The primary purpose of task models is to systematically capture the *way* users achieve a goal when interacting with the system [24]. Different presentations of task models exist, ranging from narrative task descriptions, work flow diagrams to formal hierarchical task descriptions.

Figure 3 shows an adapted ConcurTaskTreesEnvironment (CTTE) [16]

visualization of the user task model. CTTE is a tool for graphical modeling and analyzing ConcurTaskTrees (CTT) models [17]. The figure illustrates the hierarchical break down and the temporal relationships between tasks involved in the "Order Product" functionality (depicted in the use case of Section 3.1). More precisely the depicted task model specifies how the user makes use of the system to achieve his/her goal but also indicates how the system supports the user tasks. An indication of task types is given by the used symbol to represent tasks. The task model is organized as a directed graph. Tasks are hierarchically decomposed into sub-tasks and atomic actions. Leaf tasks are also called actions, since they are the task that actually carried out by the user or the system. The execution order of tasks is determined by temporal operators that are defined between peer tasks. Various temporal operators exist; the most popular are: *enabling* (>>), *choice* ([]), *concurrency* (|||), and *disabling* ([>). A complete list of the CTT operators together with definition of their interpretation can be found in [17].

We note that the binary temporal operator used between the tasks "Display Purchase Summary" and "Display out of Stock" and between the tasks "Display Confirmation" and "Display Payment Error" is not part of CTT. We have introduced the operator as an extension to CTT. It is called the *Abort Choice* operator and is represented by the symbol ($[]_A$) and the *STOP* sign hovering over its right operand. The interpretation of the *Abort Choice* operator is similar to the build-in *Choice* operator, in the sense that either the task specified by the first operand or the task specified by the second operand is executed. However, after the execution of the second operator all tasks of the model become disabled. Hence, no more tasks can be executed and the scenario ends.

Main motivation for the introduction of this temporal operator was the fact that without it we were not able to conveniently implement the flow specified in the "Order Product" use case as a CTT task model. Particularly problematic are the use case steps which prematurely lead to termination. The only way to simulate such an effect in a CTT task model is to create several main alternative branches in the task tree. One branch represents the case when the "Order Product" is completely performed, whereas the other branches represent cases when the task terminates prematurely. Such a modeling however, creates a significant amount of duplication (since identical starting tasks would be repeated in each brand) and unnecessarily increases the complexity of the visualization of the task tree.

3.3 Use Cases vs. Task Models: A Comparison

In the previous two sections, the main characteristics of use cases and task models were discussed. In this section, we compare both approaches and outline noteworthy differences and commonalities.

Both, use cases and task models, belong to the family of scenario-based no-

tations and as such capture sets of usage scenarios of the system. On the one hand, a use case specifies system behavior by means of a main success scenario and any corresponding extensions. On the other hand, a task model specifies system interaction within a single "monolithic" task tree. In theory, both notations can be used to describe the same information. In practice however, use cases are mainly employed to document functional requirements whereas task models are used to describe UI requirements/design details. Taking this perspective, use cases capture requirements at a higher level of abstraction whereas task models are more detailed. Hence, the atomic actions of the task model are often lower level UI details that are irrelevant (actually contraindicated [4]) in the context of a use case.

The above mentioned difference is manifest in the use case and task model provided as examples. Compared to the "Order Product" use case the corresponding "Order Product" task model has more UI details as it contains steps that are pertinent to a graphical UI. For example the task model contains additional tasks which deal with the submission of selected or entered values (e.g. "Submit Search Parameters" or "Submit Payment Information"). These steps are not specified in the corresponding use case, as they are geared to a UI which requires an extra submission step as a confirmation for a data input. In addition, some of the use case steps (which are the smallest possible units of a use case) have been split into even smaller action tasks in the task model. For example, use case step 2 corresponds to one connected user activity, which however needs to be supported by three UI elements capturing the input of the Product Category, Series and Brand.

In many cases however, a use case will contain (behavioral) information that is not present in the task model. Task models concentrate on aspects that are relevant for UI design and as such, their usage scenarios are strictly depicted as input-output relations between the user and the system. Interactions with secondary actors (which are specified in the use case model) are omitted since they are irrelevant for UI design. An example of this is use case step 7 ("System interacts with the payment authorization system to carry out the payment") of the "Order Product" use case of Figure 2.

4 Semantic Domains for Use Cases and Task Models

In this section we begin with a review of formalism used for scenario-based notations, and thus, those most likely to serve as a common foundation for use cases and task models. This is followed by a discussion of the *requirements* that would need to be addressed by a common semantic framework for use cases and task models. Finally, we present our proposed semantic domain, which is based on partial order sets.

4.1 Related work

Within the domain of scenario-based notations the behavioral aspects of a system (capturing the ordering and the progression of events) play a pivotal role. While several different formalisms have been proposed for scenario-based notations, in what follows we briefly discuss three prominent approaches, namely: process algebras, partial order sets and graph structures.

A formalism that has been widely used to define *interleaving* semantics of scenario-based notations is process algebras. In this approach, the behavior of a system is modeled by a set of (possibly concurrently running) processes. The formalism itself is presented as a formal calculus (which defines terms of algebra) with associated "deduction/transformation" rules for reasoning about algebraic specifications. The International Telecommunication Union (ITU) has published a recommendation for the formal semantics of basic Message Sequence Charts (MSCs) based on the Algebra of Communicating Processes (ACP) [2][6]. This work is a continuation of preliminary research first established by Mauw et. Reniers [13]. In more recent work, Rui and Butler also suggest a process algebraic semantics for use case models, with the overall goal of formalizing use case refactoring [22][25]. In their approach, scenarios are represented as basic MSCs–as suggested by [21]. In Rui's proposal, he assigns meaning to a particular use case scenario (episode) by partially adapting the ITU MSC semantics. In addition, semantics are defined for related scenarios of the same use case as well as for related use cases. The following use case relations are formally defined: includes, extends, generalization, proceeds, similar, and equivalence.

Formalisms suitable for the definition of *non-interleaving* semantics are based on partial orders. For example, Zheng et. al. propose a non-interleaving semantics for timed MSC 2000 [7] based on timed labeled partial order sets (lposets) [26]. Partial order semantics for (regular, un-timed) MSCs have been proposed by Alur [1] and Katoen and Lambert [9]. Alur et. al. propose a semantics for a subset of MSCs which only allow message events as possible MSC events types. In contrast, the semantics of Katoen and Lambert is more complete. They map MSCs to a set of partial order multi-sets (pomsets). A pomset is a so-called isomorphic class of a corresponding lposet. A pomset contains all objects that can be derived by a bijective projection from a base lposet. Approaches based on pomsets are very similar to approaches based on lposets.

Mizouni et. al. propose use case graphs as an intermediate notation for use cases [15]. Use case graphs are directed, potentially cyclic graphs whose edges represent use case actions and nodes represent system states. This allows for a natural representation of the order in which actions are to be performed. In order to integrate several use cases into a single specification, Mizouni et. al. describe an algorithm for transforming a set of (related) use case graphs (each representing one use case) into an extended finite state machine (EFSM). The

merging of the graphs is done on the basis of common states within the use case specifications.

The semantic definition proposed in this paper was originally inspired by the lposet approach proposed by Zheng et. al. In addition, similar to the work of Mizouni et. al., we employ use case graphs as an intermediate notation for use cases. Before we present our approach, we discuss some of the core requirements that need to be addressed by any formalism that is to be used to model both use cases and task models.

4.2 Requirements for a Semantic Framework

In Section 3 we reviewed key characteristics of use cases and task models and discussed their current (and specialized) areas of application. In this section, we will re-consider this information in order to compile a set of requirements that would be particular to any common semantic framework for use cases and task models. Both notations are used to specify scenarios that indicate how the system is used. Technically a scenario consists of a, possibly infinite, sequence of events. Therefore, we require that a semantic model for use cases and task models formally **captures sets of usage scenarios**. It should be possible to mechanically extract valid usage scenarios from formal specifications. Also, given a specification and a scenario, it should be possible to unambiguously decide whether the scenario is valid or not, relative to the given specification.

In task modeling (e.g. CTT), one often distinguishes between different task types. Examples are: "data input", "data output", "editing", "modification", or "submit". In the corresponding semantic model events should be **distinguishable by their types** as well. Based on the typing, the sequencing of events may be further constrained. An example of such a constraint is that an event representing the entry of information ("data input") must precede an event of submitting the very same data ("submit"). Of the formalisms we surveyed in Section 4.1, the approach based on labeled partial order sets formalism also distinguishes between different types of events that can occur during a run of a MSC. (In particular, it is the purpose of the labeling function to assign a type to each MSC event.)

In use case modeling, state conditions often constrain the execution of use case steps. For example the pre-condition attribute of a use case denotes the set of states in which the use case is to be executed. In addition, every use case extension is triggered by a condition that must hold before the steps defined in the extension are executed. In order to be able to evaluate conditions, the semantic model must provide means to **capture the notion of the state** and should be able to map state conditions to the appearance of events.

So far we have bound the requirements for the semantic model to the intrinsic characteristics of use cases and task models. The next requirement, however, is more tightly related to the software development process within which use cases and task models are to be crafted. One view of a software

development process is as a series of "disciplines" during which models are iteratively transformed/refined until an implementation level has been reached. Use case and task modeling are part of such a lifecycle. Therefore, a common semantic model should easily **support refinement**.

This last requirement is directly related to one's choice of concurrency models. In interleaving models, the concept of true concurrency is omitted and concurrent system behavior is said to be equivalent to the non-deterministic choice of all possible (interleaved) sequential executions. As it turns out, interleaving approaches typically do not support arbitrary refinement of events (or actions) into sub-events (or sub-actions). The main reason is that, in an interleaving model, *"exactly what is interleaved depends on which events of a process one takes to be atomic"* [19]. Therefore, if a formerly atomic action is further refined, new interleavings among the sub-actions are introduced, which were not taken into account prior the refinement. Hence most of the equivalence relations (e.g. trace equivalence and bissimulation equivalence) are not preserved under arbitrary refinement [3]. This problem does not occur in non-interleaving concurrency models (also referred to as partial order semantics or true concurrency semantics) because the concept of concurrency is fundamental. System behavior is represented in terms of causally inter-related events based on a partial order relation. Events, that are not causally related, are interpreted as concurrent.

4.3 Semantic Domain Based on Sets of Posets

In this section, we illustrate an approach to semantics in which we demonstrate how CTT task models and use cases can be mapped to sets of partially ordered sets. We start by reiterating the definition of a partially ordered set (poset) and then define some operators over posets. Finally, we will describe semantics functions that will define a mapping from use cases and task models into sets of posets.

4.3.1 Mathematical Preliminaries (and Notation)

Definition 4.1 For our purposes, a partially ordered set (poset) is a tuple $(E, \leq)$, where

E : is a set of events, and

$\leq \subseteq E \times E$: is a partial order relation (reflexive, anti-symmetric, transitive) defined on E. This relation specifies the casual order of events.

In order to be able to compose posets we define the following operations:

Definition 4.2 The binary operations: sequential composition (.) and parallel composition (||) of two posets p and q are defined as next. Note that R^* denotes the reflexive, transitive closure of R.

Let $p = (E_p, \leq_p)$ and $q = (E_q, \leq_q)$ with $E_p \cap E_q = \emptyset$ then:

$p.q = (E_p \cup E_q, (\leq_p \cup \leq_q \cup \{(e_p, e_q) |\ e_p \in E_p \text{ and } e_q \in E_q\})^*)$
$p||q = (E_p \cup E_q, \leq_p \cup \leq_q)$

In our approach we define semantics for use cases and task models using the following operations over sets of posets.

Definition 4.3 For two sets of posets P and Q, sequential composition (.), parallel composition (||), and alternative composition (#) are defined as follows:

$P.Q = \{p_i.q_j \mid p_i \in P \text{ and } q_j \in Q\}$
$P||Q = \{p_i||q_j \mid p_i \in P \text{ and } q_j \in Q\}$
$P\#Q = P \cup Q$

Also fundamental to our model is the notion of a trace. Next we define the set of traces for a given poset, and for a given set of posets.

Definition 4.4 A *trace* t of a poset $p = (E, \leq)$ is defined as a (possibly infinite) sequence of events from E such that

$\forall(i, j \text{ in the index set of } t).\ i < j \Longrightarrow \neg(t(j) \leq t(i))$ and
$\bigcup t(i) = E$
where $t(i)$ denotes the i^{th} event of the trace.

Definition 4.5 The set of all traces of a poset p is defined as $tr(p) = \{t \mid t$ is a trace of $p\}$.

Definition 4.6 The set of all traces of a set of posets P is defined as:

$$Tr(P) = \bigcup_{p_i \in P} tr(p_i)$$

Using the set of all traces as a basis, we can define refinement among two sets of posets through trace inclusion.

Definition 4.7 A set of posets Q is a refinement of a set of posets P if, and only if:

$Tr(Q) \subseteq Tr(P)$

4.3.2 Mapping CTT Task Models to Sets of Posets

We now briefly outline how CTT task models can be mapped to sets of posets. The mapping process consists of two steps: (1) conversion of a CTT task tree into a task expression; (2) application of a mapping function that relates the task expression to a corresponding set of posets.

In order to derive a task expression from the task model we first create a corresponding expression tree. In general, an expression tree is a tree whose leaves are operands and whose inner nodes are operators. In this case, the

operands of the expression tree are actions (tasks at the leaf-level) and the operators are the temporal relations defined in CTT. In CTT, all temporal relations are defined as either binary operators (e.g. *enabling*, *disabling*) or unary operators (i.e. *iteration*, *option*). Hence in the expression tree all inner nodes have between one and two children. Since the conversion of trees to expressions is fairly conventional, it will not be described any further.

The next step consists of mapping that task expression into a corresponding set of posets. Action tasks correspond to the elements of the poset. Composite tasks are represented by sets of posets, which have been composed using the composition operators, defined in Section 4.3.1. Our (compositional) semantic function is defined in the common denotational style.

Definition 4.8 The semantic function $\mathcal{M}$ is inductively defined over the possible terms within CTT task expressions, with the following interpretations:

$\mathcal{M}[\![t]\!] = \{(\{t\}, \{(t,t)\})\}$ *//atomic action*
$\mathcal{M}[\![t_1 >> t_2]\!] = \mathcal{M}[\![t_1]\!] \,.\, \mathcal{M}[\![t_2]\!]$ *//enabling*
$\mathcal{M}[\![t_1 \,||\, t_2]\!] = \mathcal{M}[\![t_1]\!] \,||\, \mathcal{M}[\![t_2]\!]$ *//concurrent execution*
$\mathcal{M}[\![t_1 \,[]\, t_2]\!] = \mathcal{M}[\![t_1]\!] \,\#\, \mathcal{M}[\![t_2]\!]$ *//choice*
$\mathcal{M}[\![t_1|+|t_2]\!] = (\mathcal{M}[\![t_1]\!] \,.\, \mathcal{M}[\![t_2]\!]) \,\#\, (\mathcal{M}[\![t_2]\!] \,.\, \mathcal{M}[\![t_1]\!])$ *//order independency*
$\mathcal{M}[\![t^{opt}]\!] = \mathcal{M}[\![t]\!] \,\#\, (\emptyset, \emptyset)$ *//optional execution*
$\mathcal{M}[\![t^*]\!] = \{(\emptyset,\emptyset), \mathcal{M}[\![t]\!], (\mathcal{M}[\![t]\!].\mathcal{M}[\![t]\!]), (\mathcal{M}[\![t]\!].\mathcal{M}[\![t]\!].\mathcal{M}[\![t]\!]), ...\}$ *//iteration*

Note that if the CTT task expression contains the temporal operators *Disabling* ($[>$), *Suspend/Resume* ($|>$) or the newly introduced operator *Abort Choice* ($[]_A$) it needs to be transformed into an equivalent task expression which does not involve these operators, prior to the application of the mapping function.

A simple example illustrates how a task model of a "Search" task (illustrated in Figure 4) is mapped into a corresponding set of posets. In order to perform a search, the user first enters the search string. Next the user either directly submits the search parameter or further refines the search criteria. The "Refine" task consists of the sequential execution of the "Select Category" task and the "Select Sub-Category" task, and may be interrupted (and disabled) at any time by executing the "Submit" task. We employed the binary disabling ($[>$) operator to specify the desired behavior. The meaning of the operator is defined as follows: both tasks specified by its operands are enabled concurrently. As soon as the first (sub) task specified by the second operand (in this case, the "Submit" task) is executed, the task specified by the first operand (in this case the "Refine" task) becomes disabled.

From the task model we can derive the following task expression:

$$t_1 >> ((t_2 >> t_3)[> t_4)$$

Note that the various tasks are represented by using the identifiers (t_1, t_2, t_3, t_4). Next we have to transform the task expression into an equivalent task expres-

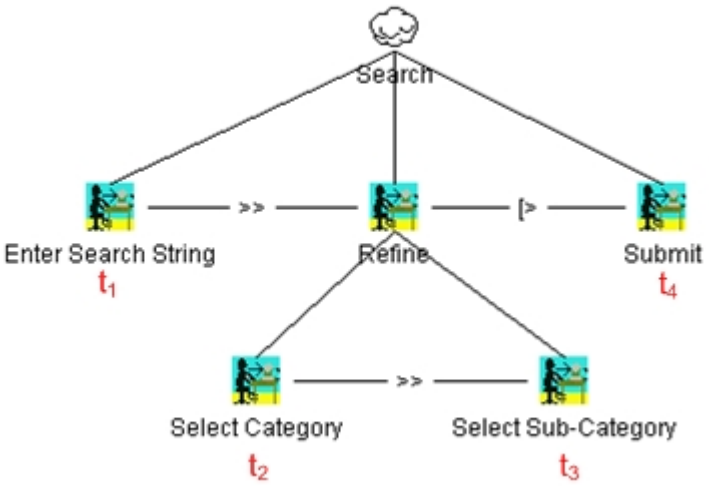

Fig. 4. "Search" Task

sion, which does not make use of the *disabling* operator. This can be done by examining the set of possible scenarios that can be extracted from the specification. In our example we have the choice between the following three scenarios:

(i) The user enters and submits the search string ($t_1 >> t_4$).

(ii) The user enters the search string, selects a category and then submits the search parameter ($t_1 >> t_2 >> t_4$).

(iii) The user enters the search string and selects a category as well as a sub-category before submitting ($t_1 >> t_2 >> t_3 >> t_4$).

Consequently the task expression can be rewritten as follows:

$$(t_1 >> t_4)[](t_1 >> t_2 >> t_4)[](t_1 >> t_2 >> t_3 >> t_4)$$

The task expression is now in elementary form and hence we can apply our semantic function $\mathcal{M}$. According to its recursive definition, the application can be broken down into the following steps:

$$\begin{aligned}
&\mathcal{M}[\![(t_1 >> t_4)[](t_1 >> t_2 >> t_4)[](t_1 >> t_2 >> t_3 >> t_4)]\!] \\
&= \mathcal{M}[\![t_1 >> t_4]\!] \,\#\, \mathcal{M}[\![t_1 >> t_2 >> t_4]\!] \,\#\, \mathcal{M}[\![t_1 >> t_2 >> t_3 >> t_4]\!] \\
&= \mathcal{M}[\![t_1]\!].\mathcal{M}[\![t_4]\!] \,\#\, \mathcal{M}[\![t_1]\!].\mathcal{M}[\![t_2]\!].\mathcal{M}[\![t_4]\!] \,\#\, \mathcal{M}[\![t_1]\!].\mathcal{M}[\![t_2]\!].\mathcal{M}[\![t_3]\!].\mathcal{M}[\![t_4]\!] \\
&= \{(\{t_1, t_4\}, \{(t_1, t_4)\}^*)\} \cup \\
&\quad \{(\{t_1, t_2, t_4\}, \{(t_1, t_2), (t_2, t_4)\}^*)\} \cup \\
&\quad \{(\{t_1, t_2, t_3, t_4\}, \{(t_1, t_2), (t_2, t_3), (t_3, t_4)\}^*)\}
\end{aligned}$$

As a result we obtain a set of three posets, where each poset represents one of the scenarios discussed before.

4.3.3 Transforming Use Cases to Sets of Posets

In this section we discuss how use cases can be transformed into sets of posets. The transformation consists of two parts. First the textual use case is transformed into an intermediate graph form, which we will refer to as the use case graph. Next, based on the use case graph a corresponding set of posets is iteratively constructed.

Definition 4.9 A use case graph is a labeled transition system

$U = (Q, q_0, q_f, T)$ *where*,

Q is a finite set of states
$q_0 \in Q$ is the initial state
$q_f \in Q$ is the final state
$T \subseteq Q \times Q$ is the transition relation.

Similar to the work of Mizouni et. al [15] (discussed in Section 4.1), the transitions of the labeled transition system represent use case steps, whereas the nodes represent states. The composition of the use case graph from the actual use case depends on the flow constructs, which are implicitly or explicitly entailed in the use case. Examples of such flow constructs are: jumps (e.g. *use case resumes at step X*), sequencing information (e.g. the numbering of use case steps), or branches to use case extensions. It is to be noted that if the use case is captured in purely narrative form the derivation of the use case graph will be a manual activity.

Based on the use case graph a set of posets is constructed. The construction can be performed mechanically using the following two steps: **First**, we assign a set of posets to each transition in the use case graph. Typically the set of posets consists of a single poset, which in turn defines a single event representing the execution of the corresponding use case step. **Second**, the use case graph is iteratively transformed into a labeled transition system that only consists of an initial state and a final state. With each iteration one node of the use case graph is eliminated and a new transition is defined between its incoming node(s) and its outgoing node(s). Similar to the first step, a set of posets is assigned to the newly inserted transition. This set of posets is the result of the composition of the sets of posets attached to the incoming transition and the outgoing transition. At this point it no longer represents a single use case step, but a composition of use case steps. Special care must be taken if the eliminated node contains a self loop or if there already exists a transition from the incoming node to the outgoing node.

Once the graph consists of only the initial and the final state, the set of posets associated to the transition between the two states denotes the set of posets representing the original use case graph. We note that the main idea of the presented algorithm stems from the well-known algorithm that transforms a deterministic finite automaton (DFA) into an equivalent regular expression. However, instead of step-wise composition of regular expressions, we compose

sets of posets. We refer the reader to [12] for more details.

In the next and final Section we conclude by summarizing the main ideas of this paper. The proposed semantic domain is related back to our enumerated requirements and an outlook to future avenues is given.

5 Conclusion and Future Work

In this paper we highlighted the need for an integrated methodology for developing use cases and task models. This methodology would rest upon a common semantic framework. In theory, both notations can be used to describe the same information. However, in practice, use cases are mainly employed to document functional requirements whereas task models are used to describe UI requirements and design decisions.

With respect to the definition of the semantic framework we reviewed related work and formalisms. Based on the intrinsic characteristics of use cases and task models, we compiled a list of four core requirements that should be met by any formal framework: (1) capture of sets of usage scenarios, (2) offer a distinction between different event types, (3) capture the notion of the state, and (4) support for event refinement. We then presented our initial approach which maps use cases and task models to partially ordered sets. Thus far, the poset formalism, as presented in this paper, only meets the first and the last requirement. Valid event sequences are specified by relating events using a partial order relation. A scenario is said to be valid if a trace can be extracted from the corresponding set of posets that resembles the event sequence of the scenario. Regarding the requirement of supporting event refinement, we used the poset formalism to specify non-interleaving semantics, which naturally support refinement [3].

As future work, we will be tackling the remaining two requirements. For example, the requirement of supporting different event types can be addressed by defining a labeling function, which maps an event type to each element of the poset. Additionally, rules to further restrict the definition of a valid trace need to be introduced. An example of such rule may be the condition that an event of type *data input* must always be followed by a corresponding event of type *submit*. In the same manner as the labeling function assigns types to events, a similar function can be defined to associate state conditions to occurrence of events. For example: an event execution may be conditional to the satisfaction of a pre-condition; furthermore, the event execution may result in a state satisfying a certain post condition.

Another future avenue deals with the definition of a *satisfiability* relation for use case and task model specifications. A natural definition of *satisfiability* with respect to the used formalism can be formulated through refinement. In this paper we formally defined refinement between two sets of posets based trace inclusion. In this vein, a specification *satisfies* another specification, if the corresponding set of posets of the former specification is a refinement of

the set of posets representing the latter specification. Our ongoing efforts aim at further investigating the definition of a suitable *satisfiability* relation but also focus on tool support for the actual verification process.

References

[1] Alur, R., G. Holzmann, and D. Peled, An Analyzer for Message Sequence Charts, *in Proc. of TACAS'96*, (1996) pp. 35-48.

[2] Baeten, J., and W. Weijland, Process Algebra, *Cambridge Tracts in Theoretical Computer Science* 18. Cambridge University Press., Cambridge (1990).

[3] Castellano, L. and G. de Michelis, Concurrency vs. interleaving: introductive example. *Bulletin of the EATCS*, **31**, February, (1987) pp 12-25.

[4] Cockburn, A., Writing Effective Use Cases, Addison-Wesley, (2001).

[5] Dittmar, A. and P. Forbrig, Higher-Order Task Models, *in Proceedings of Design, Specification and Verification of Interactive Systems 2003.*

[6] ITU-T, Recommendation Z.120- Message Sequence Charts, Geneva, 1996.

[7] ITU-T, Recommendation Z.120- Message Sequence Charts, Geneva, 1999.

[8] Jacobson, I., P. Jonsson, M. Christerson, and G. Overgaard, Object-Oriented Software Engineering - A Use Case Driven Approach, Addison Wesley Longman, Upper Saddle River, N.J., 1992.

[9] Katoen, J. and L. Lambert, Pomsets for Message Sequence Charts, *in Proc. of SAM'98*, 1998.

[10] Kuutti, K., Activity Theory as a Potential Framework for Human-Computer Interaction Research. *In Context and Consciousness: Activity Theory and Human Computer Interaction*, MIT Press, 1996.

[11] Larman, C., Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process, second edition, Prentice-Hall, 2002.

[12] Linz, P., An Introduction to Formal Languages and Automata, Jones and Bartlett Publishers, Second edition, pp. 83-86, 1997.

[13] Mauw, S., and M.A. Reniers, An Algebraic Semantic of Basic Message Sequence Charts, *In the Computer Journal*, Vol. 37, No. 4, 1994.

[14] Merrick P., and P. Barrow, The Rationale for OO Associations in Use Case Modelling, *In Journal of Object Technology*, Vol. 4, No. 9, 2005.

[15] Mizouni, R., A. Salah, R. Dssouli, and B. Parreaux, Integrating Scenarios with Explicit Loops, *in Proceedings of NOTERE 2004*, Essaidia Morocco, June 2004.

[16] Mori, G., F. Paterno and C. Santoro, CTTE: Support for Developing and Analyzing Task Models for Interactive System Design, *in IEEE Transactions on Software Engineering*, August 2002, pp. 797-813, 2002.

[17] Paterno, F., Model-Based Design and Evaluation of Interactive Applications, Springer, 2000.

[18] Paterno, F., Towards a UML for Interactive Systems, *in Proceedings of EHCI 2001*, Toronto, Canada, pp. 7-18, 2001.

[19] Pratt, V.P., Modeling Concurrency with Partial Orders, *International Journal of Parallel Programming*, 15(1), pp. 33-71, February 1986.

[20] Pressman, R., Software Engineering - A Practitioner's Approach, 6th Edition, Mc Graw Hill, 2005.

[21] Regnell, B., M. Andersson, and J. Bergstrand, A Hierarchical Use Case Model with Graphical Representation, *Proceedings of ECBS'96*, IEEE International Symposium and Workshop on Engineering of Computer-Based Systems, March 1996.

[22] Rui, K., A Process Algebraic Semantics for Refactoring Use Case Models, Doctoral Proposal, Concordia University, 2004.

[23] Seffah, A., M. Metzger, and D. Engelberg, Software and Usability Engineering: Prevalent Myths, Obstacles and Integration Avenues. *In Human-Centered Software Engineering -Integrating Usability in the Software Development Lifecycle*, Springer, 2005.

[24] Souchon, N., Q. Limbourg, and J. Vanderdonckt, Task Modelling in Multiple Contexts of Use, *in Proceedings of Design, Specification and Verification of Interactive Systems*, Rostock, Germany, pp. 59-73, 2002.

[25] Xu, J., W. Yu, K. Rui, and G. Butler, Use Case Refactoring: A Tool and a Case Study, *in Proceedings of APSEC 2004*, Busan, Korea, pp. 484-491.

[26] Zheng, T., F. Khendek and B. Parreaux, Refining Timed MSCs, *in SDL 2003: System Design*, Lecture Notes in Computer Science, Volume 2708/2003, pp. 234-250.

Towards a Coordination Model for Interactive Systems

Marco Antonio Barbosa [1,4] Luís Soares Barbosa [2,4]
José Creissac Campos [3,5]

DI-CCTC – Universidade do Minho
Braga, Portugal

Abstract

When modelling complex interactive systems, traditional interactor-based approaches suffer from lack of expressiveness regarding the composition of the different interactors present in the user interface model into a coherent system. In this paper we investigate an alternative approach to the composition of interactors for the specification of complex interactive systems which is based on the coordination paradigm. We layout the fundations for the work and present an illustrative example. Lines for future work are identified.

Key words: Interactors, Coordination models, Configuration.

1 Introduction

Interactive systems can be seen as a special case of the more general class of reactive systems. However, interactive systems have specificities that present new challenges when considering modelling and reasoning about them. One major aspect is the need to consider interaction with the user, and not only between components of the user interface.

The notion of interactor has long been proposed as an approach to structuring and organizing models of interactive systems. Different authors use different flavours of interactors. A common trait being the view of interactors

[1] Email: marco.antonio@di.uminho.pt
[2] Email: lsb@di.uminho.pt
[3] Email: jose.campos@di.uminho.pt
[4] Research carried out in the context of the PURe Project supported by Fct, the Portuguese Foundation for Science and Technology, under contract POSI/ICHS/44304/2002.
[5] Research carried out in the context of the IVY Project supported by Fct, the Portuguese Foundation for Science and Technology, and FEDER, the European regional development fund, under contract POSC/EIA/56646/2004.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

as components capable, not only of communicating between themselves, but also of conveying information to the user(s).

Two main flavours of interactors are York [11] and CNUCE [19] interactors. York interactors are basically objects equipped with a rendering relation that maps their internal state into some presentation medium. More than a concrete specification formalism, they offer a framework for structuring the user interface specifications, whatever formalism is being used. CNUCE interactors (see Fig. 1) can be seen as blackbox components that communicate, with each other, and with the user(s), through input and output ports (for more on CNUCE interactors see section 3).

One main distinction between the two approaches is that with York interactors state can be specified explicitly (cf. MAL interactors [9]), while with CNUCE interactors state is only referred two indirectly through the interactor's ports. Whatever the approach, modelling complex interactive systems entails creating architectures of interconnected interactors. In the case of York interactors, there is no prescription about how that should be accomplished (it will depend on the particular specification approach being used). In the case of CNUCE interactors, specifications are built by connecting the different ports into an adequate architecture by means of synchronous channels. Interactor behaviour is modelled in LOTOS by expressing the relation between input and output ports.

Managing the coordination between the different interactors is typically achieved by the introduction of additional interactors to express the control logic for their communication. This, in turn, adds to the complexity of the models. Ideally we should be able to express the logic of the coordination between the different interactors in an as natural and simple way as possible. In this paper we explore the application of the coordination paradigm to model architectures of interactors. The approach is based on previous work of some of the authors (see, [5,6]).

2 Coordination

The *coordination paradigm* [13,18] offers a promising way to address issues related to the development of complex systems. Since the coordination component is separate from the computational one, the former views the processes comprising the latter as black boxes, whose internal implementation is hidden from the outside world. Instead, the composition of components is defined in terms of their (logical) interfaces which describe their externally observable behavior. By hiding all system computation in the components, a system can be described in terms of the observable behavior of its components and their interactions. As such, component-based software modelling provides a high-level abstract description of a system that allows for a clear separation of concerns between the coordination and the computational aspects.

Closely related to the concept of coordination is that of configuration and

architectural description. They view a system as comprising components and interconnections, and aim at separating structural description of components from component behaviour. Furthermore, they support the formation of complex components as compositions of more elementary components. Finally, they understand changing the state of some system as an activity performed at the level of interconnecting components rather than within the internal purely computational functionality of some component.

Our approach is based on the coordination model Reo [2], a subset of Reo, to be more exact. A more formal treatment of the semantics of our approach is shown in [5,6] where we describe an exogenous coordination model wherein complex coordinators, called "connectors" are compositionally built out of simpler ones. This implies that not only should it be generally possible to produce different systems by composing the same set of components in different ways (creating different configurations), but also that the difference between two systems composed out of the same set of components must arise out of the actual rules that comprise their two different compositions, *i.e.*, their glue code. In such a context we may specify different configurations for a given scenario only by constructing different connectors and patterns of interactions.

Another feature of this work is that our approach takes advantage of the authors' previous work on named *generic process algebra* [3,21]. Such work provides a more general and adaptable approach to the design of complex systems using process algebras. For example, some applications may require similar constructs coexisting with different interaction disciplines (see section 4.2).

Using process algebra to model interactors is not new, and we may refer to the usage of LOTOS in [19] and CSP in [10] (just to name a few). However, our generic approach provides a more flexible way to represent interactors by proposing a clear separation between structural aspects and interaction disciplines.

3 CNUCE Interactors

Paternò views interactors (CNUCE interactors — see Fig. 1) as blackbox entities which communicate through a public interface identified by ports with opposite polarity (*i.e.*, either input or output). Ports are divided into different categories. There are ports to communicate with the users (somewhat equivalent to the rendering relation in York interactors) and ports to communicate with the underlying application functional core. There are also triggers, needed to synchronize the flow of information from input to output ports.

Specifications are built by connecting the ports of different interactors into an adequate architecture by means of synchronous channels. Interactor behaviour is modelled in LOTOS by expressing the relation between input and output ports.

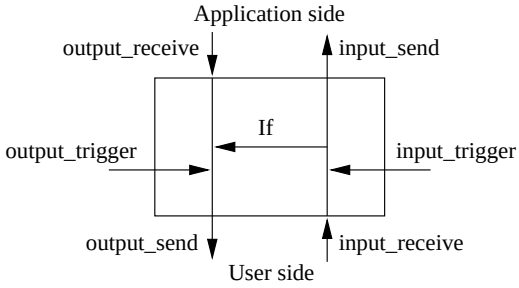

Fig. 1. CNUCE Interactors

An interactor can generate data in two directions: towards the user, and towards the application. This means that interactor behaviour is divided into two distinct parts: the *external* one, which contributes to the definition of the appearance, and the *internal* one, which consists of sending data to other interactors or application processes. Hence, an interactor is defined by a couple of functions: FI is associated with the internal behaviour (the information flow from the user towards the application side); FO is associated with the external behaviour (the information flow from the application towards the user side).

An interactor I, with input_receive ports I_{m_1} to I_{m_n}, input_send ports Inp_1 to Inp_t, output_receive ports I_{c_1} to I_{c_s} and output_send ports Out_1 to Out_z, is defined as (with Φ representing the absence of information)

$I = (FI, FO)$, where:
$FI : (I_m \times Bool \times T) \rightarrow (Inp \cup \Phi) \times If$
with $I_m = I_{m_1} \times \ldots \times I_{m_n}, If = If_1 \times \ldots \times If_k, Inp = Inp_1 \times \ldots \times Inp_t$
$FO : (I_c \times If \times Bool \times T) \rightarrow (Out \cup \Phi)$
with $I_c = I_{c_1} \times \ldots \times I_{c_s}$, and $Out = Out_1 \times \ldots \times Out_z$

As can be seen I_m, If, Inp, I_c, and Out are domains defined by Cartesian products of subdomains. This is mainly because an interactor can receive (and generate) different data types from (to) collections of channels.

In the definition of FO, I_c represents the domain describing the output entities which it receives from the outside. Out is the type which describes the external appearance which can be generated, and If is the data type which the input part of the interactor passes to the output part for echoing. T, in both the FI and FO definitions, is the time that can be considered as a one-dimensional quantity, made up of points, where each point is associated with a value. At moment t, FO is applied to data from I_c and to an element in domain If produced by FI at moment $t-1$.

For interactors without explicit triggers (interactors that generate meaningful results whenever they receive any input), the Boolean in the above can be ignored.

Generally speaking the main difference between the two functions describing one interactor is that the external function receives input data from the

input part of the interactor (in order to echo the current measure value) as well as from the outside. This indicates that the presentation of an interactor is defined by the information it receives from higher levels (levels closer to the application) and the feedback information generated by the users' input.

4 Interactors and coordination

The main aspects of the CNUCE model of interactors can be summarized as follows: interactors are seen as black-box entities communicating through identified ports (input/output), a notion of discrete time and synchronization constraints (involving a notion of trigger) are included in the model, and composition is used in order to construct complex interactive systems from simple components. Such features resemble previous work on coordination models by some of the authors [6,5].

The goal of this paper is to provide an alternative model for expressing the composition of interactors. Central to our approach to the rigorous representation of *interactors* is the notion of *configuration*. This captures the intuition that interactors may be seen as components which cooperate through their specific *interfaces* using *connectors* as the unique communication mechanism, *i.e.*, interactors do not directly interact among themselves. Such idea of connector abstracts the idea of an intermediate *glue code* to handle interaction.

In order to represent a configuration we need a notion of *a)* interactor's *interface*, *b)* what *connectors* are and how they compose, and *c)* how interactors' *interfaces* and *connectors* interact in a *configuration*. These points are tackled in the following sub-sections.

4.1 Interfaces

In exogenous coordination models, like [2] or [5], components are black box entities accessed by purely syntactic interfaces. The role of an interface is restricted to keeping track of port names and, possibly, of admissible types for data items flowing through them [6]. Such a notion of components interface is perfectly extensible with the notion of CNUCE interactors. So, let us define an interface as

Definition 4.1 An interface for a component C is specified by a *port signature*, $sig(C)$ over $\mathbb{D}$, given by a port name and a polarity annotation (either in(put) or out(put)), and a *use pattern*, $use(C)$, given by a process term over port names.

Typically the behaviour of a component's interface can be expressed using transition systems [16], regular-expressions [20] or process algebras [1]. Process algebra, in particular, provides an expressive setting for representing

[6] In the sequel, however, we assume a unique, general data domain, denoted by $\mathbb{D}$, as the type of all data values flowing in an application.

behavioural patterns and establish/verify their properties in a compositional way. Some flexibility, however, is required with respect to the underlying interaction discipline (captured in this work by θ). Actually, different such disciplines have to be used, at the same time, to capture different aspects of component coordination. For example the discipline governing the composition of *software connectors* (to build the overall *glue code*) differs from the one used to capture the interaction between the connectors and the relevant components' interfaces. Meeting this goal entails the need for a *generic* way to *design* process algebras.

The model proposed in this work resorts to the rigorous discipline of process calculi, namely the calculational style presented in [3] to express both component and connectors behaviour.

4.2 Generic Process Algebra

References [3,4] introduced a denotational approach to the *design* of process algebras in which processes are identified with inhabitants of a final coalgebra [15] and their combinators defined by coinductive extension (of 'one-step' behaviour generator functions). The *universality* of such constructions entails both definitional and proof principles on top of which the development of the whole calculus is based.

As we shall see in the following our generic approach to process algebras maintains the basic combinators present in classical processes algebras as CCS, CSP or LOTOS. The fundamental point to be noted is the presence of a more flexible way to represent an *interaction discipline* which parametric on θ. Technically, an *interaction discipline* is modeled as an Abelian positive monoid $\langle Act; \theta, 1\rangle$ with a zero element 0. The intuition is that θ determines the interaction discipline whereas 0 represents the absence of interaction: for all $a \in Act$, $a\theta 0 = 0$. On the other hand, being a positive monoid entails $a\theta a' = 1$ iff $a = a' = 1$. A typical example of an interaction structure captures action co-occurrence as in CSP, in which case θ is defined as $a\theta b = \langle a, b\rangle$, for all $a, b \in Act$. Another example is provided by the action complement match used in CCS [17], *i.e.*, $a\theta\bar{a} = \tau$.

Definition 4.2 Let P be the set of port identifiers and S stand for (the specification of) a component. Its use pattern, denoted by $use(S)$ is given by a process expression over $Act \triangleq \mathcal{P}P$, given by the following grammar:

$$\begin{aligned} P \ ::= \ & \mathbf{0} \mid \alpha.P \mid P + P \mid P \otimes P \mid P \,|||\, P \mid P; P \mid P \mid P \mid \\ & \sigma\, P \mid \mathsf{fix}\ (x \ = \ P) \end{aligned}$$

where α is an element of Act (*i.e.*, a set of port identifiers) and σ is a substitution.

Notice that choosing Act as a *set* of port identifiers allows for the synchronous activation of several ports in a single computational step.

Combinators $\mathbf{0}$, ., $+$, $\otimes$, $|||$, and $|$, represent inactive process, prefix, nondeterministic choice, synchronous product, interleaving, and parallel composition, respectively. *Renaming* is given by term substitution. The $\mathsf{fix}\ (X = P)$ is a fixed point construction, which, as usual, can be abbreviated in an explicit recursive definition. Sequential composition, as in CSP [14], is given by ';' and requires its first argument to be a terminating process.

The semantics of such expressions is fairly standard, but for the parametrization of all forms of parallel composition (*i.e.*, $\otimes$ and $|$) by an interaction discipline as discussed above. The reader is referred to [21] for the full details.

Definition 4.3 The joint behaviour of a collection $\{S_i |\ i \in n\}$ of components is given by

$$use(S_1)\ |\ \ldots\ |\ use(S_n)$$

where the interaction discipline is fixed by $\theta = \cup$, *i.e.*, the synchronisation of actions in α and β corresponds to the simultaneous realization of all of them.

This joint behaviour is computed by the application of Milner's *expansion law* [7] , while obeying the interaction discipline given by θ. The following example illustrates this construction.

Example 4.4 Consider a component C_1 with two ports a and b whose use pattern is restricted to the activation of either a or b, forbidding their simultaneous occurrence. The expected behaviour is captured by

$$use(C_1)\ =\ \mathsf{fix}\ (x\ =\ a.x\ +\ b.x)$$

Now consider another component, C_2, with ports c and d whose behaviour is given by the co-occurrence of actions in both ports. Therefore,

$$use(C_2)\ =\ \mathsf{fix}\ (x'\ =\ cd.x'), \quad \text{where,}\ \ cd\ \stackrel{\text{abv}}{=}\ \{c, d\}$$

According to definition 4.3, the joint behaviour of C_1 and C_2 is

$$use(C_1)\ |\ use(C_2)\ =\ \mathsf{fix}\ (x\ =\ acd.x\ +\ bcd.x\ +\ a.x\ +\ b.x\ +\ cd.x)$$

As a final example, consider still another component C_3, with ports e and f activated in strict order, *e.g.*, first input e and then output f

$$use(C_3)\ =\ \mathsf{fix}\ (y\ =\ e.f.y)$$

Clearly, expansion leads to

$$use(S_2)\ |\ use(S_3)$$
$$=\ \mathsf{fix}\ (x\ =\ cd.x\ +\ e.f.x\ +\ cde.f.x\ +\ cde.cdf.x\ +\ e.cdf + \cdots + cdf.x)$$

[7] This law, which states that a process is always equivalent to the non deterministic choice of its derivatives, is a fundamental result in interleaving models for concurrency.

4.3 Connectors

Our approach resorts to connectors as the only inter-component communication mechanism. This allows a clean, flexible, and expressive model for construction of the glue code for component composition which also supports exogenous coordination.

Connectors are *glueing devices* between services which ensure the flow of data and the meet of synchronization constraints. Their specification builds on top of our previous work on component interconnection [5], extended with an explicit annotation of activation, or *use*, patterns for their *ports*.

Ports are *interface points* through which messages flow. Each port has an *interaction polarity* (either *input* or *output*). Another particular characteristic is the capability to construct complex connectors out of simpler ones using a set of *combinators*.

Let $\mathbb{C}$ be a connector with m input and n output ports. Assume, again, $\mathbb{D}$ as a generic type of data values and $\mathbb{P}$ as a set of (unique) *port identifiers* Formally, the behaviour of a connector may be given by

Definition 4.5 The specification of a connector $\mathbb{C}$ is given by a relation $\mathsf{data}.[\![\mathbb{C}]\!] : \mathbb{D}^m \longrightarrow\!\!\!\!\!\rightarrow \mathbb{D}^n$ which records the flow of data, and a process expression $\mathsf{port}.[\![\mathbb{C}]\!]$ which gives the pattern of port activation.

The model provides a set of basic connectors and combinators which allow us to construct more elaborated connectors and define more complex patterns of coordination and interaction. In the following let us consider some of these basic connectors. For more connectors and a more formal treatment of them we refer to [5,6].

4.3.1 Synchronous channel.

The *synchronous channel* has two ports of opposite polarity. This connector forces input and output to become mutually blocking, in the sense that any of them must wait for the other to be completed.

$$\mathsf{data}.[\![\, \bullet \longmapsto\!\!\!\!\!\rightarrow \bullet \,]\!] \;=\; \mathsf{Id}_{\mathbb{D}} \quad \text{and} \quad \mathsf{port}.[\![\, \bullet \longmapsto\!\!\!\!\!\rightarrow \bullet \,]\!] \;=\; \mathsf{fix}\,(x \;=\; ab.x)$$

Its semantics is simply the identity relation on data domain $\mathbb{D}$ and its behaviour is captured by the simultaneous activation of its two ports.

4.3.2 Drain.

A drain has two input, but no output, ports. Therefore, it loses any data item crossing its boundaries. A drain is *synchronous* if both write operations are requested to succeed at the same time (which implies that each write attempt remains pending until another write occurs in the other end-point). It is *asynchronous* if, on the other hand, write operations in the two ports do

not coincide. The formal definitions are, respectively,

$$\mathsf{data}.[\![\bullet \vdash\!\!\blacktriangledown\!\!\dashv \bullet]\!] = \mathbb{D} \times \mathbb{D} \quad \text{and} \quad \mathsf{port}.[\![\bullet \vdash\!\!\blacktriangledown\!\!\dashv \bullet]\!] = \mathsf{fix}\ (x = ab.x)$$

and,

$$\mathsf{data}.[\![\bullet \vdash\!\!\triangledown\!\!\dashv \bullet]\!] = \mathbb{D} \times \mathbb{D} \quad \text{and} \quad \mathsf{port}.[\![\bullet \vdash\!\!\triangledown\!\!\dashv \bullet]\!] = \mathsf{fix}\ (x = a.x + b.x)$$

4.3.3 $Fifo_1$.
This is a channel with a buffer of a single position.

$$\mathsf{data}.[\![\bullet \longmapsto \bullet]\!] = \mathsf{Id}_{\mathbb{D}} \quad \text{and} \quad \mathsf{port}.[\![\bullet \longmapsto \bullet]\!] = \mathsf{fix}\ (x = a.b.x)$$

4.4 Combinators

Connectors can be combined to build more complex *glueing code*. The following are the required combinators.

4.4.1 Aggregation.
This combinator places its arguments side-by-side, with no direct interaction between them.

$$\mathsf{port}.[\![\mathbb{C}_1 \boxtimes \mathbb{C}_2]\!] = \mathsf{port}.[\![\mathbb{C}_1]\!] \mid \mathsf{port}.[\![\mathbb{C}_2]\!], \text{ with } \theta = \cup \tag{1}$$

4.4.2 Hook.
This combinator encodes a *feedback* mechanism, drawing a direct connection between an output and an input port. Formally, $\mathsf{port}.[\![\mathbb{C}\ \curvearrowleft_i^j]\!]$ is obtained from $\mathsf{port}.[\![\mathbb{C}]\!]$, by deleting references to ports i and j. To be well-formed it is required that i and j appear in different factors of some form of parallel composition ($|||$, $\otimes$, or $|$).

4.4.3 Join.
Its effect is to plug ports with same polarity. The aggregation of *output ports* is done by a *right join* ($\mathbb{C}\ {}^{i}_{j} > z$), where $\mathbb{C}$ is a connector, and i and j are ports and z is a fresh name used to identify the new port. Port z receives asynchronously messages sent by either i or j. When messages are sent at the same time the combinator chooses one of them in a nondeterministic way. On the other hand, aggregation of *input ports* resorts to a *left join* ($z <^{i}_{j} \mathbb{C}$). This behaves like a *broadcaster* sending synchronously messages from z to both i and j. Formally, at a behavioural level, both operators effect is that of a renaming operation

$$\mathsf{port}.[\![(\mathbb{C}\ {}^{i}_{j} > n)]\!] = \mathsf{port}.[\![(n <^{i}_{j} \mathbb{C})]\!] = \{n \leftarrow i, n \leftarrow j\}\mathsf{port}.[\![\mathbb{C}]\!]$$

4.5 Configurations

Finally, let us complete the whole picture providing a notion of configuration. A *configuration* is simply a collection of *components*, characterized by their *interfaces*, interconnected through a *connector* network built from elementary connectors using the combinators mentioned above. Formally,

Definition 4.6 A configuration involving a collection $C = \{C_i |\ i \in n\}$ of components is a tuple

$$\langle U, \mathbb{C}, \sigma \rangle \tag{2}$$

where $U = use(C_1) \mid use(C_2) \mid \cdots \mid use(C_n)$ is the (joint) use pattern for C, $\mathbb{C}$ is a connector and σ a mapping of ports in C to ports in $\mathbb{C}$.

The relevant point concerning configurations is the semantics of the interaction between the *connector's port behaviour* and the *joint use patterns* of the involved components. This is captured by a synchronous product $\otimes$ for a quite peculiar θ, which is expected to capture the following requirements:

- Interaction is achieved by the simultaneous activation of identically named ports.
- There is no interaction if the connector intends to activate ports which are not linked to the ones offered by the interactors' side. For example if a port a of an interactor S is connected to the input end of a synchronous channel whose output end is disconnected, no information can flow and port a will never be activated.
- The dual situation is allowed, *i.e.*, if the interactors' side offers activation of all ports plugged to the ones offered by the connectors' side, their intersection is the resulting interaction.
- Moreover, and finally, activation of unplugged interactors' ports is always possible.

Formally, this is captured in the following definition.

Definition 4.7 The behaviour $bh(\Gamma)$ of a configuration $\Gamma = \langle U, \mathbb{C}, \sigma \rangle$ is given by

$$bh(\Gamma) \ = \ \sigma\, U \ \otimes \ \mathsf{port}.[\![\mathbb{C}]\!] \tag{3}$$

where θ underlying the $\otimes$ connective is given by

$$c\ \theta\ c' \ = \ \begin{cases} c \cap (c' \cup \mathsf{free}) \Leftarrow \ c' \subseteq c \\ \emptyset \qquad\qquad\qquad\quad \Leftarrow \ otherwise \end{cases} \tag{4}$$

and free denotes the set of unplugged ports in U, *i.e.*, not in the domain of mapping σ.

5 An Example

As an example let us consider a variation of an air traffic control system presented in [12]. Our example (see Fig. 2) is centred in a scenario where aircrafts A_2 and A_3 are on their final approach to the runway, aircraft A_1 is

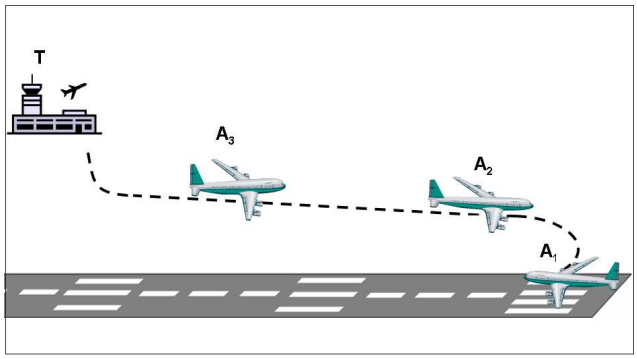

Fig. 2. Air Traffic Control Configuration

on the runway waiting the response for its 'accepted' to take off requirement, and the tower T is responsible for air traffic control. Aircraft A_2 and A_3 are on their "downwind leg" and are to be turned onto a heading towards the runway. Before A_2 can be turned it must reduce speed. This means that A_3 must reduce speed also to avoid loss of separation with A_2. Of course, A_2 will be allowed to land just after A_1 has taken off.

At this stage we are mainly interested in investigating how to combine interactors in different ways for different scenarios. Investigating the appropriateness of each configuration would be the next step in the design process.

First we express the expected behaviour of the interactors involved in this configuration.

interactor: A_i
ports: $\mathsf{slow}'_{\mathsf{i}}, \mathsf{turn}'_{\mathsf{i}}, \mathsf{accept}'_{\mathsf{i}}$
external behaviour:
$use(A_i) \;=\; \mathsf{fix}\,(x \;=\; \mathsf{slow}'_{\mathsf{i}}.x \;+\; \mathsf{turn}'_{\mathsf{i}}.x \;+\; \mathsf{accept}'_{\mathsf{i}}.x)$, where $0 < i \leq 3$.

Such a specification represents the three aircrafts involved in the scenario. Each aircraft has three input ports (distinguished by the symbol: $'$) available for communication in a non-deterministic manner. The tower is represented by interactor T.

interactor: T
ports: $\mathsf{slow}_{\mathsf{i}}, \mathsf{turn}_{\mathsf{i}}, \mathsf{accept}_{\mathsf{i}}$
external behaviour:
$use(T) \;=\; \mathsf{fix}\,(x \;=\; \mathsf{slow}_{\mathsf{i}}.x \;+\; \mathsf{turn}_{\mathsf{i}}.x \;+\; \mathsf{accept}_{\mathsf{i}}.x)$, where $0 < i \leq 3$.

Once the interactors defined, the following step is to define how they will

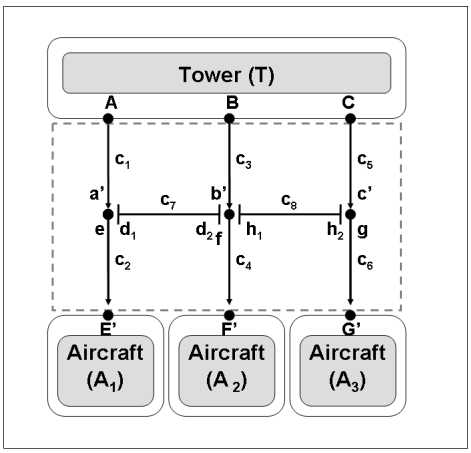

Fig. 3. Air Traffic Control Configuration

cooperate, *i.e.*, we need to represent how the whole system will behave. Such is done by creating an architecture of interactors and connectors.

The scenario captured by Fig. 2 represents a critical situation where the aircrafts must respond to actions appropriately or the safety will be dangerously compromised. So, let us consider a situation where T sends a message $\mathsf{accepted}_1$ to A_1, in order for A_1 to take off, the message slow_2 to A_2, in for A_2 to slow before turning to the runway, and the message slow_3 to A_3 in order for A_3 decrease speed maintaining a safety distance to A_2. In order to ensure that the response to these actions will happens synchronously we may consider a special connector, called *synchronization barrier* SB which enforces that all messages are delivered to their destinations in a synchronous way.

Such a connector (see Fig. 3) is an aggregation among six synchronous channels ($c_1, \ldots, c_6$) and two synchronous drains (c_7 and c_8) which are composed using hook and join combinators. This connector is computed starting from the behaviours of the elementary connectors, *e.g.*, $\mathsf{port}.[\![c_1]\!] = \mathsf{fix}\ (x = aa'.x)$, till the behaviour of the whole connector is calculated: $\mathsf{port}.[\![SB]\!] = \mathsf{fix}\ (x = abce'f'g'.x)$

The resulting behaviour of this connector means that the six ports must be activated synchronously. It should be noted that, since we are not considering timing issues at this stage, this synchronicity does not meant that the ports are activated concurrently. In the current context, what we are stating is that if one port is activated, then all the other must be activated, before the connector can engage in a new interaction.

The configuration of such a scenario is given by

$$\begin{aligned} C_{f_1} &= \langle USC, \mathbb{C}, \sigma_{SC} \rangle, \text{ where} \\ USC &= use(T) \mid use(A_1) \mid use(A_2) \mid use(A_2) \\ \mathbb{C} &= SB \\ \sigma_{cf_1} &= \{a \leftarrow A, b \leftarrow B, c \leftarrow C, e' \leftarrow E', f' \leftarrow F', g' \leftarrow G'\} \end{aligned}$$

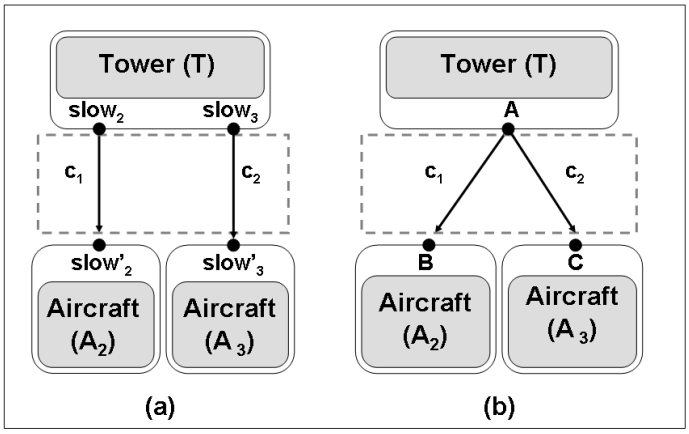

Fig. 4. Parallel and broadcaster connectors

For a cleaner notion let us consider $A = \mathsf{accept}_1$, $B = \mathsf{slow}_2$, $C = \mathsf{slow}_3$, $E' = \mathsf{accept}'_1$, $F' = \mathsf{slow}'_2$, and $G' = \mathsf{slow}'_3$.

The result of the $\otimes$ composition of USC and SB is the behaviour of configuration C_{f_1}. There is no need, however, to compute the complete expansion of the parallel composition in USC expression, which is

$$\begin{aligned}\mathsf{fix}\,(x \;=\; & a.x + \cdots + e'.x + f'.x + g'.x + \\ & ae'.x + \cdots + be'.x + \cdots + ce'.x + \cdots + abce'.x + \cdots + \\ & ae'f'.x + \cdots + be'f'.x + \cdots + ce'f'.x + \cdots + abce'f'.x + \cdots + \\ & ae'f'g'.x + \cdots + be'f'g'.x + \cdots + ce'f'g'.x + \cdots + \underline{abce'f'g'}.x + \cdots + \\ & e'f'.x + e'g'.x + f'g'.x + e'f'g'.x)\end{aligned}$$

because, according to interaction discipline (4), the only successful case of composition with $\mathsf{port}.[\![SB]\!]$ corresponds to the underlined alternative in the expression above. Clearly, the θ-composition of $abce'f'g'$ with $abce'f'g'$ (from the connector side) is $abce'f'g'$, while for all other cases it results in the empty set $\emptyset$. Therefore, and finally,

$$bh(C_{f_1}) \;=\; \mathsf{fix}\;(x \;=\; abce'f'g'.x) \tag{5}$$

Consider now the configuration in Fig. 4 (a) where T sends messages to A_2 and A_3 synchronously. We may specify a situation where T can only send a message for A_2 to slow down if it also sends a slow down message to A_3. This is captured by

interactor: T
ports: $\mathsf{slow}_2, \mathsf{slow}_3$
external behaviour: $use(T) \;=\; \mathsf{fix}\;(x \;=\; \mathsf{slow}_3.x \;+(\; \mathsf{slow}_2.\mathsf{slow}_3.x))$

Consider now a situation where T needs to send synchronously a message to both A_2 and A_3. A solution for this situation is pictured in Fig. 4 (b).

As a final remark is important to note that this work reports on the main ideas of this approach only. The full specification of the calculi involved in the development of the examples was not demonstrated in this paper. We refer to [7] for a complete view of this approach applied to another kind of application.

6 Conclusions and Future Work

When modelling complex interactive systems, traditional interactor-based approaches suffer from lack of expressiveness regarding the composition of the different interactors present in the user interface model into a coherent system. In this paper we have started exploring the application of a coordination based approach to express the *interconnection glue* between interactors.

By using the notion of a black-box component, defined only by its interface to the outside, this approach can be closely related to that of CNUCE interactors. The definition of an interactor in CNUCE as $I = (FI, FO)$, is a relation among input port to output ports, *i.e.*, $I : FI \rightarrow FO$ quite similar to our approach where an interactor is given by a relation $\mathsf{data}.[\![\mathbb{C}]\!] : \mathbb{D}^m \rightarrow \mathbb{D}^n$. Although the definition does not clearly separate ports according to categories (as is done in CNUCE interactors), this can be easily accomplished by using syntactic annotation, as hinted at in the example. Nevertheless, the rendering of information to users is one characteristic of interactors that has not been fully explored in this paper. With the formal underpinning now in place, we intend to explore this as the next step in this work.

Our approach promotes a clear separation of concerns between the specification of the individual components of the model (the interactors), and the specification of how they are organized into an architecture, and how they interact with each other. This separation of concerns was not as clear neither in CNUCE interactors, or in the York based MAL interactors [9], but it is fundamental to enable the modelling of complex systems in a more clear and concise manner.

In the CNUCE model we have an explicit representation of time and a trigger to regulate the synchronisation constraints. Although this aspect has not been addressed here, in [6] a preliminary version of our approach was presented where time was also explicitly defined by a time stamp $\mathbb{T}$ representing, in fact, not real time but a way to represent an order of data occurrence. In the current model the notion of 'time' is implicitly represented by the sequence in which the ports are activated, *i.e.*, the sequence in which the data flows thought the ports. For instance, if we have a synchronous channel, both ports are activated 'at same time' *i.e.*, ports are activated in an atomic way without being interleaved by another operation while both operations have not been well succeed. If we model an asynchronous channel between the activations of both ports, then other port activations might succeed in between the two. Another point to note is that with a parametric interaction discipline and the

rigour provided by the connectors, there is no need for triggers in our model.

The use of connectors allows for more flexibility in the design of complex systems. This constitutes an advantage not only compared with CNUCE models but with any model which uses simple channels as communication medium. The capability to define a filter in the connectors without the need to change the definition of an interactor can be a desirable feature. Or, as shown in our example, we may construct different configurations from a scenario. The theme of defining different architectures to achieve different interaction effects in the user interface is also one that deserves further research.

As a final note, it should be point out that, when modelling complex interactive systems, the need arises to express dynamic aspects of the user interface, such as user interface components being created and destroyed, or the interconnections between components being changed in runtime. This is a complex area which we have not addressed here. A very preliminary work in this direction was presented in [8]. In that work the basic connectors are enriched with a special connector called orchestrator which is responsible to handle the mobility and the dynamism of the system. We plan to explore this aspect further, as it is one of the main drives for our research in identifying alternative modelling notations for interactive systems.

References

[1] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM TOSEM*, 6(3):213–249, 1997.

[2] F. Arbab. Reo: a channel–based coordination model for component composition. *Mathematical Structures in Comp. Sci.*, 14(3):329–366, 2004.

[3] L. S. Barbosa. Process calculi *à la* Bird-Meertens. In M. L. Andrea Corradini and U. Montanari, editors, *CMCS'01*, volume 44.4, pages 47–66, Genova, April 2001. Elect. Notes in Theor. Comp. Sci., Elsevier.

[4] L. S. Barbosa and J. N. Oliveira. Coinductive interpreters for process calculi. In *Proc. of FLOPS'02*, pages 183–197, Aizu, Japan, September 2002. Springer Lect. Notes Comp. Sci. (2441).

[5] M. Barbosa and L. Barbosa. Specifying software connectors. In K. Araki and Z. Liu, editors, *Proc. First International Colloquim on Theoretical Aspects of Computing (ICTAC'04), Guiyang, China*, pages 53–68. Springer Lect. Notes Comp. Sci. (3407), 2004.

[6] M. A. Barbosa and L. S. Barbosa. A relational model for component interconnection. *Journal of Universal Computer Science*, 10(7):808–823, 2004.

[7] M. A. Barbosa and L. S. Barbosa. Configurations of web services. In *Proceedings of the 5th International Workshop on the Foundations of Coordination Languages and Software Architectures (FOCLASA'06)*, Electr. Notes Theor. Comput. Sci., Bonn, Germany, August 2006. Elsevier. To appear.

[8] M. A. Barbosa and L. S. Barbosa. An orchestrator for dynamic interconnection of software components. In *Proc. 2nd International Workshop on Methods and Tools for Coordinating Concurrent, Distributed and Mobile Systems (MTCoord'06)*, Electr. Notes Theor. Comput. Sci., Bologna, Italy, June 2006. Elsevier. To appear.

[9] J. C. Campos and M. D. Harrison. Model checking interactor specifications. *Automated Software Engineering*, 8(3/4):275–310, August 2001. ISSN: 0928-8910.

[10] D. A. Duce, R. van Liere, and P. J. W. ten Hagen. An approach to hierarchical input devices. *Comput. Graph. Forum*, 9(1):15–26, 1990.

[11] D. J. Duke and M. D. Harrison. Abstract interaction objects. *Computer Graphics Forum*, 12(3):25–36, 1993.

[12] B. Fields, P. Wright, and M. Harrison. Time, tasks and errors. *SIGCHI Bull.*, 28(2):53–56, 1996.

[13] D. Gelernter and N. Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):97–107, 1992.

[14] C. A. R. Hoare. *Communicating Sequential Processes*. Series in Computer Science. Prentice-Hall International, 1985.

[15] B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:222–159, 1997.

[16] J. Magee, J. Kramer, and D. Giannakopoulou. Behaviour analysis of software architectures. In *WICSA1: Proc. of the TC2 First Working IFIP Conf. on Software Architecture (WICSA1)*, pages 35–50. Kluwer, B.V., 1999.

[17] R. Milner. *Communicating and Mobile Processes: the π-Calculus*. Cambridge University Press, 1999.

[18] G. Papadopoulos and F. Arbab. Coordination models and languages. In *Advances in Computers — The Engineering of Large Systems*, volume 46, pages 329–400. 1998.

[19] F. D. Paternò. *A Method for Formal Specification and Verification of Interactive Systems*. PhD thesis, Department of Computer Science, University of York, 1995. Available as Technical Report YCST 96/03.

[20] F. Plasil and S. Visnovsky. Behavior protocols for software components. *IEEE Trans. Softw. Eng.*, 28(11):1056–1076, 2002.

[21] P. R. Ribeiro, M. A. Barbosa, and L. S. Barbosa. Generic process algebra: A programming challenge. In *Proc. 10th Brazilian Symposium on Programming Languages*, Itatiaia, Brasil, 2006.

Some Issues in Modeling the Performance of Soft Keyboards with Scanning

Samit Bhattacharya [1,2]

Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur
Kharagpur, West Bengal, India

Anupam Basu

Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur
Kharagpur, West Bengal, India

Debasis Samanta

School of Information Technology
Indian Institute of Technology Kharagpur
Kharagpur, West Bengal, India

Souvik Bhattacherjee, Animesh Srivastava

Department of Computer Science and Informatics
Haldia Institute of Technology
Haldia, West Bengal, India

Abstract

Persons suffering from severe speech and motion impairments like quadriplegics or cerebral palsy depend on augmentative communication systems for their daily communication. One group of widely used augmentative communication system consists of soft keyboards. Soft keyboards are on-screen representation of physical keyboards. A user can compose text with a soft keyboard by selecting character keys from the soft keyboard interface. Motion impaired users access soft keyboard keys using access switches supported by scanning, which is successive highlighting of on-screen items. To select an item on a scan enabled interface, the user has to activate an access switch when the item is highlighted. While designing soft keyboards for motion impaired users, it is necessary to make appropriate design choices from multiple layouts and scanning mechanisms to improve user performance. However,

This paper has been presented at the
1st International Workshop on Formal Methods for Interactive Systems
URL: `fmis.iist.unu.edu`

each such design decision should be properly evaluated with the user which is a difficult if not impossible task due to the disabilities of the user. One way to alleviate this problem is to use an automatic performance evaluation model. In this paper, we propose a model to automatically evaluate performance of soft keyboards with scanning. The proposed model was developed based on an existing model. The issues encountered during the development and application of the proposed model are discussed in this paper.

Key words: Soft Keyboard, Fitts-Digraph model, Scanning, Focus movement time.

1 Introduction

Text composition is perhaps the single most important task done with a computer system. The traditional technique for text composition in a computer system is through a physical keyboard like QWERTY. However, in many cases, text entry using a physical keyboard becomes impossible. For example, in small portable devices like PDAs, the small size of the device does not allow incorporating a physical keyboard for entering texts. Alternate strategies have been devised for such situations. One of these is the use of *soft (ware)* keyboards [12][24]. A soft keyboard refers to a software systems having an on-screen representation of a physical keyboard. The soft keyboards can be of two types, namely *ambiguous* and *unambiguous*. In ambiguous keyboards, each soft keyboard key represents multiple characters while in unambiguous keyboards, each key corresponds to a single character. In this paper, we consider only unambiguous keyboards. The character keys of the soft keyboard can be operated with touch, stylus or a conventional mouse.

Apart from being useful for mobile devices, another important application of the soft keyboards is as *augmentative communication aids* for persons with severe speech and motor impairments like quadriplegics, cerebral palsy etc. [1][9]. Being devoid of their normal means of communication (viz. speech, writing and gesture) due to their physical disabilities, these people depend solely on such augmentative communication aids to express themselves [2][3]. With a soft keyboard, the disabled users can compose text letter-by-letter by selecting character keys from the interface. However, their physical disabilities prevent them from using touch or stylus or a conventional mouse to select the character keys of the soft keyboard. Alternate mechanisms have been devised to make the soft keyboard interfaces accessible to the disabled user.

The most common among these alternate access mechanisms constitute of the *access switches*. Access switches (see for example, www.abilityhub.com) are specially designed devices that require lesser motor control to operate. Any

[1] Corresponding author
[2] Email:samit@cse.iitkgp.ernet.in

active body part of the user including hand, foot, mouth or head can be used to operate such switches. Accordingly, there are different types of switches with wide variation in shapes and sizes. The switch based access mechanism is supported by *scanning* [19][22]. Scanning is the successive highlighting of on-screen elements [6]. When the highlighter reaches the desired element, the user activates an access switch to select that element. The switch activation depends on the type of access switch. For example, in case of a hand operated press switch, switch activation implies pressing and releasing the switch while for eye operated switches, eye blink (s) are used to activate an access switch.

While designing soft keyboards for the disabled users, the designer is faced with several design alternatives. First, s/he has to choose from a large number of alternative layouts. These alternate layouts result from the possible ways in which the character keys of the soft keyboard can be organized on the screen. While this is true for any soft keyboard, the numbers of possible designs choices to the designer of soft keyboards with scanning increases significantly due to the wide variation in the scanning process itself. This is so since the performance of a disabled user communicating through a soft keyboard with scanning depends on both the layout and the scanning process. Thus, to a designer of soft keyboard systems with scanning, each design alternative consists of a layout and a scanning mechanism. Choosing from the large number of design alternatives is very difficult. Apart from relying on the intuition, the only choice left to the designer is to implement prototype systems and evaluate the prototypes with real users before finalizing on a design alternative. However, evaluating prototype systems with real users is not an easy task due to the following reasons.

- It is difficult to find sufficient number of disabled users for evaluation.
- It is difficult to generate sufficiently large usage data for analysis due to the physical disabilities of the user.

These problems can be alleviated if there exists a model to automatically evaluate the performance of soft keyboard interfaces with scanning. With such a model, all the alternate designs can be evaluated automatically (without requiring user involvment), thus aiding the designer in making an appropriate choice.

Several issues are involved in the design of an automatic performance evaluation model of the soft keyboards with scanning. In this paper, those issues are discussed along with the description of an automatic performance evaluation model that we have developed. Section 2 of the paper presents the existing automatic performance evaluation model developed for soft keyboards. This is followed by a discussion in Section 3 on the applicability of the existing model in predicting performance of soft keyboards with scanning. The model we have developed is described in Section 4 of the paper. To evaluate our model, we have also developed a soft keyboard which is described in Section 5. The soft keyboard is developed in Bengali, a language belonging to the Indo-Aryan

language family spoken primarily in Eastern India and Bangladesh. The development of Bengali soft keyboard gave rise to many language related issues in automatic performance evaluation with the model. These are discussed in Section 6. Section 7 presents the performance predicted by the model for the soft keyboard. Section 8 is the concluding section of the paper.

2 Related Work

The soft keyboard performance evaluation model most cited in the literature is the Fitts-Digraph model. The model was first proposed by Soukoreff and MacKenzie [21]. In subsequent years, a number of works were carried out on this model [12][15][24]. The model is comprised of the following three components.

- The Fitts law [4][5][10][11] component to predict the movement (of finger or stylus) time of the user among character keys on the soft keyboard. The Fitts law predicts the time required to perform a motor movement from a source character key k_i to a target character key k_j on the soft keyboard interface using the following equation (Equation 1).

$$MT_{i,j} = A + B \times log_2(D_{i,j}/W_j + 1) \tag{1}$$

  In Equation 1, $MT_{i,j}$ is the movement time from the source to target character key, A and B are constants, $D_{i,j}$ is the Cartesian distance between the two character keys and W_j is the width of the target character key. To measure the Cartesian distance $D_{i,j}$, each character key on the soft keyboard interface is assigned a co-ordinate. For example, if the co-ordinates of k_i is (x_i, y_i) and the co-ordinates of k_j is (x_j, y_j), then $D_{i,j}$ is measured using the following equation (Equation 2).

$$D_{i,j} = \sqrt{(x_j - x_i)^2 + (y_j - y_i)^2} \tag{2}$$

- The Hick-Hymans law [7][8] to predict the time to visually search and locate the desired character key from the group of character keys present on the interface. The Hick-Hymans law predicts the time with the following equation (Equation 3).

$$RT = A' + B' \times log_2(N) \tag{3}$$

  In Equation 3, RT is the time to visually search and locate a character key on the interface, A' and B' are constants and N is the number of character keys present on the interface.

- A language component in the form of a table for the relative frequencies of letter pairs, or *digraphs*, in common English. In the performance model, the probability of occurrence of each digraph is used. The digraph probability is calculated by the following equation (Equation 4).

$$P_{i,j} = f_{i,j} / \sum_{i=1}^{N} \sum_{j=1}^{N} (f_{i,j}) \tag{4}$$

In Equation 4, $f_{i,j}$ is the digraph frequency of character keys k_i and k_j obtained from the table and $P_{i,j}$ is the probability of occurrence of the digraph represented by the character keys k_i and k_j.

In the performance model, the above three components are combined together. The resulting performance model is represented by the following equation (Equation 5).

$$\text{Performance (in CPS or character/second)} = 1/(RT + MT_m) \tag{5}$$

In Equation 5, MT_m represents the mean movement time between any pair of character keys on the soft keyboard and is calculated using the following equation (Equation 6).

$$MT_m = \sum_{i=1}^{N} \sum_{j=1}^{N} (P_{i,j} \times MT_{i,j}) \tag{6}$$

Assuming that average word length in English is five characters (including spaces), the performance of soft keyboard interfaces is often measured in *words per minute* (WPM). The resulting modification needed in Equation 5 to measure performance in WPM is shown in Equation 7.

$$Performance(WPM) = CPS \times 60/5 \tag{7}$$

2.1 Assumptions and Constant Values used for Model Prediction

In the prediction of soft keyboard performance using Fitts-Diagraph model (Equation 5), the users are assumed to be of two categories; *novice* and *expert*. The novice users are characterized by the fact that they require non-zero time to visually locate character key on the interface. Thus, their performance (in CPS) is governed by Equation 5. On the other hand, expert users are assumed to have complete familiarity with the interface layout and hence have a zero visual search time. This leads to a modification of Equation 5 with RT set to zero. Thus, the expert users performance is represented by the following equation (Equation 8) instead of Equation 5.

$$\text{Performance (in CPS or character/second)} = 1/MT_m \tag{8}$$

The constants A present in Fitts law (Equation 1) and A' present in Hick-Hymans law (Equation 3) are both set to zero in calculating performance. Note that a special case has to be made for Equation 1 when $i = j$. This is when the user taps on the same character key successively (e.g., *oo* as in look). In this case, the second term in Equation 1 is zero. In [14][26], 0.127 and 0.135 seconds were used as the value of A in such cases. In [24], the value 0.127 second was used. The influence of this number is small, however, due to the low frequency of such cases. Moreover, the constant B present in Equation 3 was set to 0.2. In their original work, Soukoreff and MacKenzie [21] used the value 1/4.9 for the constant B present in Equation 1. In [24], three values for B (1/4.9, 1/6.0 and 1/8.0) were used to predict expert performance.

3 Applicability of the Model to the Soft Keyboards with Scanning

The Fitts-Digraph model was developed to predict performance of an able-bodied user. The basic assumption behind the model was that the interaction was carried out by the user through substantial motor movement (movement of finger or stylus). Accordingly, a major component of the model is the movement time model represented by the Fitts law. This assumption does not hold in the case of soft keyboards for motion impaired user. For the motion impaired user, the interaction is carried out through alternate access mechanism comprising of scanning and access switches.

3.1 The Scanning Mechanisms

There are a large number of scanning mechanisms developed to make computer interfaces accessible to the motion impaired users. All these scanning mechanisms can broadly be classified into two groups, namely *co-ordinate* scanning and *matrix* scanning [17][23]. These are described below.

3.1.1 Co-ordinate Scanning

In co-ordinate scanning, the computer screen is assumed to represent a two-dimensional co-ordinate system. Any element on it is treated as a point. To get to a specific point, one or both of the axes of the co-ordinate systems are moved automatically as if sweeping the entire screen area. The movement of the axes can be rotational (either of the clockwise or anticlockwise direction) or translational. Once an axis passes through the desired on-screen point, an access switch is activated to stop the axis (axes) movement. Next, another activation of the access switch is performed that starts an automatic cursor movement towards the desired point. Once the cursor reaches the point, the access switch is activated again to stop the cursor movement. Further switch activations are done subsequently to select the point.

3.1.2 Matrix Scanning

In matrix scanning, the screen is assumed to represent a matrix. The items that are present on the screen are individual cells of that matrix. The matrix scanning can have the following five variants.

- Three level matrix scanning (block, row and column levels): In a three level matrix scan technique, the on-screen items are divided into blocks. Each block is further divided into a set of rows and each row is in turn divided into a few columns. In this scheme, the system initially starts a block level scan. During this process, the block that contains the desired item can be selected by the user. Once a block is selected, the system begins a row level scan inside the block. During the row level scanning, the row in which the desired item lies is selected. Then the columns of the selected row are

scanned. When the scanning reaches the desired item, the item is selected.

- Two level matrix scanning (row and column levels): In this type of scanning, the on-screen items are divided into rows and columns only. Here the system initially performs the row level scan. Once the user selects the desired row, the system performs the column level scan. When the scanning reaches the desired item, the item is selected.
- Single level matrix scanning (column level): In the single level matrix scan technique, the region of interest is broken into columns only. Each column represents an on-screen item. These columns are scanned periodically. When the scanning reaches the column that holds the desired item, the item is selected.
- Alternate selection mode: This is a variant of block-row-column scanning, with the block and row concepts replaced by *subset* concepts. A subset of items may not have a spatially coherent structure as that of block or rows.
- Diagonal selection mode: This is another three level scanning. In this case, the major matrix (a block) is split into two triangular matrices based on the main diagonal. In the first stage of the scanning, the two parts of the matrix are periodically highlighted and the user selects the triangle where the target column is located. Then a row scanning is applied for its rows and so on.

Each of these broad categories of scanning can have numerous variations in itself. These variations result from the use of different *scan step* (i.e. the time to move the focus between two successive elements), number of access switches (auto scanning with single switch versus guided scanning with multiple switches) and so on.

4 Incorporating Scanning in the Model

Thus, in a scanning and access switch based interaction, a highlighter (also called *focus*) moves periodically and successively on the on-screen item. In the context of soft keyboard, such on-screen items correspond to the character keys present on the interface. The user waits until the desired character key is highlighted and then activates the access switch to select the item. Use of scanning and access switches implies that the Fitts law component in the Fitts-Digraph model is not applicable to predict performance of motion impaired users. Clearly, in stead of (motor) movement time, the model should incorporate a component that calculates the *focus movement time*, i.e. the time to move the focus (highlighter) from the source to target character key.

4.1 Assumptions about the Soft Keyboard Layout

In this paper, we propose a modification of the Fitts-Digraph model to account for a particular type of scanning, namely *three level matrix scanning*. In the

Event	**Time Taken**
The block b containing k is highlighted again	$T_{COL-BLK}$
Focus moves from b to the block b'	$T_{b-b'}$
User selects b' and the row level scanning in b' starts	$T_{BLK-ROW}$
Focus moves from the first row to r' in the block b'	$T_{1-r'}$
User selects r' and column level scanning in r' starts	$T_{ROW-COL}$
Focus moves from the first column in r' to c'	$T_{1-c'}$
User selects c' once c' is focused	T_{SEL}

Table 1
The events that take place in selecting k' after selecting k

modification, we have replaced the Fitts law component of the Fitts-Digraph model with a focus movement time component. The basic assumption made behind the formulation of the focus movement time is that the character keys of the soft keyboard are organized into a three dimensional matrix comprising of blocks, rows and columns. Each block is assigned an integer according to the order of scan with the block assigned the highest integer getting the focus last before the focus goes back to the first block again. Thus, if there are four blocks then the first block that gets the focus when the system starts will be assigned the integer 1, the next block getting the focus will be assigned the number 2 and so on. Similarly, each row in a block and each column in a row are assigned integer according to the order of scan. Moreover, when a column (key) is selected, the scanning mechanism returns to block scanning (i.e. the blocks are scanned again).

4.2 Calculating Focus Movement Time

With the scheme described in the previous section, each key in the keyboard is represented by a triplet of the form $\langle b, r, c\rangle$ where b is the block number, r is the row number and c is the column number of the key. Now, let there be two different keys k and k' on the soft keyboard represented by the triplets $\langle b, r, c\rangle$ and $\langle b', r', c'\rangle$ respectively. Then once the key k is selected by the user, the events shown in Table 1 take place before the key k' is selected.

With a proper implementation, the quantities $T_{COL-BLK}$ (i.e. the time taken to make a transition from column level scanning to block level scanning),

b > b'	**b ≤ b'**
$\sigma(k') = b' + r' + c'$	$\sigma(k') = b' + r' + c'$
$X(k) = b$	$X(k) = b$
A = 1+total no. of blocks	A = 1

Table 2
Meaning of symbols used in equation 9

$T_{BLK-ROW}$ (i.e. the time taken to make a transition from block level scanning to row level scanning) and $T_{ROW-COL}$ (i.e. the time taken to make a transition from row level scanning to column level scanning) can be set to zero. Moreover, assuming T_{SEL} to be negligible (i.e., the user activates the access switch as soon as a key is focused), the focus movement time (FMT) between these two keys can be represented by the following expression (Equation 9).

$$FMT(k, k') = \{A + \sigma(k') - X(k)\} \times T \tag{9}$$

The variable T in Equation 9 represents the average *scan step*, i.e. the average time required to move the focus from one element (a block, row or column) on the soft keyboard interface to the next element. The meanings of the rest of the symbols present in Equation 9 are summarized in Table 2. It should be noted from Table 2 that the focus movement time among a pair of distinct keys represented by Equation 9 is applicable for all five variants of the matrix scanning mechanisms.

The modified predictive model is obtained by replacing the Fitts law component in Equation 5 with the focus movement time in the following way (Equation 10).

$$CPS = 1/(RT + FMT_m) \tag{10}$$

The mean focus movement time (FMT_m) among any pair of character keys on the soft keyboard is calculated using the following expression (Equation 11). Note that Equation 11 is obtained by modifying Equation 6.

$$FMT_m = \sum_{k=1}^{N} \sum_{k'=1}^{N} (P_{i,j} \times FMT(k, k')) \tag{11}$$

5 System used for Model Validation

We have developed a soft keyboard with scanning to validate our predictive model (Equation 10). The soft keyboard is in Bengali, which is a member of the Indo-Aryan language family. The language is spoken primarily in the eastern Indian regions and Bangladesh. The soft keyboard used to test our model is a modification of an existing system described in [18]. A screenshot of the modified keyboard interface is shown in Figure 1.

The keys on this keyboard are organized in the form of blocks, rows and

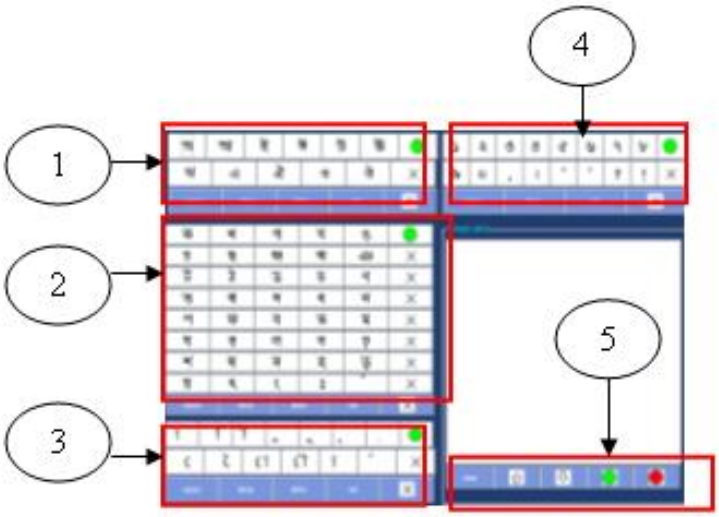

Fig. 1. The soft keyboard interface

columns. A matrix scanning scheme is implemented on this soft keyboard interface that works in the following way; initially, the blocks of the keyboard are highlighted periodically. Once the user selects a block (through activation of an access switch), the rows in the block are periodically highlighted. Activation of the access switch at this stage selects the currently highlighted row. Once a row is selected, the columns of the row are highlighted periodically. To select a column, switch activation is needed. Moreover, once a column is selected, the scanning process goes back to the block highlighting phase again.

Figure 1 shows that there are five blocks present in the keyboard. The blocks are numbered according to the order of scan. Thus, block 1 will be highlighted first, followed by block 2 and so on. Moreover, after block 5, block 1 will be highlighted again. Each of the block contains a number of rows ranging from one (block 5) to nine (block 2). The top-most row in each block is the first one to get the highlighter and the lower-most row in each block is the last. The rows are assigned numbers accordingly.

The columns in each row contain the character keys of the soft keyboard. The order of scanning on these columns is from left to right. Thus, the leftmost column in a row is assigned the column number 1.

5.1 Special Rows

The last row in blocks 1-4 contains some special characters as shown in Figure 2. Among the special keys, the *Space*, *Backspace* and *Enter* have their usual meaning. However, the *yuktakshar* key is a system specific special key. Its function is described below.

In Bengali, two or three alphabetic characters can be combined together to produce a **yuktakshar** or conjugate symbol. To compose a yuktakshar with the soft keyboard, the *yuktakshar* key is used. For example, to compose

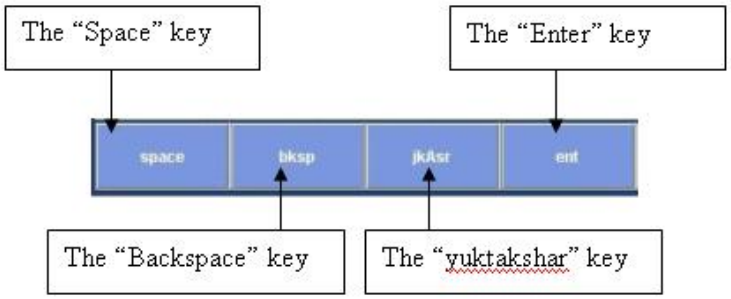

Fig. 2. The special keys in the soft keyboard

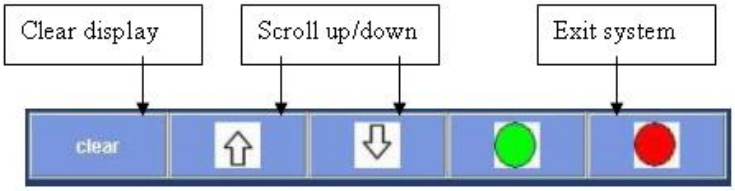

Fig. 3. The special keys in block 5

Fig. 4. The special keys to handle scan error

a two characteryuktakshar, the sequence of selections to be performed by the user is; select first character, select *yuktakshar* key, select second character.

Apart from these special keys, the row in block 5 contains some more special keys shown in Figure 3.

The *clear display* is used to clear the display area (the white part in Figure 1). The *scroll up/down* keys are used to perform scroll up and down operations on the display area using access switches. The *exit system* key is used to exit the keyboard.

5.2 Special keys to Handle Scan Error

Due to their involuntary muscle movement, the user may inadvertently make errors in selecting blocks, rows or columns in the keyboard. While column selection error can be rectified with Backspace key, the block and row selection errors are handled using two special keys, namely *block cancel* and *row cancel*. Figure 4 shows these two keys.

If the user wrongly selects a block during scanning, the user can come out of the block by selecting the *block cancel* key. Same holds true for the *row cancel* key.

6 Language Issues in the Model Prediction and Validation

We have used our model to predict the performance of the soft keyboard described in the Section 5. However, to predict the soft keyboard performance using our model, we had to carry out the following additional tasks.

6.1 Generation of Bengali Digraph Probability Table[3]

To calculate soft keyboard performance, it is necessary to have a table listing the probabilities of all possible diagraph present on the soft keyboard (see Section 2). Although such tables were developed for English (reported in [21][24]), we had to develop such a table ourselves for Bengali to evaluate the performance of the Bengali soft keyboard interface.

The diagraph probability table in Bengali was developed from a corpus[4] which in turn was created from a collection of a large number of editions of the *Anandabazar Patrika*, a vernacular daily newspaper in Bengali. The total size of the corpus was 96,012,779 characters. The digraph probability table created from this corpus has a size of 104×104 $characters^2$. However, the table contains probabilities of digraphs having non-alphanumeric characters as well (for example, digraph probability of *Enter-Space* etc).

It should be noted that the soft keyboard shown in Figure 1 contains some characters for which no digraph probabilities are available as these do not occur in the corpus. These include *Backspace,row cancel,block cancel* and the keys present in block 5. Apart from these, there are 83 distinct keys containing alphanumeric characters and punctuation marks. The probabilities of all possible diagraphs from these 83 characters (there are 83×83 numbers of possibilities) are obtained from the digraph probability table developed from the corpus.

Another point to be noted here is that there are multiple *Space,yuktakshar* and *Enter* keys present on the soft keyboard (Figure 1). While considering a digraph between any key (apart from these three) k and any one of these keys k', the k' which is nearest to k was considered.

6.2 Estimation of Average Bengali Word Length

To measure performance of the soft keyboard in WPM, it was also necessary to know the average length of a Bengali word. Since we did not have any such data with us, we had estimated it using the same corpus described before. In the process, we found that the average word in Bengali is approximately six characters long (including Space). Thus, the performance of the Bengali

[3] we will make the table available to other researchers on request.
[4] The corpus that we used was developed by another institution.

soft keyboard in WPM can be measured with the following expression (Equation 12).

$$Performance(WPM) = CPS \times 60/6 \tag{12}$$

The CPS in Equation 12 above is calculated using Equation 10.

6.3 Test Bench Generation[5]

To evaluate our predictive model, we plan to perform an extensive user trial of the soft keyboard described in Section 5. The user performance observed from the trial will be compared to that predicted by the model. Evaluation of any system through user testing requires use of appropriate test benches. In the context of soft keyboards, such test bench implies a *chunk* of text that will be given to the user. The user will compose the given text using the keyboard. The performance of the user in composing the given text will be taken as the performance of the soft keyboard. In [13], a test bench was described to evaluate soft keyboard performances in English. However, no such test bench exists to evaluate Bengali soft keyboards. In our work, we have developed such a test bench from the Bengali corpus described in Section 6.1.

The test bench we have developed is a chunk of text that will be given to the user. The user will compose the text using a soft keyboard with scanning. To determine the size of the chunk, we have used the following formula (Equation 13).

$$\text{chunk size} = WPM \times \text{duration of typing in minutes} \tag{13}$$

It is generally observed that the performance of a motion impaired user is about 1-3 WPM with computerized text entry interfaces. Accordingly, we have used the value of 1 WPM to calculate chunk size. Moreover, we assumed that the user will compose text for 2 hrs/day for a week. With these assumptions and the fact that the average Bengali word length is 6, the chunk size comes out to be $1 \times 6 \times 60 \times 2 \times 7 = 5040$ characters.

To identify a chunk (test bench) of 5040 characters that is representative of the corpus (and hence the Bengali language), we have used the *cross entropy* based similarity measure [16]. There are 104 distinct characters (comprising of Bengali alphanumeric characters and punctuation symbols) present in the corpus. We calculated the probability of occurrence of each of these characters in the corpus, thus getting the *unigram character probability distribution* of the Bengali alphanumeric and other characters present in the corpus. From this probability distribution, we calculated the *entropy* of the whole corpus. Next, chunks of 5040 characters were generated randomly from the whole corpus. We found that we could generate 32,760 distinct chunks in that way. For each of this randomly generated chunk, we calculated the unigram probability distribution of the characters present in the chunk and with that information, its *cross entropy* with the corpus. The cross entropy is then compared with

[5] We will make the test bench we developed available to other researchers on request.

T (sec.)	WPM (novice)	WPM (expert)
0.5	3.46	6.62
1.0	2.27	3.31
1.5	1.69	2.21
2.0	1.35	1.65
2.5	1.12	1.32
3.0	0.96	1.10

Table 3
The novice and expert users performance predicted by the model

the entropy of the whole corpus. Among all the randomly generated chunk, the chunk with minimum absolute difference between its cross entropy and the entropy of the corpus was selected as the test bench.

7 Model Prediction

We have predicted the performance of the soft keyboard described in Section 5 using our model for both the novice and expert users. For novice users, the performance in WPM is obtained by using Equation 12. To predict performance of expert users, the CPS in Equation 10 was modified by removing the *RT* component and substituting this modified CPS in Equation 12. The predicted performance of the novice and expert users for different values of T (the *scan step*) is summarized in Table 3.

7.1 Problem with Novice Performance Prediction

We have used the reaction time (RT) component of the Fitts-Digraph model to predict the novice users performance. However, Sears et al [20] in their work had demonstrated that the Hick-Hymans law used to predict RT for a novice user is not an appropriate choice. Moreover, in the soft keyboard layout described in Section 5, the character keys are organized into groups. The scanning mechanism provides additional visual cue to the user on the group organized interface. Clearly, prediction of RT in such situation using Hick-Hymans law requires further investigation.

8 Conclusion

In this paper, we have described a model to automatically evaluate user performance for a soft keyboard with scanning. We are presently performing extensive user testing of the soft keyboard described in the paper to validate our predictive model (i.e. how close the prediction is to observation). Even

though we are awaiting the test results, we can make some general comments about the models ability.

Apart from the problem with prediction of novice performance, one important factor not accounted for in the model is the users error. No matter what the level of expertise achieved by the user, s/he is bound to make mistakes. This is more so since the severely motion impaired users sometimes make mistakes due to their involuntary muscle movements. Clearly, in the absence of error, the model predictions are bound to be higher than the actual user performance. Another limitation of the model is that it ignores *learning time* [25] of an interface layout in its performance measure. These show that the model considers only a few of all the human factors involved in the interaction between a user and a soft keyboard interface with scanning. It is necessary to identify and incorporate all these factors into the existing model to provide accurate performance prediction of soft keyboard interfaces in general and for motion impaired users in particular.

9 Acknowledgement

We thank our friends and colleagues at Communication Empowerment Laboratory, IIT Kharagpur to help us in carrying out our research.

References

[1] Arnott, J. L. Text Entry in Augmentative and Alternative Communication. *Proceedings of Efficient Text Entry*, 2005.

[2] Beukelman, R. D. and Mirenda, P. "Augmentative and Alternative Communication," 2nd Eds. Baltimore, MD: Brookes publishing Co., 1998.

[3] Cook, A. M. and Hussey, S. M. "Assistive Technologies: Principles and Practice". 2nd Eds. Mosby-Year Book, 2001.

[4] Fitts, P. M. *The information capacity of the human motor system in controlling the amplitude of movement.* Journal of Experimental Psychology, **47**(1954), 381-391.

[5] Fitts, P. M. *Information Capacity of Discret Motor esponses.* Journal of Experimental Psychology, **67**(1964), 103-112.

[6] Harris, D. and Vanderheiden, G. C. "Augmentative Communication Techniques". R.L. Schiefelbaush Eds. Baltimore: University Park Press, 1980.

[7] Hick, W. E. *On the Rate of Gain of Information.* Quarterly Journal of Experimental Psychology, **4**(1952), 11–36.

[8] Hyman, R. *Stimulus Information as a Determinant of Reaction Time.* Journal of Experimental Psychology, **45**(1953), 188–196.

[9] Johansen, A. S. and Hansen, J. P. Augmentative and Alternative Communication: The Future of Text on the Move. *Proceedings of the seventh ERCIM workshop User Interface for All.* Paris, France, 2002, pp 367–386.

[10] MacKenzie, I. S. *Fitts Law as a Research and Design Tool in Human Computer Interaction.* HumanComputer Interaction, **7**(1992), 91-139.

[11] MacKenzie, I S. and Buxton, W. Extending Fitts' Law to Two-Dimensional Tasks. In Bauersfeld, P., Bennett, J. Lynch, G. (ed.): *Proceedings of the ACM CHI 92 Human Factors in Computing Systems Conference.* California, 1992, 219–226.

[12] MacKenzie, I. S. and Soukoreff, R. W. *Text Entry for Mobile Computing: Models and Methods, Theory and Practice.* Human-Computer Interaction, **17**(2002), 147–198.

[13] MacKenzie, I. S. and Soukoreff, R. W. Phrase sets for evaluating text entry techniques.*CHI '03 extended abstracts on Human factors in computing systems*, Ft. Lauderdale, Florida, US, 2003, 754–755.

[14] MacKenzie, I. S. and Zhang, S. X. The Design and Evaluation of a High Performance Soft Keyboard. *Proc. of CHI99*, 1999, 25-31.

[15] MacKenzie, I. S., Zhang, S. X. and Soukoreff, R. W. *Text Entry using Soft Keyboards.* Behavior and Information Technology, **18**(1999), 235–244.

[16] Manning, C and Schtze, H. "Foundations of Statistical Natural Language Processing". MIT Press. Cambridge, MA, 1999.

[17] Mukherjee, A., Bhattacharya, S., Chakraborty, K. and Basu, A. Breaking the Accessibility Barrier: Development of Special Computer Access Mechanisms for the Neuro-Motor Disabled in India. *Proceedings of the International Conference on Human Machine Interfaces-2004 (ICHMI).* Bangalore, India, 2004, 136–142.

[18] Mukherjee, A., Bhattacharya, S., Halder, P. and Basu, A. A Virtual Predictive Keyboard as a Learning Aid for People with Neuro-motor Disorders. *Proc. 5th IEEE Int. Con. on Advanced Learning Technology (ICALT)*, 2005.

[19] Oneill, P., Roast, C. and Hawley, M. Evaluation of scanning user interfaces using realtime-data usage logs. *Proceedings of the Fourth Annual ACM Conference on Assistive Technologies.* ACM, 2000, 137-141.

[20] Sears, A., Jacko, J. A., Chung, J. C. and Moro, F. *The role of visual search in the design of effective soft keyboards.* In Behaviour and Information Technology - BIT, **20**(2001). pp. 159–166.

[21] Soukoreff, W. and MacKenzie, I. S. *Theorectical upper and lower bounds on typing speed using a stylus and soft keyboard.* Behaviour Information Technology, **14**(1995), 370–379.

[22] Stephanidis, C. and Emiliani, P. L. Design for all in the TIDE ACCESS project. *Proceedings of the 3rd TIDE Congress.* Vol. 4, 1998.

[23] Steriadis, C. E. and Constantinou, P. *Designing Human-Computer Interfaces for Quadriplegic People.* ACM Transactions on Computer-Human Interaction, **10**(2003), 87-118.

[24] Zhai, S., Hunter, M. and Smith, B. A. *Performance Optimization of Virtual Keyboards.* Human-Computer Interaction, **17**(2002), 89–129.

[25] Zhai, S., Sue, A. and Accot, J. Movement Model, Hits Distribution and Learning in Virtual Keyboarding. *Proceedings of CHI-2002*, Minneapolis, Minnesota, US, 2002, 17–24.

[26] Zhang, S. X. "A High Performance Soft Keyboard for Mobile Systems". Unpublished Master of Science Thesis, University of Guleph, Ontario, Canad, 1998.