SOFTWARE VERIFICATION RESEARCH CENTRE

DEPARTMENT OF COMPUTER SCIENCE

THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 96-06

Using Object-Z to specify a web browser interface

Andrew Hussey and David Carrington

May 1996

Phone: +61 7 3365 1003
Fax: +61 7 3365 1533

Using Object-Z to specify a web browser interface

Andrew Hussey and David Carrington
Department of Computer Science, The University of Queensland
Brisbane, Qld, 4072, Australia
email: {ahussey,davec}@cs.uq.edu.au

June 19, 1996

Abstract

A specification describes the functions which a system provides to its users without explanation of how those functions are implemented. A specification of an interactive system's user-interface is an abstract description of the interactive system itself. Such a specification indicates which functional aspects of the system are perceivable by the user and the logical organisation of information provided by the interface. To illustrate, we specify the user-interface for a World-Wide-Web browser. We use Object-Z, a formal specification and design language. Formal methods are increasingly accepted for specifying functional aspects of software systems, however their use in user-interface development is less common. This is despite widespread support for the view that formal methods ease system development. We demonstrate that Object-Z can be used to specify the user-interface for an interactive system. Such a specification may form the basis for an Object-Z user-interface design.

Keywords: User-interface specification, formal methods, web browser, Object-Z, CSP.

1 Introduction

A specification describes the functions which a system provides to its users without explanation of how those functions are implemented. A *specification* of an interactive system's user-interface is simply an abstract description of the interactive system itself. Such a specification indicates which functional aspects of the system are perceivable by the user and the logical organisation of information provided by the interface. Describing the logical organisation of information does not constitute design because the purpose of such description is clarity of the specification. There is no implication that the interactors identified in such a specification would be directly implemented in a design. However, such information may be useful to a designer. Organising information so that a reader can understand the specification may assist in identifying how the *interface* should be structured for enhanced usability and maintainability. de Baar et al. identify "strong similarities between the objects, actions and attributes of the data model and those of the user interface" [9, p.1]. The first step therefore, in the development of an interactive application is the definition of the application's data model. A user-interface specification does not describe the perceptual (visual/audio) aspects of user-interfaces because such a description represents design. For this reason, a specification of the user-interface for an interactive system provides only part of the information needed to construct an implementation.

We use the Object-Z specification and design language to formalise user-interface specification. A brief description of Object-Z is provided in Appendix A. Object-Z is used in software engineering as a specification language. An object-oriented analysis of a system can be described using Object-Z. Because we can use Object-Z to describe the objects which comprise an implementation of a user-interface, Object-Z is also suited to modelling the design of a user-interface. Because of space constraints, this paper does not consider the analysis of the specification to determine user-interface properties.

Formal methods, such as Object-Z, are increasingly accepted for specifying functional aspects of software systems; however their use in user-interface development is less common. This is despite widespread support for the view that formal methods ease system development. Formal methods help reduce ambiguity, inconsistency and incompleteness [5, p.2]. The precision required to write a formal specification assists in detecting "inconsistencies and ambiguities" in a system [13]. Of course, using formal methods does not guarantee correct software; "on the contrary,... they are merely a means of achieving higher integrity systems when applied appropriately" [7]. Formalisms do not guarantee that a design will be aesthetically pleasing or even that it will conform to established guidelines [12, p.14].

Section 2 describes Foley and Van Dam's language model which provides a framework for explaining the scope of interactive system specification. Section 3 describes the interactor model which is an object-oriented model for structuring specifications of interactive systems. We apply the language model in the context of an interactor-based approach to specification. The specification techniques described in Section 3 are applied to a case study in Section 4. Section 5 provides an analysis of the conclusions which may be drawn from this work.

2 The Language Model

Foley and Van Dam (cited in [25]) regard interface implementations as having three distinct perspectives (see Figure 1). These are:

1. semantic — abstract meaning associated with syntactic constructs;

2. syntactic — valid sequences of events;

3. lexical — lexemes comprising events (e.g., keystrokes, mouse actions, visual presentation of system output).

For example, to draw a line in a graphics package, the user selects start and finish points using the mouse (see Figure 1). The mouse manipulations and subsequent output are lexemes,

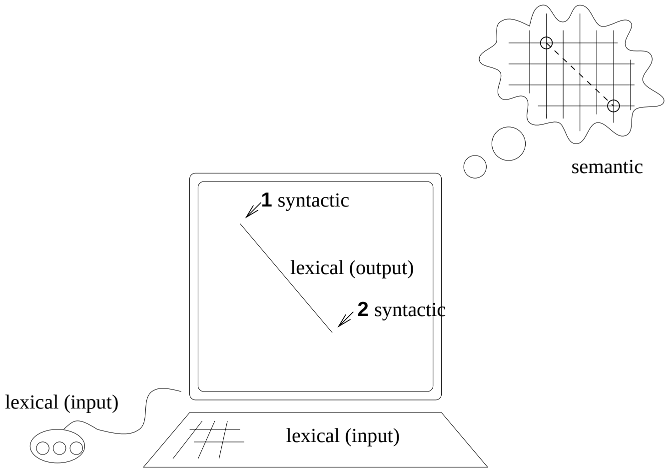

Figure 1: Semantics, Syntax and Lexemes

whereas the required sequence of point selections defines syntax. The abstract semantics of a line are that it is the most direct path between two points.

The syntactic level describes the sequencing of inputs and outputs whereas the focus of semantics is the effect of syntactic events on the state of the system *to the extent which it is perceivable by the user* [6]. Both syntax and semantics can be defined by state-based notations. A syntactic event may be defined by a combination of one or more lexemes [14]. Different implementations may provide differing lexical mappings for particular syntactic events. For example, a syntactic event may be mapped to the concrete actions of clicking on an icon or selecting the appropriate entry from a pull-down menu (or both!). The syntactic and lexical aspects of an interactive system define the *dialogue* between the user and the computer [11, p.252].

Determining lexemes is a *design* activity rather than a specification activity because it involves the description of *how* the system presents its functions to the user. A specification of a system should not constrain the modality (visual, audio etc.) of the interaction [13, p.23]. Modelling lexemes at the specification level would constrain an implementation to use a particular interaction paradigm. Very specific lexical models may even dictate a particular User Interface Management System (UIMS). For these reasons, we do not attempt to deal with lexemes when specifying. Instead we assume that the abstract state of the system is always presented, by the interface, to the user through some modality. The description of lexical aspects of a user-interface design is typically partially informal. In contrast, syntactic and semantic modelling is design only when the issue being considered is *how* to construct the system rather than *what* the system is to achieve. Because a user-interface specification does not consider lexical issues, such a specification can be regarded as an abstract description of the system as a whole. In taking this view, we give up the opportunity to analyse lexemes in the specification phase. However, we gain the simplicity of restricting our initial description of the system to an abstract functional level. Lexical description and analysis can be more appropriately carried out during the design phase. Note however that, for conciseness, we do not consider the analysis of syntactic and semantic properties in this paper.

The language model implies that we should be able to produce separate syntactic and semantic specifications. However splitting syntax and semantics is difficult, to the extent that attempting to separate the two may impair the clarity of a specification. There is usually significant overlap between syntactic and semantic descriptions. At least some parts of a syntactic description will also impinge on semantics and vice-versa so that there is a close relationship between the two. In this situation it is the description of the whole rather than the parts which is important. Loss of clarity in a specification may also arise because of arbitrary distinctions between what falls within the syntactic specification and what is semantic. Further, there is little gain in producing a separate semantic description for analysis purposes, though it may be useful to derive a separate syntactic specification for syntactic analysis. Such a syntactic specification can also assist in distinguishing syntax from semantics in a combined specification.

What is needed is a notation which can be used to specify both syntax and semantics within the one description. In the following section we examine Object-Z as a proposed notation which fulfills these requirements. We introduce interactors as a structural mechanism for user-interface specification using Object-Z.

3 Interactors

Interactive systems can be modelled formally as a collection of interacting agents [3]. The agent model is derived from Sufrin and He's state-based process model of interactive systems [28]. The definition of an agent consists (conceptually) of two parts: one describing the state of the agent and operations upon that state (the semantics of the agent) and the other describing the ordering of events to which the agent can respond (the syntax of the agent) [1]. In most current interactive systems, agents are only reactive, i.e., they are only capable of processing external stimuli [8, p.4].

Agents which communicate directly with the user are called *interactors* [14, 15]. Interactors have both a presentation (i.e., lexical) and a functional (i.e., semantic and syntactic) component. Interactors are system components whose purpose is to mediate between non-interactor agents and the user [14]. Output corresponds to the perceivable state of the interactor, i.e., that part of the system state which the interactor presents to the user. The specification of an interactive system using interactors describes the data which the system presents to the user and the logical organisation of that data. Describing the logical organisation of data does not imply how the system is to be implemented but merely explores what the system must achieve, from the user's perspective. Interactors can be used to model a user-interface at both an abstract (specification) and concrete (implementation) level. The refinement from a specification to an interactor-based implementation is clearer if the specification is also structured using interactors. Refinement of the specification may involve the identification of non-interactor agents which assist in providing interactor functionality.

We use Object-Z [16] to describe both interface semantics and syntax. Describing both syntax and semantics using Object-Z allows us to emphasise their relationship more effectively than if their descriptions were separated. Object-Z is an object-oriented extension of Z [26]. State-based notations, such as Z, have been applied previously to the specification of interactive systems by Sufrin [27, 28] and Duke and Harrison [15, p.17]. Because Object-Z is object-oriented, it can be used to structure specifications as systems of interactors. Furthermore, Object-Z is a formal notation, therefore using Object-Z provides the many benefits of formal methods, some of which we described in Section 1. An Object-Z specification consists of classes which together describe the functionality of the system. Each class has a state and operations upon that state. Object-Z extends Z to deal with concurrency (by adding the operator $\|$). The operators $\|$ and $\fatsemi$ permit temporal relationships between operations. However, although Object-Z operations can describe sub-traces for the system, the sequence in which operations may occur is not explicit. Supplementary process-based descriptions can be used to clarify the allowable sequences of operations.

Interactor-based analysis, using Object-Z is compatible with a task-oriented view of a user-interface. Task analysis is oriented toward specifying state transitions from the user's perspective, hence entire tasks are described rather than the state transitions of each interactor. An interactor-based analysis differs from a task analysis because an interactor-based analysis focuses on the behaviour and interactions of *each* interactor constituting the user-interface. The integration of task analysis and interactor-based analysis offers a means of reconciling interface design and software engineering methods. In practice, the identification of functionality and objects often proceeds in parallel [24, p.52]. The link between task analysis and interactor-based analysis is sufficiently strong that an interactor-based description of a user-interface may be derived from a task analysis of that system. Paterno and Mezzanotte

[23] have shown that a task analysis may be refined by a series of transformation rules to an interactor-based architectural description.

We do not attempt to distinguish those aspects of the Object-Z specification which relate to the sequencing of operations and those which do not because syntax and semantics are usually highly interdependent. Instead, we provide a CSP [22] description of interactor syntax which is intended as an aid to the reader in understanding the sequencing of operations defined by the Object-Z and which could be used to analyse syntactic properties. Usually such supplementary description would be unnecessary but we provide it here to explain the syntactic behaviour which the Object-Z captures. We use CSP because event-based notations such as CSP have better scalability than either grammars or state-transition networks [20]. CSP has been applied to syntax description by Alexander [4] and by Abowd [2]. Analysis of the specification syntax is simplified by the provision of a separate syntactic specification. However, because the Object-Z specification describes both syntax and semantics, care needs to be taken that the syntactic description is consistent with the Object-Z.

An object-oriented notation which is conceptually similar to Object-Z has been developed and applied to the specification of user-interfaces by Abowd [3]. Abowd's model divides interacting agents into three sub-elements: the internal state-based specification, the external event-based specification and the communication specification linking internal operations and external events. Abowd's notation is a hybrid of Z and CSP [3, p.104]. However, the notation is not widely used whereas Object-Z is comparatively well known and while Abowd's notation is as expressive as Object-Z, Object-Z is more flexible (Abowd only permits the combination of whole interactors, rather than the combination of interactor operations). In addition, Object-Z can be used as a *design* language when constructing the user-interface and functional components of a system.

Likewise, Fields, Harrison and Wright have modelled several systems using an interactor notation [17, 18]. The internal specification of an interactor (i.e., interactor semantics) is modelled using VDM and the external specification (i.e., interactor syntax) is modelled using CSP. Their work differs from our approach in that interactor rendering is regarded as a specification concern and both system and environmental entities (e.g., the user) are modelled as interactors. In addition, the descriptions of external and internal behaviour describe distinct domains. This enables external specifications to be modified without altering the internal specification. However, where there exists a choice between interaction events, the availability of those events is dictated by the internal specification so syntactic information is split across internal and external descriptions. We prefer to specify both syntax and semantics using Object-Z (with supplementary CSP descriptions of syntax) because the separation of syntax and semantics is often difficult.

In Section 4, we develop a case study based on a generic "web browser". A web browser is a program which provides a user-perceivable representation of the information provided by the World-Wide-Web (the "web"). We apply interactors as a structural mechanism for reducing complexity. Interactors allow us to structure the specification according to the logical organisation of information provided by the interface. Grouping related data and operations in this way inherently simplifies the object interactions in the specification. Additionally, because interactors group related data and operations, an interactor specification provides clues to the designer as to how the interface implementation's structure ought to appear *to the user*.

4 Case Study

The current extensive use of the web motivates its choice as an area for study. Each year, web-usage multiplies and this trend appears likely to continue. In this section we present a specification of a user-interface for a generic web browser. Our specification is divided into several interactors. For each interactor, the specification is in two sections: the first section defines the semantics of the interactor using Object-Z [1]; the second explicitly defines syntax using CSP.

The Dexter model of hypertext [21] provides a starting point from which part of the state of a web browser may be defined because much of the data manipulated by a web browser is hypertext. However the Dexter model is far more complex than the hypertext models actually used by most existing web browsers because the Dexter model provides many additional capabilities such as bi-directional links between pages. In this specification, we use only a few central ideas from the Dexter model. Our model is similar to other (simplified) models of hypertext, e.g., [10, 29]. In particular, we regard our hypothetical system as operating on data whose structure can be treated as consisting of components organised into groups called *pages*. Only one page is accessible by the user at a time. We only deal with the *http* aspects of the web, so each link within the web is to another component and page in the web. A *TextComponent* is represented by a sequence of characters. A *GraphicComponent* is represented by one or more pictorial regions in the browser's display. We are not concerned with the internal details of text and graphic components. A component is either a text component or graphical component.

$$Component == TextComponent \bigcup GraphicComponent$$

We model a page as consisting of one or more components. Some components may be anchors. An anchor is a component whose selection leads to an alteration of the currently viewed page. We regard the mappings between pages as unidirectional, although other researchers have modelled such links as bidirectional. The current component of the page ($curr$) indicates the location within the page which the user is viewing. Selecting a component which is an anchor, produces the page (and hence component) in the web to which that anchor refers. Note that

[1]The specifications comprising the case study have been checked using the Wizard type checker which was developed by the Software Verification Research Centre at The University of Queensland.

we do not provide accessor operations on the interactor state because the absence of a visibility list (see Appendix A) means that the state of the object is accessible (e.g., for an instance $p : Page$, the attribute *contents* can be accessed as $p.contents$).

Page

$$
\begin{array}{l}
contents : \mathrm{seq}_1\ Component \\
anchors : Component \nrightarrow Page \\
curr : \mathbb{N}_1 \\
\Delta \\
component : Component \\
\hline
\mathrm{dom}\ anchors \subseteq \mathrm{ran}\ contents \\
component = contents(curr) \\
curr \leq \#contents
\end{array}
$$

PreviousComponent

$$
\begin{array}{l}
\Delta(curr, component) \\
\hline
curr > 1 \Rightarrow curr' = curr - 1 \\
curr = 1 \Rightarrow curr' = curr
\end{array}
$$

NextComponent

$$
\begin{array}{l}
\Delta(curr, component) \\
\hline
curr < (\#contents) \Rightarrow curr' = curr + 1 \\
curr = (\#contents) \Rightarrow curr' = curr
\end{array}
$$

Select

$$
\begin{array}{l}
source? : Component \\
dest! : Page \\
\hline
source? \in \mathrm{dom}\ anchors \\
dest! = anchors(source?)
\end{array}
$$

The *Navigator* interactor provides the user's interface to the pages and components which form the web. Initially the navigator views the user's "home" page. The user has a current location within the web which can be altered by the user clicking on an anchor component. The user's interactions with the system build a sequence of locations within which the user can move backward and forward.

Navigator

$$
home : Page
$$

$$
\begin{array}{l}
history : \mathrm{seq}_1\ Page \\
curr : \mathbb{N}_1 \\
\Delta \\
currlocn : Page \\
\hline
currlocn = history(curr)
\end{array}
$$

JumpTo

$$
\begin{array}{l}
\Delta(history, curr) \\
dest? : Page \\
\hline
history' = (\{1 \mathrel{..} curr\} \lhd history) \frown \langle dest? \rangle \\
curr' = curr + 1
\end{array}
$$

$\underline{\text{INIT}}$
$history = \langle home \rangle$
$curr = 1$

$\underline{CurrLocn}$
$current! : Page$

$current! = currlocn$

$\underline{Back}$
$\Delta(curr)$

$curr > 1$
$curr' = curr - 1$

$\underline{Forward}$
$\Delta(curr)$

$curr < \#history$
$curr' = curr + 1$

$LinkClick \mathrel{\widehat=} currlocn.Select \parallel JumpTo$
$NextComponent \mathrel{\widehat=} currlocn.NextComponent$
$PreviousComponent \mathrel{\widehat=} currlocn.PreviousComponent$

Navigation within the web is restricted by the proviso that the user cannot backtrack further than the first location viewed in the current session. A new location can be added to the sequence, either by the user clicking on a hyperdata link or specifying the page explicitly. If the user is currently viewing the last component entered into the sequence, they cannot move forward. A click by a user on an anchor component alters the current web location displayed by the browser. Note that operations such as *LinkClick* are modelled as atomic (although this is clearly not how the system would be implemented). A *design* of the system may refine *LinkClick* to describe the behaviour of the system while it is waiting for data to be transmitted, however this is not a specification concern. In general, there are many such syntactic and semantic issues with which we do not deal, because they consider matters which can be regarded as design concerns (how to implement the system) rather than specification concerns (what the system must achieve)

The syntax of the *Navigator* interactor, for navigation between web pages, is modelled using CSP. We choose not to model the manipulation of the current component within the current page because the syntax involving movement between components is clear and adding it to the page navigation syntax would reduce clarity. We also do not include the operation *CurrLocn* because that operation can be invoked at any time. It is not dependent on the prior sequence of operations. Further, *CurrLocn* is used only for communicating with the *HotList* interactor, which we describe below. It is not available to the user. We provide a CSP model of the interactor syntax to explain the syntactic behaviour which the Object-Z captures. In addition, a supplementary description of syntax allows the reader to concentrate on purely semantic issues of individual operations without concern for what global sequences of operations are valid or invalid.

$$\alpha\mathbf{NAVIGATOR} = \{LinkClick, Forward, Back, JumpTo\}$$
$$\mathbf{NAVIGATOR} = END_0$$
$$END_0 = x : \{LinkClick, JumpTo\} \rightarrow END_1$$
$$\begin{aligned} END_j = {} & x : \{LinkClick, JumpTo\} \rightarrow END_{j+1} \mid \;\; (j > 0) \\ & Back \rightarrow WITHIN_{j-1} \end{aligned}$$

The user can navigate forward through the sequence of locations until they are viewing the last location in the sequence. Likewise, the user can navigate backward until they are at the first location in the sequence. A click on an anchor alters the sequence so that the new web location becomes the last element in the sequence (i.e., the page most recently accessed).

$$\begin{aligned} WITHIN_k = {} & Forward \rightarrow WITHIN_{k+1} \mid \;\; (k > 0 \wedge k < j) \\ & Back \rightarrow WITHIN_{k-1} \mid \\ & x : \{LinkClick, JumpTo\} \rightarrow END_{k+1} \\ WITHIN_j = {} & END_j \\ WITHIN_0 = {} & Forward \rightarrow WITHIN_1 \mid \\ & x : \{LinkClick, JumpTo\} \rightarrow END_1 \end{aligned}$$

There are some behaviours which we may wish to include in our web browser specification but which, on closer examination, are not specification concerns. For example we may require that the browser provide a user-accessible operation "reload" which re-loads the current web page. However such an operation has no effect on the state of the *Navigator* interactor. Furthermore the "reload" operation has no effect on the user's capacity to perform other actions. In this sense the re-load is lexical, so it is not strictly a specification concern. Such operations would be dealt with during design, when considering how the current page was to be displayed.

We introduce a *VisitableList* interactor which will form the basis for several interactors. A visitable list allows the selection and retrieval of a web location.

VisitableList

$$contents : \mathrm{seq}\ Page$$
$$selection : \mathbb{P}\,\mathbb{N}$$
$$\#selection \leq 1$$

Visit

$$dest! : Page$$
$$selection \neq \varnothing$$
$$dest! \in \mathrm{ran}(selection \lhd contents)$$

INIT

$selection = \varnothing$

Select

$\Delta(selection)$
$new? : \mathbb{N}$

$\#contents > 0$
$new? \in \operatorname{dom} contents$
$selection' = \{new?\}$

Cancel

$\Delta(selection)$

$selection' = \varnothing$

Add

$\Delta(contents)$
$dest? : Page$

$contents' = contents \frown dest?$

The history list interactor is simply a visitable list. As the user traverses the web via the navigator interactor, web locations are added to the history list.

HistoryList

$VisitableList$

The history list interactor allows the user to select and view components from a list of previously accessed components. Until a selection has been made, the operation *Visit* is unavailable. The corresponding CSP description follows:

$$
\begin{aligned}
&\alpha\mathbf{HISTORY} = \{Select, Visit, Add, Cancel\}\\
&\mathbf{HISTORY} = Select \rightarrow SELECTED\_HISTORY \mid\\
&\qquad Add \rightarrow HISTORY \mid\\
&\qquad Cancel \rightarrow HISTORY\\
&SELECTED\_HISTORY = Select \rightarrow SELECTED\_HISTORY \mid\\
&\qquad Visit \rightarrow SELECTED\_HISTORY \mid\\
&\qquad Add \rightarrow SELECTED\_HISTORY \mid\\
&\qquad Cancel \rightarrow HISTORY
\end{aligned}
$$

The hotlist interactor is a visitable list for which the user controls the deletion of elements.

HotList

$VisitableList$

DeleteCurrent

$\Delta(contents, selection)$

$\exists\, n : selection \bullet$
$\quad contents' = squash(\{n\} \lhd contents)$
$selection' = \varnothing$

The corresponding CSP specification is similar to that for the *HistoryList* interactor, but also allows the user to delete the current selection from the list.

The web browser may be defined by instantiating interactors whose behaviour corresponds to the classes defined above. We promote operations of the instantiated interactors to the top-level.

$$
\begin{array}{l}
\textit{WebBrowser} \\
\hline
\quad \textit{navigator} : \textit{Navigator} \\
\quad \textit{history} : \textit{HistoryList} \\
\quad \textit{hotlist} : \textit{HotList} \\
\hline
\textit{Forward} \mathrel{\widehat=} \textit{navigator}.\textit{Forward} \\
\textit{Back} \mathrel{\widehat=} \textit{navigator}.\textit{Back} \\
\textit{NextComponent} \mathrel{\widehat=} \textit{navigator}.\textit{NextComponent} \\
\textit{PreviousComponent} \mathrel{\widehat=} \textit{navigator}.\textit{PreviousComponent} \\
\textit{HistorySelect} \mathrel{\widehat=} \textit{history}.\textit{Select} \\
\textit{HistoryCancel} \mathrel{\widehat=} \textit{history}.\textit{Cancel} \\
\textit{HotListSelect} \mathrel{\widehat=} \textit{hotlist}.\textit{Select} \\
\textit{HotListCancel} \mathrel{\widehat=} \textit{hotlist}.\textit{Cancel} \\
\textit{LinkClick} \mathrel{\widehat=} \textit{navigator}.\textit{LinkClick} \parallel \textit{history}.\textit{Add} \\
\textit{HistoryJumpTo} \mathrel{\widehat=} \textit{history}.\textit{Visit} \parallel \textit{navigator}.\textit{JumpTo} \parallel \textit{history}.\textit{Add} \\
\textit{HotListJumpTo} \mathrel{\widehat=} \textit{hotlist}.\textit{Visit} \parallel \textit{navigator}.\textit{JumpTo} \parallel \textit{history}.\textit{Add} \\
\textit{AddCurrent} \mathrel{\widehat=} \textit{navigator}.\textit{CurrLocn} \parallel \textit{hotlist}.\textit{Add}[\textit{current}?/\textit{dest}?] \\
\textit{DeleteCurrent} \mathrel{\widehat=} \textit{navigator}.\textit{CurrLocn} \parallel \textit{hotlist}.\textit{DeleteCurrent} \\
\hline
\end{array}
$$

Note that some operations such as *history.Add* and *navigator.CurrLocn* are not directly accessible by the user. Also, because we are specifying rather than designing, we do not indicate how the user invokes operations such as *LinkClick*. For example, a design based on an architecture derived directly from the specification might give control to the *navigator* interactor. However our specification is not concerned with this.

The specification can be used to check user-interface properties however, because our specification is only small and the focus of the paper is specification, rather than analysis of properties, we do not consider the derivation of interface properties in depth. Fields et al. have identified a number of classes of errors defined as properties of the sequences of events which are required to perform a task [19, p.191]:

1. omission of events (or whole sequences of events);

2. substitution of events (one for another);

3. reversal of two events (or two sequences of events).

To prevent undesirable traces of events occurring, additional interface moding may be required [19, p.193]. For example, the *Navigator* interactor does not permit the user to *JumpTo* a web page unless they have first selected a page to jump to. This implies that the a jump is not possible unless a page identifier has been supplied by the user (implicitly, a bad page identifier will produce an error message). Where moding is not desired, it may be sufficient to simply provide more information to the user (i.e., regarding the consequences of a user action) [19, p.193]. After recognising the absence of moding, the designer can take steps to ensure sufficient information is available to the user in the design. A specification of a user-interface provides a designer with the capacity to analyse *fundamental* properties of the system such as moding.

5 Conclusions

The case study demonstrates that formal methods are useful for describing the syntax and semantics of user-interfaces. Web browsers are complex systems for which a precise formal model provides valuable documentation. Further, the case study shows that using Object-Z helps the specifier to structure the user-interface specification using interactors and an explicit CSP model helps clarify syntax.

An Object-Z specification of a user-interface provides an initial description from which design issues may be assessed. An interactor-based Object-Z *design* of the system can be derived from an Object-Z specification. Because Object-Z is used as a design notation in software engineering, Object-Z specification and design provides a bridge between software engineering and human-computer-interface concerns. An Object-Z design may be used to construct a system and to evaluate the system's lexical properties. In contrast, the Object-Z specification permits evaluation of preliminary syntactic and semantic issues without concern for the lexical issues which dominate human-computer-interface implementation and we believe this separation of concerns is helpful. Because user-interface specification is not concerned with lexical issues, a user-interface specification is essentially an abstract specification of the system itself (of which the user-interface is only a part). The user-interface specification provides a view (the view provided by the interface to the user) of a potentially much larger system. Specific techniques for deriving Object-Z designs from Object-Z user-interface specifications are a topic of future research. Such designs may reflect architectural models which can be super-imposed on the interactor model, such as MVC.

There are many issues which have not been considered in this paper but which are suitable for further research. We have not considered the proof of syntactic properties from a CSP description because such issues have been the focus of much previous research and our focus in this paper has been on methods for specification rather than the proof of properties from such specifications. Additionally, we have not considered how Object-Z might be used to prove semantic properties. In particular, we need to assess the ease with which such proofs may be conducted in comparison with other specification methods.

6 Acknowledgments

We thank Jay Burmeister for his insightful comments on earlier drafts of this paper. We also thank Wendy Johnston whose *Wizard* type-checker for Object-Z type-checked the specifications in this paper.

References

[1] G. D. Abowd. Agents: Communicating Interactive Processes. In D. Diaper, D. Gilmore, G. Cockton, and B. Shackel, editors, *Human-Computer Interaction – INTERACT '90*, pages 143–147. Elsevier Science Publishers B.V., 1990.

[2] G. D. Abowd. *Formal Aspects of Human-Computer Interaction*. PhD thesis, Oxford University Computing Laboratory; Trinity College, October 1991.

[3] G. D. Abowd. Formal descriptions of user interfaces. *Colloquium Digest*, 192:7/1–7/3, 1991.

[4] H. Alexander. *Formally-based Tools and Techniques for Human-Computer Dialogues*. Ellis Horwood, 1987.

[5] M. J. Aslett. *A Knowledge Based Approach to Software Development: ESPRIT Project ASPIS*. North Holland, 1991.

[6] C. Attiogbe and J. Durieux. Handling interaction in software specification. In T. Grechenig and M. Tscheligi, editors, *Lecture Notes in Computer Science*, volume 733, pages 155–170. IRIT-Universite Paul SABATIER, Springer-Verlag, 1993.

[7] J. P. Bowen and M. G. Hinchey. Ten Commandments of Formal Methods. *IEEE Computer*, 28(4):56–63, April 1995.

[8] J. Coutaz, L. Nigay, and D. Salber. Agent-Based Architecture Modelling for Interactive Systems. Technical Report SM/WP53, LGI-IMAG, Grenoble, April 1995. ESPRIT BRA 7040 Amodeus-2.

[9] Dennis J. M. J. de Baar, James D. Foley, and Kevin Mullet. Coupling application design and user interface design. Retrieved from ftp://ftp.gvu.gatech.edu/pub/gvu/tech-reports/91-10.ps.Z, August 1991.

[10] M. d'Inverno and M. Priestley. Structuring Specifications in Z to build a Unifying Framework for Hypertext Systems. In *Z User Meeting '95*, pages 81–102. Springer-Verlag, 1995.

[11] A. Dix, J. Finlay, G. Abowd, and R. Beale. *Human-Computer Interaction.* Prentice Hall, 1993.

[12] A. J. Dix. *Formal Methods for Interactive Systems.* Academic Press, 1991.

[13] D. J. Duke and M. D. Harrison. From Formal Models to Formal Methods. In R. N. Taylor and J. Coutaz, editors, *ICSE Workshop on SE-HCI: Joint Research Issues: Sorrento, Italy*, Lecture Notes in Computer Science, pages 159–173. Springer-Verlag, 1994.

[14] D. J. Duke and M. D. Harrison. Event model of human-computer interaction. *Software Engineering Journal*, pages 3–11, January 1995.

[15] D. J. Duke and M. D. Harrison. Mapping user requirements to implementations. *Software Engineering Journal*, pages 13–20, January 1995.

[16] R. Duke, G. Rose, and G. Smith. Object-Z: a Specification Language Advocated for the Description of Standards. Technical Report 94-45, Software Verification Research Centre, Dept. of Computer Science, University of Queensland, 1994.

[17] B. Fields, M. Harrison, and P. Wright. From Informal Requirements to Agent-Based Specification. *SIGCHI Bulletin*, 26(2), April 1994.

[18] B. Fields, M. Harrison, and P. Wright. Modelling Interactive Systems and Providing Task Relevant Information. In Fabio Paterno, editor, *Interactive Systems: Design, Specification and Verification - 1st Eurographics Workshop*, pages 253–266. Springer-Verlag, June 1994.

[19] B. Fields, P. Wright, and M. Harrison. Applying Formal Methods for Human Error Tolerant Design. In R. N. Taylor and J. Coutaz, editors, *ICSE Workshop on SE-HCI: Joint Research Issues: Sorrento, Italy,*, Lecture Notes in Computer Science, pages 184–195. Springer-Verlag, 1994.

[20] M. Green. A Survey of Three Dialogue Models. *ACM Transactions on Graphics*, 5(3):244–275, 1986.

[21] F. Halasz and M. Schwartz. The Dexter Hypertext reference model. *Communications of the ACM*, 37(2):30–39, February 1994.

[22] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, 1985.

[23] F. Paterno and M. Mezzanotte. Formal Analysis of User and System Interactions in the CERD Case Study. Technical Report SM/WP48, CNR CNUCE, Pisa Italy, February 1995. ESPRIT BRA 7040 Amodeus-2.

[24] M. B. Rosson and J. M. Carroll. *Extending the Task-Artifact Framework: Scenario-Based Design of Smalltalk Applications*, volume 9 of *Advances in human-computer interaction*, chapter 2, pages 31–57. Norwood, 1993.

[25] B. Shneidermann. *Designing the user interface : strategies for effective human-computer interaction*. Addison-Wesley, 1992.

[26] J. M. Spivey. *The Z notation: a reference manual*. Prentice-Hall, 2nd edition, 1992.

[27] B. Sufrin. Formal Specification of a Display-Oriented Text Editor. *Science of Computer Programming*, 1:157–202, 1982.

[28] B. Sufrin and J. He. *Formal Methods in Human-Computer Interaction*, chapter 6 - Specification, analysis and refinement of interactive processes, pages 153–200. Cambridge University Press, Cambridge, 1990.

[29] B. Wang, T. Holden, and P. Hitchcock. An Object Oriented Database Approach for Supporting Hypertext. In C. Rolland, F. Bodart, and C. Cauvet, editors, *Advanced Information Systems Engineering 5th International Conference: CAiSE '93*, pages 601–628. Springer-Verlag, 1993.

Appendix A: Modelling using Object-Z

In an Object-Z description, a class is defined by a named box encapsulating an export list, state, initialisation and operations. The export list (prefixed by $\upharpoonright$), defines which operations are visible to users of the class. If there is no export list, then implicitly all operations are visible. The state schema is un-named and contains attribute declarations and a constraining invariant. Some attributes may be dependent on other attributes (i.e., their value is always derivable from those other attributes). Such dependencies are denoted by a Δ on a separate line preceding the dependent attributes (e.g., *top*). The nature of this dependency is captured by the class invariant. The initialisation schema is labelled INIT and defines the initial state of instances of the class. An operation schema is divided into two parts. The upper part defines the context of the operation including inputs and outputs. The Δ-list defines which state attributes are altered by an operation. State variables which are not listed in the delta-list are unchanged by an operation. The lower part defines a predicate relating the initial and final states of the operation. A simple example class defining a stack of natural numbers appears below:

$Stack$

$\upharpoonright(Push, Pop, Top)$

$$
\begin{array}{l}
elements : \mathrm{seq}\,\mathbb{N} \\
\Delta \\
top : \mathbb{N} \\
\hline
top = last(elements)
\end{array}
$$

$INIT$

$$
\begin{array}{l}
elements = \langle\, \rangle
\end{array}
$$

Top

$$
\begin{array}{l}
top! : \mathbb{N} \\
\hline
top! = top
\end{array}
$$

$Push$

$$
\begin{array}{l}
\Delta(elements) \\
num? : \mathbb{N} \\
\hline
elements' = elements \frown \langle num? \rangle
\end{array}
$$

Pop

$$
\begin{array}{l}
\Delta(elements) \\
num! : \mathbb{N} \\
\hline
elements = elements' \frown \langle num! \rangle
\end{array}
$$

Classes may inherit from other classes. Inheritance results in a merger of state, $INIT$ and operation schemas which have the same name. For example, we define a new stack class which can return its size:

$NewStack$

$Stack$

$Size$

$$
\begin{array}{l}
size! : \mathbb{N} \\
\hline
size! = \#elements
\end{array}
$$

A class may be instantiated. For example, the declaration $s : Stack$ defines an attribute s of type $Stack$. The polymorphic operator ($\downarrow$) defines the type of an attribute as any class which inherits from the specified base class. Operations applied to such an attribute must be polymorphic, i.e. regardless of the actual class of the object, the operation must be able to proceed. For example, the declaration $stacks : \downarrow Stack$ defines an attribute $stacks$ which can be an instance of any class which inherits from $Stack$. The containment operator (©) defines an ownership relationship between class instances. For example, the declaration $stacks : \mathbb{P} \downarrow Stack_{©}$ ensures that the instances of type $\downarrow Stack$ which define the set $stacks$ can appear in the state of no other object instance but can be referenced.

We can define a set of stacks as follows:

$$
\begin{array}{l}
\textit{StackSet} \\
\hline
\quad \textit{stacks} : \mathbb{P} \downarrow \textit{Stack}_{©} \\
\\
\quad \textit{INIT} \\
\quad \textit{stacks} = \varnothing \\
\\
\quad \textit{AddStack} \\
\quad \Delta(\textit{stacks}) \\
\quad s? : \downarrow \textit{Stack} \\
\quad \hline
\quad \textit{stacks}' = \textit{stacks} \cup \{ s? \} \\
\\
\quad \textit{Swap} \mathrel{\widehat=} [\, s1?, s2? : \textit{stacks} \,] \bullet s1?.\textit{Pop} \parallel s2?.\textit{Push} \\
\quad \textit{Push} \mathrel{\widehat=} [\, s? : \textit{stacks} \,] \bullet s?.\textit{Push} \\
\quad \textit{Pop} \mathrel{\widehat=} [\, s? : \textit{stacks} \,] \bullet s?.\textit{Pop} \\
\quad \textit{TwoPops} \mathrel{\widehat=} [\, s1?, s2? : \textit{stacks} \,] \bullet s1?.\textit{Pop} \mathbin{;} s2?.\textit{Pop} \\
\quad \textit{PopAll} \mathrel{\widehat=} \bigwedge s : \textit{stacks} \bullet s.\textit{Pop} \\
\quad \textit{PopEither} \mathrel{\widehat=} [\, s1?, s2? : \textit{stacks} \,] \bullet s1?.\textit{Pop} \; [] \; s2?.\textit{Pop} \\
\quad \textit{Top} \mathrel{\widehat=} [\, s? : \textit{stacks} \,] \bullet s?.\textit{Top} \\
\quad \textit{Size} \mathrel{\widehat=} [\, s? : \textit{NewStack} \mid s? \in \textit{stacks} \,] \bullet s?.\textit{Size}
\end{array}
$$

Operations and attributes of class instances may be accessed. For example, the term $s.elements$ denotes the attribute *elements* of s while the term $s.Push$ denotes the operation *Push* applied to s. Operations may be combined using the Z schema calculus [26], including conjunction ($\wedge$) and sequential composition ($\mathbin{;}$). Object-Z provides two further operators:

- concurrency ($\parallel$) which conjoins operation schemas (i.e., merges state and predicates) and identifies inputs with outputs. We illustrate concurrency in the operation *Swap*, which pops one stack and pushes the result on the other.

- angelic choice ([]) which selects one of two operations according to which can be performed. If both operations can be performed, a non-deterministic choice is made between the two. We illustrate angelic choice in the operation *PopEither* which pops either of two selected stacks.

Operators may be applied over a set of instances, e.g., the operation *PopAll*, specifies an operation in which all instances in the set *stacks* are subject to the operation *Pop*. The meaning is the same as $s_1.Pop \wedge s_2.Pop \wedge \ldots$ for all s_1, s_2, ... in the set *stacks*.