SOFTWARE VERIFICATION RESEARCH CENTRE

DEPARTMENT OF COMPUTER SCIENCE

THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 96-07

A formal basis for modelling process and task management aspects of user interface design

Peter A. Lindsay

May 1996

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

A formal basis for modelling process and task management aspects of user interface design

Peter A. Lindsay
email: pal@cs.uq.edu.au

To appear:
Proceedings of BCS FACS Workshop *Formal Aspects of The Human Computer Interface*, Sheffield, UK, September 1996 (Springer-Verlag).

Abstract

This paper presents a method for formally specifying and reasoning about process models for interactive systems. The method addresses two important aspects of user interface design: controlled but flexible access to functionality; and provision of useful task management information, such as indicating what progress has been made towards achieving goals and what remains to be done.

The method is well suited to "data intensive" applications in which the system is being used to manage complex "configurations" of interconnected objects, and for which task goals can be expressed in terms of properties of the underlying configuration of objects. The method includes proof obligations to check the accuracy of the task management information.

The paper illustrates the method on a Theory Manager, which manages a store of theorems and proofs; the store has complex consistency and completeness requirements.

1 Introduction

This paper describes a formal method for modelling and reasoning about functionality aspects of User Interface (UI) design for interactive systems. The method is primarily applicable to systems in which

- the user constructs and maintains large or complex "configurations" of objects and relationships between objects, and
- the user's task goals can be expressed in terms of establishing or maintaining consistency and completeness conditions on the underlying store of objects.

Common examples of such systems include programming environments, software engineering environments, hypermedia networks, airline reservation systems, and so on. For example, in a programming environment the objects might be individual program statements, and there might be consistency checks such as ensuring that correct syntax has been used, and completeness checks such as checking that all variables have been initialized, all paths have been tested, and so on. In a software engineering environment the objects might be coarse-grained objects such as software development documents, or finer-grained objects such as individual definitions in a specification [15].

For such systems, two of the most important aspects of UI design are

1. controlled and flexible access to functionality. By controlling how objects are accessed and changed, the UI can play an important role in maintaining consistency of the object store; on the other hand, systems which enforce consistency (e.g. syntax-directed editors) can be annoyingly inflexible to use.

2. provision of useful task management information, such as how far the user has gone towards meeting their goals, and what remains to be done. When the consistency and completeness conditions are complex or hard to check, the availability of such information can be crucial to the system's usability. In high integrity applications, the accuracy of such information can be of critical importance.

Process modelling has been proposed as a way of guiding and controlling access to functionality, and a number of process modelling languages are now available (e.g. [13, 14]). State-based process models are ones in which certain objects are assigned a state – or *status* as we prefer to call it, so we can use the term state in a more general sense. The status of an object is a representation of what the system "believes" about the object based on what process steps have taken place. Guided by status information, the process engine invokes tools and changes objects and their statuses accordingly.

This paper presents a systematic method – originally proposed in [16] – for formally specifying and reasoning about state-based process models and describes how object statuses can be used to provide useful task management information. (Note that we do not however consider the problem of task analysis here.) The method includes proof obligations for verifying the accuracy of task management information and for checking control of access to functionality. The method could be used in conjunction with more general methods for modelling interactive systems, such as interactors [6]. VDM-SL notation [3, 12] is used for formal definitions here, although other state-based formal specification notations could be used equally as well.

The method is outlined briefly in Section 2 below and illustrated in detail on an example application in Sections 4-6. The application concerns a specialized object-store management system which, although limited in its functionality, has complex, critical consistency and completeness conditions. We describe a process model for a flexible style of interaction in which users can take the object store into "inconsistent" states and recover from inconsistencies in a controlled manner. The process engine itself makes use of the task management information to optimize its performance. Some of the correctness proofs are sketched.

2 Outline of the method

2.1 The problem

Suppose the UI designer is given a set of application tools for a system such as described above, together with an informal description of the user's tasks. The designer's problem is how to design a coherent UI for the tool-set which:

- gives users access to the functionality they desire;
- prevents access to operations that are meaningless, irrelevant or undefined;
- reports on the state of progress in the task; and
- indicates how further progress can be made.

The following factors may affect the design:

- The task may or may not have a pre-defined "final goal": e.g. the task may simply involve ongoing processing of new inputs.
- The tools may have "prerequisites" (constraints on how they can be used): e.g. a back-end compiler may require the source program to be syntax correct before it can generate object code.
- There may be performance requirements: e.g. the user may be willing to accept slow response times on "critical" operations (such as those which have the potential to avoid a lot of subsequent re-work) but may expect quick responses on non-critical operations. Formalizing performance requirements is however beyond the scope of the method discussed here.

We assume the designer has a good understanding of the application domain and of formal modelling techniques.

2.2 The method

The following method can be used to specify and reason about possible designs for the UI:

1. Build a formal model of the application domain as follows:
   (a) Define the data model: i.e., the kinds of object in the system and the possible relationships between them.
   (b) Specify the functionality of the tools, including their input and output types, the input/output relationships they establish, and any prerequisites they may have. The tool descriptions should be "functionalized" (i.e., not use internal states).
   (c) Define the task goals, including intermediate goals and situations to be avoided. Express the goals as consistency and completeness checks on the object store: i.e., define what makes a configuration "consistent" and "complete".

   This model is called a *configuration model* in [16]. Note that, in practice, the different parts of the model would usually be defined in parallel. VDM-SL type and function definitions will be used in the body of this paper to formally specify the configuration model for the example application.

2. Define a process model which incorporates task management information as follows:
   (a) Determine the objects which are to be tracked and their possible statuses. Status values shoud be chosen to represent the important process attributes of the objects (e.g. 'HAS-CORRECT-SYNTAX' for the source code program example above). Note that it may not be practical – or even desirable – to track the status of every object in the system.
   (b) Define the *state* of the process as consisting of the object store itself together with the status information. Also define the possible initial states of the system (including initial statuses).
   (c) Specify any extra tools required by the process engine (e.g. for checking relationships or creating new objects).
   (d) Define the functionality of the UI in terms of operations (transitions between process states). Operations may seek input from the user or return output to the

user. Also include a definition of the conditions under which the operation may be invoked; this allows operations to be "disabled" in certain states.[1]

(e) State the *behavioural properties* which capture the intended meaning of status information in terms of the consistency and completeness checks defined earlier. These properties are typically statements of the form "if objects $x_1, \ldots, x_n$ have statuses $s_1, \ldots, s_n$ then property P holds" (sufficient conditions) or "if property P' holds then object x has status s" (necessary conditions) or some combination of these.[2]

In the example below we use VDM-SL state and operation definitions to formally specify the process model. Status information is modelled using mappings (finite partial functions) from objects to their status. State invariants record constraints on the object store, such as structural relationships that should always hold.

3. Verify that the process model is mathematically consistent and has the required properties:

   (a) Check that tool prerequisites have been met and that the specified transitions are mathematically feasible. In VDM these checks are called well formedness and satisfiability proof obligations: see [1, 12] for details.

   (b) Show that the behavioural properties hold in all "reachable" states of the system: i.e., they are true in the initial state(s) of the system and are preserved by all enabled operations.[3] The proof obligations are described in more detail in [16].

3 Example: a theory manager

3.1 Motivation and background

We illustrate the method on the design of an UI for a *Theory Manager* such as might be used in conjunction with a mechanical proof assistant to develop a collection of mathematical theorems and their proofs. The user will be provided with operations to add and delete proofs of theorems, together with status information for individual theorems in the theory and for the overall theory. Note that the proof of a theorem can use other theorems, and that the entire network of dependencies between theorems can become large and complex.[4] It is *crucial* to the logical soundness of the theory that circular reasoning be disallowed (cf. Fig. 1). The user's task is to develop the theory by giving proofs of its theorems. We shall not consider how such a task would be carried out by the human user (see instead e.g. [4]); rather, we assume the Theory Manager's role is to track dependencies and watch for circularities, and to present useful status information to the user. The user's overall goal is to create a "complete" theory – one in which all theorems have complete proofs, with no circularities.

[1] Low-level design decisions (such as how operations are disabled, what error messages are produced, etc) will not be treated here.

[2] Note that status information is "indicative" only: i.e., it may not be possible to characterize the meaning of a given status in terms of an equivalent "static" property of the underlying store of objects.

[3] We assume that all state-changing operations are specified in the process model. It is however possible to add new state-changing operations to the design without having to re-verify the process model *provided* preservation of the behavioural properties is added as a requirement for each new operation.

[4] For example, the `mural` support system for VDM [11] has several thousands of theorems, as does the `Ergo` proof assistant for `Sum` (the Z-like specification language of the `Cogito` methodology) [2].

`Theorem A:` $1 + 1 = 3$
`proof:`

1.	1+1+1=3	definition
2.	1=0	Theorem B
3.	1+1+0=3	substitution(1,2)
4.	1+1+0=1+1	Theorem C
5.	1+1=3	substitution(3,4)

`Theorem B:` $1 = 0$
`proof:`

1.	1+1+1=3	definition
2.	1+1=3	Theorem A
3.	1+1+1=1+1	substitution(1,2)
4.	1+1=1	Theorem D (3)
5.	1=0	Theorem E (4)

`Theorem C:` $\forall n \cdot n + 0 = n$
`Theorem D:` $\forall m, n \cdot m + 1 = n + 1 \;\Rightarrow\; m = n$
`Theorem E:` $\forall n \cdot n + 1 = 1 \;\Rightarrow\; n = 0$

Figure 1: An example of circular reasoning. Note that each proof is complete and correct when considered in isolation, but that Theorems A and B are mutually dependent.

The traditional approach to maintaining soundness – as exemplified by logic textbooks – is to linearly order the theorems and allow proofs to use only theorems which are proven earlier in the ordering (i.e., do not allow forward references). Automath [5] is an example of a proof assistant which enforces this style, and the highly influential LCF [8] uses a variant of this approach. In practice, however, such an approach is annoyingly inflexible, requiring that proofs be planned in detail on paper before taking them to the machine. The `mural` system [11] was developed in response to a perceived need for a more flexible style of working, whereby the user could develop a theory on-line in a piecemeal fashion, say interrupting the proof of one theorem to conjecture and prove useful lemmas, or to work on another proof in parallel. The work described here grew out of an attempt to show that the `mural` theory manager ensures logical soundness.[5]

3.2 UI requirements

The theory manager described here will support the storage and use of incomplete proofs and will even allow circular reasoning. The following task management information will be provided:

i. The UI will indicate when the theory is circular.

ii. If a circularity is present, the UI will give the user some idea of where it can be found.

iii. The UI will indicate when the theory is complete.

iv. The UI will indicate when a given theorem is fully established.

v. The UI will indicate which theorems have incomplete proofs.

We will state – and show how to prove – properties that relate status information to properties of the underlying theory store: e.g. we will show that if the theory attains the status COMPLETE then the theory really is complete.

Such a system might be used for example to track the completeness of the formal verification of a user application – for example, to check that proof obligations have been completely and correctly discharged in a Z or VDM development. Although in this paper we restrict our attention to theorems and proofs, it is easy to imagine how the example could be extended

[5]The `mural` theory manager is similar to, but different from, the one presented here.

to support formal verification more generally. In [16] the method is applied to a formal development support environment to track the co-development of a formal specification and its verification.

4 Modelling the application domain

In this section we describe and formally model the Theory Manager application domain.

4.1 The objects to be managed

We shall be concerned with three kinds of object:

- **Theorems**, which are (named) mathematical statements expressing properties which are believed to hold. For simplicity of modelling we shall not distinguish between axioms, lemmas, conjectures, postulates, proof obligations and so on – they will all simply be called theorems.

  Theorems will be modelled here using the not-further-defined type *Thm*.

- Formal (machine-checkable) **proofs**, which are structures which – if correct and complete – establish the truth of theorems. Different proof assistants use widely differing forms of proof structures, but we shall not concern ourselves here with the differences: instead, we shall simply assume there is a tool which extracts from a proof the set of theorems used in the proof. Note also that, as explained above, we wish to support the storage and use of partial and incomplete proofs; for simplicity of modelling we shall thus use the term 'proof' to cover both complete and incomplete proofs.

  Proofs will be modelled here using the not-further-defined type *Proof*.

- A **theory** which is a collection of theorems together with their proofs.

  Theories will be modelled as a mapping (i.e., a finite partial function) type from theorems to their proofs:

  $$Theory = Thm \xrightarrow{m} Proof$$

4.2 Basic tools required

As explained above, we have chosen to model the application at the level of granularity of theorems and proofs, without going into details of their internal structure. We simply assume instead that there are tools available for checking whether a given proof is **finished** (i.e., is correct and complete) and for extracting the set of theorems used in a given proof. These two tools will be modelled formally as not-further-defined functions with signatures as follows:[6]

$$\textit{is-finished} : \textit{Proof} \rightarrow \mathbb{B}$$

$$\textit{uses} : \textit{Proof} \rightarrow \textit{Thm}\text{-}\mathsf{set}$$

We shall assume both tools are reasonably efficient. In practice, the efficiency of these tools depends on whether the appropriate hooks are provided by the target proof assistant. For

[6]Note that proofs may use theorems that do not yet have proofs: i.e. for any given theory T there may be a proof p in $\mathsf{rng}\ T$ such that $\neg\,(uses(p) \subseteq \mathsf{dom}\ T)$.

systems which record proof structures explicitly, such as `mural` [11] and Ergo [18], these tools could be very fast. In other cases an alternative design strategy may be needed (e.g. cache the results and store them with the proof at the time the proof is constructed).

4.3 Consistency and completeness checks

In this section we describe the basic properties to be tracked by the Theory Manager. Note that although it would be possible to build tools that evaluate these properties directly, such tools would be woefully slow, especially when the theory store grows in size and complexity; instead, our system will use knowledge of the process model to provide an efficient means to track and manage the properties.

We say a theorem s **depends directly on** a theorem t in theory T if s has a proof in T and the proof uses t; formally: $s \in \mathsf{dom}\ T \wedge t \in uses(T(s))$.

A **reference chain** is a sequence of theorems, each of which depends directly on the next theorem in the chain:

$$
\begin{array}{l}
\textit{is-reference-chain} : \textit{Thm}^* \times \textit{Theory} \to \mathbb{B} \\
\textit{is-reference-chain}\,(ts, T) \triangleq \\
\quad \forall\, i \in \{1, \ldots, \mathsf{len}\ ts - 1\} \cdot ts(i) \in \mathsf{dom}\ T \wedge ts(i{+}1) \in uses(T(ts(i)))
\end{array}
$$

We say theorem s **depends on** t in T if there is a reference chain from s leading back to t:

$$
\begin{array}{l}
\textit{depends-on} : \textit{Thm} \times \textit{Thm} \times \textit{Theory} \to \mathbb{B} \\
\textit{depends-on}\,(s, t, T) \triangleq \\
\quad \exists\, ts : \textit{Thm}^* \cdot \mathsf{len}\ ts \geq 2 \wedge \textit{is-reference-chain}(ts, T) \wedge ts(1) = s \wedge ts(\mathsf{len}\ ts) = t
\end{array}
$$

We say a theorem t has a **circular proof** if it depends – directly or indirectly – on itself.

A **reference loop** is a reference chain which starts and ends at the same theorem:

$$
\begin{array}{l}
\textit{is-reference-loop} : \textit{Thm}^* \times \textit{Theory} \to \mathbb{B} \\
\textit{is-reference-loop}\,(ts, T) \triangleq \\
\quad \textit{is-reference-chain}(ts, T) \wedge ts(1) = ts(\mathsf{len}\ ts)
\end{array}
$$

A theory is said to **have a circularity** if it contains any theorems with circular proofs:

$$
\begin{array}{l}
\textit{has-circularity} : \textit{Theory} \to \mathbb{B} \\
\textit{has-circularity}\,(T) \triangleq \\
\quad \exists\, t \in \mathsf{dom}\ T \cdot \textit{depends-on}(t, t, T)
\end{array}
$$

A theory is said to be **complete** if it has no circularities, all of its proofs are correct and complete, and the theory is self-contained (i.e., all theorems used in its proofs are themselves proven in the theory):

$$
\begin{array}{l}
\textit{is-complete} : \textit{Theory} \to \mathbb{B} \\
\textit{is-complete}\,(T) \triangleq \\
\quad \neg\, \textit{has-circularity}(T) \wedge \forall\, p \in \mathsf{rng}\ T \cdot \textit{is-finished}(p) \wedge uses(p) \subseteq \mathsf{dom}\ T
\end{array}
$$

The overall goal of theory management is to develop a complete theory; in practice however it might not be realistic to expect to complete every proof in a theory, and some compromise might have to be reached whereby full proofs are given only for certain selected theorems (e.g. safety properties). A theorem is said to be **fully established** in a given theory if it and every theorem on which it depends (directly and indirectly) has a complete proof:

$$\textit{is-fully-established} : \textit{Thm} \times \textit{Theory} \to \mathbb{B}$$

$$\textit{is-fully-established}\,(t, T) \triangleq$$
$$t \in \mathsf{dom}\ T \wedge \textit{is-finished}(T(t)) \wedge \forall\, s \in \textit{uses}(T(t)) \cdot \textit{is-fully-established}(s, T)$$

4.4 Some useful lemmas

The following lemmas are logical consequences of the definitions above which are useful in the verification of the process model.

Lemma 1. A theory is complete if and only if it has no circularities and all of its theorems are fully established.

$$\forall\, T : \textit{Theory} \cdot \textit{is-complete}(T) \Leftrightarrow \neg\, \textit{has-circularity}(T) \wedge \forall\, t \in \mathsf{dom}\ T \cdot \textit{is-fully-established}(t, T)$$

Lemma 2. A theorem t_0 has a circular proof if and only if some reference loop involves t_0.

$$\forall\, t_0 : \textit{Thm}, T : \textit{Theory} \cdot \textit{depends-on}(t_0, t_0, T) \Leftrightarrow \exists\, ts : \textit{Thm}^* \cdot \textit{is-reference-loop}(ts, T) \wedge t_0 \in \mathsf{elems}\ ts$$

Lemma 3. When a theory T is extended with a proof p_0 of a theorem t_0, the new theory has a circularity if and only if either (a) there was already a circularity present in T or (b) p_0 uses theorems that depend on t_0 in T.

$$\forall\, t_0 : \textit{Thm}, p_0 : \textit{Proof}, T : \textit{Theory} \cdot t_0 \notin \mathsf{dom}\ T \Rightarrow$$
$$\textit{has-circularity}(T \dagger \{t_0 \mapsto p_0\}) \Leftrightarrow \textit{has-circularity}(T) \vee \exists\, t \in \textit{uses}(p_0) \cdot \textit{depends-on}(t, t_0, T)$$

5 The process model

5.1 Tracking the status of objects

The process model will track the status of individual theorems in the theory as well as the overall status of the theory. We shall not track the status of individual proofs since their status can be deduced from their corresponding theorem.

The **overall theory status** will be one of the following:

HAS-CIRCULARITY – indicates that the theory may be circular

INCOMPLETE – indicates that one or more theorems may still require proof

COMPLETE – indicates that the theory is complete

This is modelled by the following type definition:

$$\textit{TheoryStatus} = \{\text{HAS-CIRCULARITY}, \text{INCOMPLETE}, \text{COMPLETE}\}$$

A **theorem's status** will be one of the following:

CIRCULAR – indicates the theorem may have a circular proof

UNPROVEN – indicates the theorem's proof is not yet finished

PROVEN – indicates the theorem's proof is finished but the theorem may not yet be fully established

ESTABLISHED – indicates the theorem is fully established

This is modelled by the following type definition:

$$\mathit{ThmStatus} = \{\text{CIRCULAR}, \text{UNPROVEN}, \text{PROVEN}, \text{ESTABLISHED}\}$$

Note that the status is merely indicative in some cases: e.g. the theory may have status INCOMPLETE and yet actually be complete.[7] Section 5.5 below explains how status information relates to properties of the underlying theory store.

5.2 The process state

We define the state of the process model to consist of

- the current value of the theory,
- an assignment of statuses to theorems which have proofs in the theory, and
- the theory's current overall status.

Initially the theory is empty and has status COMPLETE.[8]

```
state $\mathit{TheoryManager}$ of
  $\mathit{theory} : \mathit{Theory}$,
  $\mathit{status} : \mathit{Thm} \xrightarrow{m} \mathit{ThmStatus}$,
  $\mathit{overall} : \mathit{TheoryStatus}$
  inv $T, m, \sigma \triangleq \mathsf{dom}\ m = \mathsf{dom}\ T$
  init $\mathit{mk\text{-}TheoryManager}(T, m, \sigma) \triangleq T = \{\mapsto\} \wedge \sigma = \text{COMPLETE}$
end
```

5.3 Tools used by the process engine

This section describes some of the tools that will be used by the process engine. These tools extend the functionality offered by the basic tools described in Section 4.2. Since this paper describes a high-level design, the tools are simply specified here. Note that simple tools for accessing and manipulating structures (e.g. to check $t \in \mathsf{dom}\ T$ or to construct $T \dagger \{t_0 \mapsto p_0\}$) will not be specified separately.

The first of the tools is for checking the *depends-on* relationship: i.e. given theorems s and t and theory T, determine whether or not *depends-on*(s, t, T). The definition is given in Section 4.3 above. A variety of different techniques could be used to implement such a tool (e.g. via a graph-searching algorithm). Because dependencies may reach far back into a theory, however, any such tool is likely to be quite slow, especially when the theory is large and complex. As a result, we shall make sparing use of this tool.

[7] Although it is possible in theory to design a process model in which a theory's status is COMPLETE if and only if the theory actually is complete, such a process model would require expensive checks to be run after each operation, resulting in an UI with unacceptable performance characteristics.

[8] In practice, the Theory Manager may be incorporated into a larger system in which the initial theory is non-empty. In such a case it would be necessary to initialize the statuses appropriately, after checking each theorem individually.

A second tool is for checking whether or not a given theorem is used in any of the proofs in a theory T:

$$\begin{array}{l} \textit{has-user} : \textit{Thm} \times \textit{Theory} \to \mathbb{B} \\ \textit{has-user}\,(t, T) \triangleq \\ \quad \exists\, p \in \mathsf{rng}\ \textit{theory} \cdot t \in \textit{uses}(p) \end{array}$$

A third tool uses theorem status information to perform a quick check on whether a theorem t is fully established. It does this by checking that all of the theorems used in the proof of t have status PROVEN or ESTABLISHED; for those with status PROVEN the check is then applied recursively:[9]

$$\begin{array}{l} \textit{check-if-estab} : \textit{Thm} \times \textit{Theory} \times (\textit{Thm} \xrightarrow{m} \textit{ThmStatus}) \to \mathbb{B} \\ \textit{check-if-estab}\,(t, T, m) \triangleq \\ \quad \forall\, s \in \textit{uses}(T(t)) \cdot \\ \qquad s \in \mathsf{dom}\ T \cap \mathsf{dom}\ m \wedge \\ \qquad (m(s) = \text{ESTABLISHED} \vee (m(s) = \text{PROVEN} \wedge \textit{check-if-estab}(s, T, m))) \\ \mathsf{pre}\ \ t \in \mathsf{dom}\ T \end{array}$$

It will follow from the properties of the process model that if the check evaluates to true then t is fully established in T.

Note that the specification of *check-if-estab* includes a precondition, which expresses the prerequisite that the tool should only ever be applied to theorems t in T. (The prerequisite is included here mainly for illustrative purposes and is perhaps a little artificial.)

5.4 UI operations

The high-level design of the Theory Manager UI has four core operations:

1. $\textit{GiveProof}(t_0, p_0)$ – adds theorem t_0 with proof p_0 to the theory store (assuming the theorem does not already have a proof);
2. $\textit{DeleteProof}(t_0)$ – deletes the proof of theorem t_0 from the theory;
3. $\textit{CheckProof}(t_0)$ – runs a "deep check" on theorem t_0 to see if it is fully established;
4. *CheckTheory* – updates the theory's overall status.

Later stages in the design – which are outside the scope of the present paper – would need to address questions such as how to ensure in $\textit{GiveProof}(t_0, p_0)$ that t_0 does not already have a proof, and how to present to the user details of which theorems depend (or don't depend) on a given theorem. For the purposes of this paper, however, we assume that the above four operations are the only operations that can directly change the process state.

Figures 2–5 give formal specifications of the four core operations. The operations are explained informally below:

[9] Note that the definition of *check-if-estab* given here is a specification only; an implementation would need to ensure termination (e.g. in the presence of circularities).

GiveProof(t_0, p_0)

The status s_0 of the new theorem t_0 is calculated as follows:

- Check if anything in p_0 already depends on t_0 – if so, t_0 will have a circular proof, so s_0 is set to CIRCULAR;
- otherwise, check if p_0 is correct and complete – if not, s_0 is set to UNPROVEN;
- check whether all theorems used in p_0 have status ESTABLISHED – if so, set s_0 to ESTABLISHED; otherwise set s_0 to PROVEN.

The overall status of the theory is set as follows:

- If s_0 is CIRCULAR, then the overall status is set to HAS-CIRCULARITY;
- If it was COMPLETE and s_0 is UNPROVEN or PROVEN, then it is set to INCOMPLETE;
- otherwise it is left unchanged.

```
$GiveProof\ (t_0 : Thm, p_0 : Proof)$
ext wr $theory : Theory, status : Thm \xrightarrow{m} ThmStatus, overall : TheoryStatus$
pre  $t_0 \notin$ dom $theory$
post let $s_0 =$  if $\exists\, t \in uses(p_0) \cdot depends\text{-}on(t, t_0, \overleftarrow{theory})$
                       then CIRCULAR
                  elseif $\neg\ is\text{-}finished(p_0)$ then UNPROVEN
                  elseif $\forall\, t \in uses(p_0) \cdot t \in$ dom $\overleftarrow{theory} \wedge \overleftarrow{status}(t) =$ ESTABLISHED
                       then ESTABLISHED
                  else PROVEN
     in
     $theory = \overleftarrow{theory} \dagger \{t_0 \mapsto p_0\}$
     $\wedge\ status = \overleftarrow{status} \dagger \{t_0 \mapsto s_0\}$
     $\wedge\ overall =$  if $\overleftarrow{overall} =$ HAS-CIRCULARITY $\vee\ s_0 =$ CIRCULAR
                        then HAS-CIRCULARITY
                        else if $\overleftarrow{overall} =$ COMPLETE $\wedge\ s_0 =$ ESTABLISHED
                             then COMPLETE
                             else INCOMPLETE
```

Figure 2: Operation for adding the proof of a theorem to the theory.

The strategy behind this design is as follows:

- It is important to flag any circularity as soon as it arises, so the user can see immediately that something is wrong; the user can thus be expected to be willing to wait for a full circularity check of the new proof. (Note however that a circularity check of the full theory is *not* required.) Rather than disallow the proof if it causes a circularity, we prefer to give the user the opportunity to break the circularity somewhere else in the loop, by changing one of the other theorem's proofs.

- On the other hand, when it comes to checking whether a theorem is fully established, the thoroughness of the check should be under user control. Since a full check would be very time-consuming, *GiveProof* simply does a shallow check (of the statuses of the theorems used in the proof) and leaves the deep check to *CheckTheorem*. Note that the shallow check is conservative in what it asserts: e.g. it asserts that t_0 has status ESTABLISHED only if all theorems used in the proof have status ESTABLISHED.

Note also that the theory may actually become complete as a result of adding the new theorem and proof (e.g. because it completes the proof of theorems that were not fully established before), but before the COMPLETE status is achieved it may be necessary for the user to apply the *CheckProof* and *CheckTheory* operations a number of times.

DeleteProof(t_0)

When the proof of theorem t_0 is deleted, the statuses of theorems which depend on t_0 are updated from ESTABLISHED to PROVEN. (Note that this operation should be applied cautiously, especially if many theorems depend on t_0.) The overall status of the theory changes from COMPLETE to INCOMPLETE if t_0 was used in any proofs in the theory.

```
$DeleteProof\ (t_0 : Thm)$
ext wr $theory : Theory,\ status : Thm \xrightarrow{m} ThmStatus,\ overall : TheoryStatus$
pre  $t_0 \in \mathrm{dom}\ theory$
post $theory = \{t_0\} \lhd \overleftarrow{theory}$
    $\wedge\ \forall\ t \in \mathrm{dom}\ theory \cdot$
        $status(t) =$ if $\overleftarrow{status}(t) = \text{ESTABLISHED} \wedge depends\text{-}on(t, t_0, theory)$
                      then PROVEN
                      else $\overleftarrow{status}(t)$
    $\wedge\ overall =$ if $\overleftarrow{overall} = \text{COMPLETE} \wedge has\text{-}user(t_0, theory)$
                   then INCOMPLETE
                   else $\overleftarrow{overall}$
```

Figure 3: Operation for deleting the proof of a theorem.

CheckProof(t_0)

Recall that the status of a theorem is merely indicative and does not necessarily represent the "best" status the theorem could have. For example, a theorem t_0 may have received status CIRCULAR because it caused a circularity when it was first added but since then the circularity may have been broken by deleting one of the other theorems in the dependency loop. Similarly, a theorem may have received status PROVEN rather than ESTABLISHED because one or more of the theorems on which it depends did not have a proof, but the missing proofs may have been supplied subsequently. The *CheckProof* operation uses the *check-if-estab* tool described in Section 5.3 to update the theorem's status, using information about the status of the theorems on which it depends.

```
    $CheckProof\ (t_0 : Thm)$
    ext rd $theory : Theory,$
        wr $status : Thm \xrightarrow{m} ThmStatus$
    pre  $t_0 \in$ dom $theory$
    post let $p_0 = theory(t_0),$
             $check1 =$  if $is\text{-}finished(p_0)$
                         then $check2$
                         else UNPROVEN,
             $check2 =$  if $check\text{-}if\text{-}estab(t_0, theory, \overleftarrow{status})$
                         then ESTABLISHED
                         else PROVEN,
             $s_0$                                                     $=$
cases $\overleftarrow{status}(t_0)$:
CIRCULAR $\to$ if $depends\text{-}on(t_0, t_0, theory)$ then CIRCULAR else $check1$
UNPROVEN $\to$ UNPROVEN
PROVEN $\to$ $check2$
ESTABLISHED $\to$ ESTABLISHED
          in $status = \overleftarrow{status} \dagger \{t_0 \mapsto s_0\}$
```

Figure 4: Operation for checking a theorem

CheckTheory

Finally, *CheckTheory* updates the theory's overall status by checking the statuses of the theorems in the theory.

```
    $CheckTheory\ ()$
    ext rd $status : Thm \xrightarrow{m} ThmStatus,$
        wr $overall : TheoryStatus$
    post $overall =$  if $\overleftarrow{overall}$ = HAS-CIRCULARITY $\wedge$ CIRCULAR $\in$ rng $status$
                      then HAS-CIRCULARITY
                      else if $\overleftarrow{overall}$ = COMPLETE $\vee$ rng $status$ = {ESTABLISHED}
                          then COMPLETE
                          else INCOMPLETE
```

Figure 5: Operation for checking a theory

5.5 The behavioural properties

The following properties formalize the relationship between object statuses and "static" properties of the object store. In any reachable state $mk\text{-}TheoryManager(T, m, \sigma)$ of the Theory Manager the following properties hold:

1. $\sigma =$ COMPLETE $\Rightarrow$ $is\text{-}complete(T)$
2. $has\text{-}circularity(T)$ $\Rightarrow$ $\sigma =$ HAS-CIRCULARITY

3. $\forall\, ts : Thm^* \cdot \textit{is-reference-loop}(ts, T) \;\Rightarrow\; \exists\, t \in \mathsf{elems}\; ts \cdot m(t) = \textsc{circular}$

4. $\forall\, t \in \mathsf{dom}\; T \cdot m(t) = \textsc{established} \;\Rightarrow\; \textit{is-fully-established}(t, T)$

5. $\forall\, t \in \mathsf{dom}\; T \cdot \textit{is-finished}(T(t)) \wedge \textit{check-if-estab}(t, T, m) \;\Rightarrow\; \textit{is-fully-established}(t, T)$

6. $\forall\, t \in \mathsf{dom}\; T \cdot m(t) = \textsc{proven} \;\Rightarrow\; \textit{is-finished}(T(t))$

7. $\forall\, t \in \mathsf{dom}\; T \cdot m(t) = \textsc{unproven} \;\Rightarrow\; \neg\, \textit{is-finished}(T(t))$

From the above properties we can deduce the following relationship between status information and task management information described in Section 3.2:

i. If the theory is circular, it will have status HAS-CIRCULARITY.

ii. The theory's status will change to HAS-CIRCULARITY as soon as a proof is added which results in a circularity. (The status will remain HAS-CIRCULARITY until no theorem has status CIRCULAR and the *CheckTheory* operation has been invoked.)

   Additionally, if a theorem has status CIRCULAR and the status does not change upon applying the *CheckProof* operation, the theorem has a circular proof. (Note that a theorem may have a circular proof without necessarily having status CIRCULAR.)

iii. The theory is complete if the status COMPLETE is attained.

iv. A theorem with status ESTABLISHED is guaranteed to be fully established. The converse does not necessarilly hold, however, and it may be necessary for the user to apply the *CheckProof* operation to the theorem or some of the theorems it uses before the system will recognize that the theorem is in fact fully established.

v. The theorems with incomplete proofs are those with status UNPROVEN. Note that proofs may also use theorems which do not themselves have proofs and thus do not have a status; such theorems can be tracked down by looking at the proofs of theorems which have status PROVEN but whose status does not change upon applying the *CheckProof* operation.

6 Verification of the process model

This section illustrates the steps involved in the verification of the process model given in Section 5. The proofs are too lengthy to give here in full.

6.1 Well-formedness and satisfiability of the model

To show that the model is mathematically meaningful it is necessary to check that partial functions are applied to arguments within their domains, and that operations are mathematically feasible, in the sense that their postconditions are satisfiable (can be achieved). As an example of the former, consider the term $\overleftarrow{status}(t)$ in the postcondition of *DeleteProof* (Fig. 3): it is well-formed in context since $t \in \mathsf{dom}\; \overleftarrow{theory}$ and $\mathsf{dom}\; \overleftarrow{theory} = \mathsf{dom}\; \overleftarrow{status}$ (from the state invariant). The proofs of the other cases are similarly straightforward. Regarding satisfiability, there is little to prove in this case since the operation definitions are given constructively; we need only check that the postconditions of the four operations are consistent with the state invariant,[10] which is straightforward.

[10]According to the semantics of VDM, the state invariant is implicitly part of the pre- and post-conditions of all operations.

6.2 The behavioural properties

We must show that the behavioural properties in Section 5.5 are true in the initial state of the system and are preserved by all enabled operations. From the initialization condition we know that $T = \{\mapsto\}$ and hence that $\neg\, \mathit{has\text{-}circularity}(T)$, $\mathit{is\text{-}complete}(T)$ and $\mathsf{dom}\ T = \{\,\}$ initially; it is easy to check that all 7 properties are true under these conditions.

We show below that Property 1 is preserved by the *GiveProof* operation (Fig. 2). The proofs for the other operations, and the proofs of the other properties, are similar (but generally easier).

Preservation of property 1 under $\mathit{GiveProof}(t_0, p_0)$

Let $\tau = \mathit{mk\text{-}TheoryManager}(T, m, \sigma)$ be the state in which the operation is invoked and $\tau' = \mathit{mk\text{-}TheoryManager}(T', m', \sigma')$ the state immediately after. It follows that τ and τ' are related by the postcondition of $\mathit{GiveProof}(t_0, p_0)$. Suppose also that τ satisfies the behavioural properties (the "Induction Hypothesis") and that the precondition of $\mathit{GiveProof}(t_0, p_0)$ is satisfied (i.e., $t_0 \notin \mathsf{dom}\ T$); we are required to prove that τ' satisfies Property 1.

To do this, we suppose that $\sigma' = \text{COMPLETE}$ and show that $\mathit{is\text{-}complete}(T')$. From the postcondition we know

$$T' = T \dagger \{t_0 \mapsto p_0\},\ \sigma = \text{PROVEN},\ s_0 = \text{ESTABLISHED}$$

It follows from $s_0 = \text{ESTABLISHED}$ and the postcondition that

- the theorems used by p_0 do not depend on t_0 in T;
- p_0 is a correct and complete proof; and
- every theorem used in p_0 has a proof in the theory and has status ESTABLISHED.

From $\sigma = \text{PROVEN}$ and the induction hypothesis we know that T must be complete and have no circularities. From Lemma 3 in Section 4.4 and the facts above, it follows that T' also has no circularities. Finally, since $\mathsf{rng}\ T' = \{p_0\} \cup \mathsf{rng}\ T$ and T is complete, it follows from the facts above and the definition of *is-complete* that T' is also complete, as we were required to prove. □

7 Relationship to other work

This work extends work done by others on formally modelling aspects of UI design (see [9] for a survey) by providing a proof method for verifying certain properties of UIs.

The method uses a state-based approach to the formal specification of interactive systems. As has been argued elsewhere (e.g. [6, 17]), a state-based description on its own is generally inadequate or overly awkward for modelling interactive systems, and there are times when a trace-based description (in a language such as CSP or LOTOS) is more appropriate. While agreeing with the point in general, we believe that the state-based approach is ideal for "data driven" applications such as the one described above, and contend that it would be very difficult to give as succinct a definition of the Theory Manager in a process-algebraic notation as we have given here.

The method is well suited to the specification of interactive systems in which task goals can be expressed in terms of consistency and completeness conditions on the underlying configuration of objects, especially those systems in which the user takes primary responsibility for planning and carrying out the tasks. For a description of more general methods for analysing and modelling tasks, however, the reader is referred elsewhere (e.g. chapters 11-14 of [10]).

This paper has been concerned with two aspects of usability: support for flexible, non-proscriptive work patterns, and provision of useful, accurate task management information. Other dimensions of usability not touched on here include tolerance to human errors [19] and demands on the user's memory [7].

8 Conclusions

This paper has described a method for formally specifying and reasoning about aspects of UI design to do with process control and provision of task management information. The method is well suited to "data intensive" applications in which the system is being used to develop complex "configurations" of interconnected objects, and for which task goals can be expressed in terms of properties of the underlying configuration of objects. The method includes proof obligations to check the accuracy of the task management information. The method could be used in conjunction with process modelling techniques, such as the use of state-charts as an intermediate design notation and the use of process engines provided by a variety of process modelling languages. Similarly, it could be used in conjunction with more general methods for UI design such as interactors [6].

The 'Theory Manager' application is a small but succinct example of how to apply the general method described in the paper, and illustrates a number of interesting aspects of formal modelling and verification of UIs. It is also an interesting application in itself, being to this author's knowledge the first documented design of a theory management system that ensures logical soundness of the underlying theory store while allowing incomplete proofs and the possibility of circular reasoning.

The ability to support and recover from "inconsistent" states, such as presence of circular reasoning in this case, can be vital to the usability of a system. The alternative strategy – of not allowing the system to enter inconsistent states – can force the user into highly convoluted and potentially confusing patterns of use, which can be tedious or even prohibitively complicated; such systems tend to be highly unpopular with users.

Finally, we note that the techniques described in this paper can be applied at arbitrary levels of granularity: e.g. if access to the internal structure of proofs is available then we could track dependencies within proofs and support editing of individual lines within proofs (cf. Section 4.7 of [11]). Fine-grained process models can give better support for change management, for example, by limiting the extent to which changes need to be percolated through a design [15]. In principle, the level of granularity is determined simply by what basic tools are available. In practice, however, the finer the level of granularity, the more complicated is the design and verification of the process model.

Acknowledgements: The author gratefully acknowledges the collaboration of Kelvin Ross in the original development of the method and for many subsequent discussions. We are also grateful for useful discussions with Burkhard Peuschel in relation to process modelling.

References

[1] J. C. Bicarregui, J. S. Fitzgerald, P. A. Lindsay, R. Moore, and B. Ritchie. *Proof in VDM: A Practitioner's Guide.* FACIT Series. Springer-Verlag, 1994. ISBN no. 3-540-19813-X.

[2] A. Bloesch. The standard Ergo theories. Technical Report 95-43, Software Verification Research Centre, The University of Queensland, 1995.

[3] British Standards Institute, Working Group IST/5/19. *VDM Specification Language Draft International Standard*, 1995.

[4] A. Bundy. A science of reasoning. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 178–198. MIT Press, 1991.

[5] N.G. de Bruijn. A survey of the project Automath. In Seldin and Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 579–606. Academic Press, 1980.

[6] D. Duke and M. Harrison. Abstract interaction objects. *Computer Graphics Forum*, (3):25–36, 1993.

[7] B. Fields, M. Harrison, and P. Wright. Modelling interactive systems and providing task relevant information. In F. Paterno, editor, *Proc. 1st Eurographics Workshop on Design, Specification and Verification of Interactive Systems*, Eurographics Seminar Series, pages 131–146, 1994.

[8] M.J. Gordon, R. Milner, and C.P. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science.* Springer-Verlag, 1979.

[9] M. Harrison and D. Duke. A review of formalisms for describing interactive behaviour. In *Software Engineering and Human-Computer Interaction*, pages 49–75. Springer Verlag, 1995. Proc. ICSE'94 Workshop on SE-HCI.

[10] P. Johnson. *Human Computer Interaction: Psychology, Task Analysis and Software Engineering.* McGraw-Hill, 1992.

[11] C. B. Jones, K. D. Jones, P. A. Lindsay, and R. Moore. *mural: A Formal Development Support System.* Springer-Verlag, 1991.

[12] C.B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall International, second edition, 1990.

[13] G.E. Kaiser and P.H. Feiler. An architecture for intelligent assistance in software development. In *Proc. 9th Int. Conf. on Software Engineering*, pages 180–188. IEEE Computer Society Press, 1987.

[14] B. Peuschel, W. Schäfer, and S. Wolf. A knowledge-based software development environment supporting cooperative work. *Int. J. of Software Eng. and Knowledge Eng.*, 2:79–106, 1992.

[15] K.J. Ross and P.A. Lindsay. Maintaining consistency under changes to formal specifications. In *FME'93: Industrial Strength Formal Methods.* Springer Verlag, 1993. Proc. First Internat. Symp. of Formal Methods Europe, Odense, Denmark, April 1993.

[16] K.J. Ross and P.A. Lindsay. A precise examination of the behaviour of process models. In *FME'94: Industrial Benefits of Formal Methods*, pages 251–270. Springer Verlag, 1994. 2nd International Symposium of Formal Methods Europe, Barcelona, October 1994.

[17] B. Sufrin and J. He. Specification, analysis and refinement of interactive processes. In M. Harrison and H. Thimbleby, editors, *Formal Methods in Human-Computer Interaction*, pages 153–200. Cambridge University Press, 1990.

[18] Mark Utting and Keith Whitwell. Ergo user manual. Technical Report 93-19, Software Verification Research Centre, Department of Computer Science, The University of Queensland, St. Lucia, QLD 4072, Australia, March 1994. Describes Version 4.0 of Ergo.

[19] P. Wright, B. Fields, and M. Harrison. Deriving human-error tolerance requirements from tasks. In *Proc. 1st Int. Conf. on Requirements Engineering*, pages 135–142. IEEE, 1994.