SOFTWARE VERIFICATION RESEARCH CENTRE

SCHOOL OF INFORMATION TECHNOLOGY

THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 97-23

A Formal Approach to Specification and Verification of Task Management in Interactive Systems

Peter A. Lindsay

April 1997

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

Note: Most SVRC technical reports are available via anonymous ftp, from `svrc.it.uq.edu.au` in the directory `/pub/techreports`. Individual abstracts and compressed postscript files are available from `http://svrc.it.uq.edu.au/ www/tr/list1.cgi?`

A Formal Approach to Specification and Verification of Task Management in Interactive Systems

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

To appear: *IEE Proceedings Software Engineering*, 1997.

Abstract

This paper presents an approach to formal specification of task management models for interactive systems. The approach is well suited to data-intensive applications in which the system is being used to manage complex collections of interrelated objects.

The approach consists of annotating objects with status information, and relating status back to properties of the underlying collection. Status information is used to guide and control the application of activities. The paper illustrates the approach on an example from interactive theorem proving.

1 Introduction

1.1 Motivation

As software users become more sophisticated, they increasingly rely on the software system to manage the more routine aspects of their tasks, so they can devote more energy to being creative. They expect the system to provide guidance towards performing their tasks, but they also expect the system to be flexible in how it allows them to go about their tasks.

As tasks become more complex, users rely more and more on the task-management information provided by the software. In some cases – such as in safety protection systems – the required user-response time may be so short, for example, that it is infeasible to check by hand that all tasks have been carried out consistently and completely. In other cases, manual checks may be altogether infeasible, because of the complexity or tedium involved. In all such cases, it becomes necessary to place high degrees of trust in the system's task-management capabilities and in the accuracy of the task status information provided by the system. In particular, it is vital that users have a clear idea of what can be inferred from the information presented to them by the system.

The design of task management aspects of the User Interface (UI) is critical in such situations. For critical applications where high levels of integrity are required, Standards increasingly recommend the use of formal methods [1] but give little guidance on how the methods should be applied to User Interfaces. This paper attempts to remedy this to some extent, by describing a formal approach to modelling and reasoning about task management for interactive systems, focussing on the meaning and accuracy of task status information.

There are many important aspects of task management which do not come within the scope of our approach however, such as the problem of analysing and modelling tasks (see e.g. chapters 11-14 of [2]), designing for tolerance of human errors [3], and design with respect to human limitations [4], to name just three.

1.2 The approach

The approach to task model specification taken here is to first model the underlying "state" of the interactive system as a collection of objects and relationships between objects. A number of key objects are then chosen and assigned a set of possible "statuses", to indicate their status with respect to achievement of task goals. User activities are then specified by defining how they change the state of the system and how the status of key objects are affected. Task goals are defined in terms of desired (target) statuses of key objects. Finally, task control can be imposed, if desired, by restricting users' access to functionality in certain states, using status information as a guide.

We claim that this is a natural approach to *specifying* task management for interactive systems. It concentrates very much on what is to be implemented, and leaves it to the system designer to determine how best to implement it. This claim is illustrated on a case study below.

The approach, which is described in more detail below, is primarily applicable to "data-driven" applications in which the system is used to construct and maintain large or complex "object stores" (consisting of objects and relationships between objects) and task goals are expressed in terms of consistency and completeness of the object store. Programming environments, Software Engineering Environments (SEEs), hypermedia networks, interactive theorem provers, and airline reservation systems are just a few examples of such systems. In a SEE with fine-grained traceability capabilities, for example, the objects could include individual program units, consistency conditions could include checks that correct syntax has been used, and completeness conditions could include checks that all variables have been initialized and all paths have been tested. The user would expect the SEE to report the status of individual units accurately, and to provide guidance for bringing a program suite into a consistent and complete state.

An important aspect of UI design for such systems is the provision of controlled – but *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, such as syntax-directed editors, can be annoyingly inflexible to use. Our approach aims to support

the design of flexible UIs, with accurately specified and verified behaviour.

1.3 Relationship to other work

The approach to task management described here is inspired in part by the "state-based" approach to process modelling [5, 6]. State-based process models are ones in which certain objects are assigned a state (or *status* as we prefer to call it here, 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, a process engine invokes tools and changes objects and their statuses accordingly. Process modelling languages such as Merlin and Marvel provide notations in which designers express their process models, and interpreters which construct the corresponding process engines.

The approach to specification described here is to use a state-based (sometimes called model-based [7]) notation. As such, the overall approach can be used in conjunction with more general methods for modelling interactive systems, such as interactors [8] for example. This work is however significantly different to other work on formally modelling aspects of UI design [9] in that it uses two levels of specification and considers verification of one level against the other.

It has been argued elsewhere [8, 10] that 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 this is undoubtedly true in many cases, we shall argue that the state-based approach is ideal for data-driven applications such as the ones described above. We contend that it would be difficult – if not impossible – to achieve the same aims using simple process-algebraic notations.

1.4 This paper

The approach is described in detail on an example application below, concerning the design of a Task Manager for a (hypothetical) interactive theorem prover. The case study described here grew out of an attempt to show that the `mural` proof support environment [11] is logically sound, but the amount of detail has been reduced substantially, to highlight the core issues. An earlier version of the approach was described in [12].

Section 2 describes the case study and its task management requirements. Section 3 gives formal definitions of the underlying concepts, and defines the consistency and completeness conditions that are of interest. VDM-SL notation [13, 14] is used for formal definitions here, although other state-based formal specification notations could be used equally as well.

The Task Manager is defined in two layers:

1. Section 4 gives a formal specification of the Task Manager in terms of the task status information which it will make available, and what can be inferred

from such information. Task goals are defined in terms of what statuses are desirable (positive goals) and which are undesirable (negative goals).

2. Section 5 gives a formal specification of the design of the Task Manager in terms of individual user actions and how they affect task status.

From the user's viewpoint, the first specification (the "goal model") defines the goals of the task and the second specification (the "action model") defines the rules of the task. We have thus separated the *what* from the *how* of the Task Manager. The layered approach is a useful way of separating concerns. For example, people interested only in the state of the object store (such as managers) need only study the goal model to understand what statuses mean; people who want to use the system to develop or maintain the object store need to understand both models.

Section 6 presents a proof technique for verifying that the action model preserves the semantics of the goal model. Finally, Section 7 summarizes the approach in a more general setting, and the paper concludes with a discussion of what has been achieved.

2 Example: a task manager for an interactive theorem prover

We illustrate the method on the specification of a (hypothetical) interactive theorem prover, which is being used to develop a *theory*: that is, a collection of mathematical theorems and their proofs. Theorem proving is a relatively small, but important, part of formal software development [15, 16]. It is increasingly being recommended – in conjunction with formal specification and design – by standards for critical software, such as used in safety- or security-critical applications [17, 18]. Interactive theorem proving has been chosen to illustrate the main ideas of this paper because it is relatively easy to explain.

2.1 Background

A proof of a theorem can make use of many other theorems, and the entire network of dependencies between theorems can become large and complex. For example, the `mural` support system for VDM [11], and the `Ergo` proof assistant for the `Cogito` methodology [19], each contain many thousands of theorems and proofs. Machine assistance is required to manage the consistency and completeness of the collection of theorems and proofs.

The traditional approach to maintaining soundness of mathematical theories – as exemplified by formal logic textbooks – is to linearly order the theorems and allow proofs to use only theorems which are proven earlier in the ordering (i.e., to prohibit forward references). Automath [20] is an example of a proof assistant which enforces this style, and the highly influential LCF [21] 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. More recent

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.

generations of theorem provers support more flexible styles of working, whereby theories can be developed on-line in a piecemeal fashion, for example allowing the user to interrupt a proof in order to conjecture and prove useful lemmas, or to work on another proof in parallel [15]. For the present case study, we shall thus suppose that the system supports the storage and use of incomplete proofs.

The next question is whether to allow circularities. It is *critical* to logical soundness of the theory being developed that circular reasoning be eliminated: see Fig. 1. However, while it is possible to design a task manager which prevents circularities from arising, such a system would probably be unpopular with users, since such systems usually force the user into highly convoluted and potentially confusing patterns of use. Syntax-directed editors are an example: to experienced users they can be tedious or even prohibitively complicated.

The ability to support and recover from "inconsistent" states, such as presence of circular reasoning in this case, is vital to the useability of a system. For example, the user may be content to introduce a temporary circularity and then return the system to a consistent state by breaking the circularity at a different point. (In the language editor example, this would correspond to being allowed to temporarily break the syntactical or grammatical rules, and then to reparse the expression at a later point [22].) Thus, for flexibility, our system will allow circular reasoning; the Task Manager will be designed to help users avoid and recover from such situations when they arise.

2.2 Task management requirements

The user's overall task is to develop the theory by using the theorem prover to construct theorems and their proofs. The user's ultimate goal is to create a *complete* theory – one in which all theorems have complete proofs, with no circularities. In practice, however, the user might not be expected to complete every proof in a theory, and some intermediate goal might be sufficient, whereby for example full proofs are given only for certain selected critical theorems – such as important safety properties, in a safety-critical specification – and for all the theorems on which they

depend. (We say such a theorem is "fully established".)

We shall not consider how the user might go about planning and carrying out these tasks: the interested reader is referred instead to e.g. [23]. Rather, we assume the system's task management role is to track dependencies and watch for circularities, and to present useful information to the user to help them manage their task.

The approach to task management advocated here is to associate task *status* values with individual theorems and with the overall theory, to indicate progress towards meeting task goals. Some goals are negative (such as 'try to avoid introducing circularites') and others are positive (such as 'try to fully establish theorem X').

Note however that it is generally not feasible to specify that task status values have *exact* interpretations in terms of properties of the underlying object model. To do so would for example force the introduction of potentially expensive consistency and completeness checks before and after each user action, which would result in unacceptable performance characteristics. Instead, task status values simply indicate what properties can be inferred from status values, without attempting to characterise them fully. (This point is explained in more detail in Section 4 below.)

Returning to the case study, the Task Manager will be required to provide the following task management information:

1. *Indicate if the theory contains circularities.* This is important, since it threatens to compromise the soundness of future work. It is thus critical that there be no 'false negatives': i.e., the user will want to know if there is any possibility at all that a circularity is present. A 'false positive' would be annoying, but acceptable: it might just mean the user has to do a bit more work to establish that all circularities have been removed.

2. *Indicate if the theory is complete.* This shows that the overall task has been finished, so in this case the user will want to have assurance that the theory really is complete (i.e., no false positives).

3. *Indicate if a given theorem is fully established.* Such theorems are generally safe to use in proofs, since they are guaranteed to be true (provided the axioms are true, of course). Thus again, it is critical that there be no false positives.

4. *Indicate which theorems have incomplete proofs.* This will help the user immediately identify theorems whose proofs need more work. False negatives are unacceptable (i.e., where the system indicates that the proof is finished, when it is not).

5. *If a circularity is present, give the user some idea of where it can be found.*

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. The correctness of the tools (and of task management information in particular) can be critical to the soundness of the verification, and thus the tools themselves require a high degree of assurance [1].

3 Modelling the conceptual domain

In this section we describe and formally model the main concepts underlying the system described in Section 2.

3.1 The objects to be managed

We shall be concerned with three kinds of object: theorems, proofs and theories. Each of these is described in more detail below.

Theorems are (named) mathematical statements expressing properties which are believed to be true. 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, or **proofs** for short, are structures which – if correct and complete – establish the truth of theorems (at least, relative to the truth of the other theorems referenced from the proof). 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** is a collection of theorems and their proofs. Theories will be modelled here as a mapping (i.e., a finite partial function) type from theorems to their proofs:

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

3.2 The basic relationships

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:

$$is\text{-}finished : Proof \rightarrow \mathbb{B}$$
$$uses : Proof \rightarrow Thm\text{-}\mathsf{set}$$

While the model we present below makes no assumptions about the efficiency of these tools, it implicitly assumes that it is time-consuming to check that a proof is finished, but relatively quick to check what theorems are used in a proof.

3.3 The derived relationships

In this section we introduce some terminology and describe the properties of the theory that are to be tracked by the Task Manager.

First, we shall 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: $s \in \mathsf{dom}\ T \land t \in uses(T(s))$.

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

$$is\text{-}reference\text{-}chain : Thm^* \times Theory \to \mathbb{B}$$

$$is\text{-}reference\text{-}chain\ (ts, T) \triangleq$$
$$\forall\, i \in \{1, \ldots, \mathsf{len}\ ts - 1\} \cdot ts(i) \in \mathsf{dom}\ T \land ts(i{+}1) \in uses(T(ts(i)))$$

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

$$depends\text{-}on : Thm \times Thm \times Theory \to \mathbb{B}$$

$$depends\text{-}on\ (s, t, T) \triangleq$$
$$\exists\, ts : Thm^* \cdot \mathsf{len}\ ts \geq 2 \land is\text{-}reference\text{-}chain(ts, T)$$
$$\land\ ts(1) = s \land ts(\mathsf{len}\ ts) = t$$

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

$$is\text{-}reference\text{-}loop : Thm^* \times Theory \to \mathbb{B}$$

$$is\text{-}reference\text{-}loop\ (ts, T) \triangleq$$
$$is\text{-}reference\text{-}chain(ts, T) \land ts(1) = ts(\mathsf{len}\ ts)$$

We say a theorem t has a **circular proof** if it depends – directly or indirectly – on itself. A theory is said to **have a circularity** if it contains any theorems with circular proofs:

$$has\text{-}circularity : Theory \to \mathbb{B}$$

$$has\text{-}circularity\ (T) \triangleq$$
$$\exists\, t \in \mathsf{dom}\ T \cdot depends\text{-}on(t, t, T)$$

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:

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

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

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):

$$is\text{-}complete : Theory \to \mathbb{B}$$

$$is\text{-}complete\ (T) \triangleq$$
$$\neg\, has\text{-}circularity(T) \land \forall\, p \in \mathsf{rng}\ T \cdot is\text{-}finished(p) \land uses(p) \subseteq \mathsf{dom}\ T$$

3.4 Some useful lemmas

The following lemmas are logical consequences of the definitions above which are useful in the verification of the Action Model in Section 6.

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

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

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

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

Lemma 3. If, when a theory T is extended with a proof p_0 of a theorem t_0, the new theory has a circularity, then either (a) the circularity must have already been present in T, or (b) p_0 uses theorems that depend on t_0 in T.

$$\forall\, t_0 : Thm, p_0 : Proof, T : Theory \cdot has\text{-}circularity(T \dagger \{t_0 \mapsto p_0\}) \ \Rightarrow$$
$$has\text{-}circularity(T) \lor \exists\, t \in uses(p_0) \cdot depends\text{-}on(t, t_0, T)$$

4 Specification of task status and goals

This section gives a high-level specification (called the goal model) of the Task Manager. The specification is given in terms of the status information that is to be made available, and what can be inferred from this information.

4.1 Task status information

The Task Manager will track the status of individual theorems in the theory, as well as the overall status of the theory. (The status of individual proofs does not need to be tracked, since it can be deduced from that of the corresponding theorem.) There will be three possible status values for the overall theory, and four for individual theorems, as follows:

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

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

The meaning of the different values is explained below.

4.2 The state of the Task Manager

The (abstract) state of the Task Manager will 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.

We shall suppose that the theory is initially empty, and thus has status COMPLETE. The formal specification of the state and initialisation condition is given in Fig. 2. The state invariant expresses the structural constraint that all theorems in the theory have an associated status.

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

Figure 2: Formal specification of the abstract state of the Task Manager.

4.3 The meaning of task status information

Next we specify what can be inferred about the state of the theory from task status information. Each of the task management requirements from Section 2.2 will be treated in turn, and their critical requirements will be formalized as assertions which are required to be satisfied in all states $mk\text{-}TaskManager(T, m, \sigma)$ of the Task Manager.

1. The first requirement was to indicate if the theory contains circularities. The theory status HAS-CIRCULARITY will indicate that a circularity may be present. The critical requirement was that there be no false negatives: i.e., if there is a circularity then the theory status *must* be HAS-CIRCULARITY. This is formalized as the following assertion:

   $has\text{-}circularity(T) \Rightarrow \sigma = \text{HAS-CIRCULARITY}$

2. The next requirement was to indicate when the theory is complete. We use theory status COMPLETE for this purpose. In this case the critical requirement was that there be no false positives: i.e., the status is COMPLETE only if the theory is complete:

   $\sigma = \text{COMPLETE} \Rightarrow is\text{-}complete(T)$

3. The thrid requirement was to indicate if a theorem is fully established, with no false positives:

   $\forall\, t \in \mathsf{dom}\ T \cdot m(t) = \textsc{established} \ \Rightarrow\ \mathit{is\text{-}fully\text{-}established}(t, T)$

4. The fourth requirement was to indicate if a theorem's proof is not finished, with no false negatives:

   $\forall\, t \in \mathsf{dom}\ T \cdot \neg\, \mathit{is\text{-}finished}(T(t)) \ \Rightarrow\ m(t) = \textsc{unproven}$

5. The last requirement was to indicate where to look for a circularity, if one is present. In this case, we shall require that at least one of the theorems in the reference loop has status CIRCULAR:

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

   We decided to require simply that at least one theorem in the loop is given status CIRCULAR, rather than every theorem in the loop, after considering possible designs for the Task Manager. (This is an example of how design and requirement analysis tend to be iterative, intertwined activities.) It became apparent that to insist on the stronger requirement would ultimately result in a lot more work for the user, since they would need to recheck the proofs of every theorem in the loop.

This completes the specification of the Goal Model for the Task Manager.

5 Design of the Task Manager

This section formally specifies the design of a Task Manager which is consistent with the Goal Model given above. (The proof of consistency is outlined in Section 6 below.) The specification describes the meaning of individual user actions, and how they affect task status – the so-called Action Model.

The Action Model shares the state space from the Goal Model, and consists of four core state-changing operations as follows:

1. $\mathit{EditProof}(t_0, p_0)$ – adds theorem t_0 with proof p_0 to the theory store, after deleting the previously stored proof (if any).

2. $\mathit{DeleteProof}(t_0)$ – deletes the proof of theorem t_0 from the theory.

3. $\mathit{CheckProof}(t_0)$ – checks the proof of theorem t_0 and updates its status.

4. *CheckTheory* – updates the theory's overall status.

The operations are explained in more detail below.

Later stages in the design – which are outside the scope of the present paper – would need to address questions such as how the task status information will be displayed, and what other information will be made available to help the user (for example) to

- find the unproven theorems on which a given theorem depends;

- find the theorems which depend (or don't depend) on a given theorem.

For the purposes of this paper, however, we shall assume that the above four operations are the only operations that can directly change the Task Manager's state.

5.1 Changing the proof of a theorem

The operation $EditProof(t_0, p_0)$ changes the proof of theorem t_0 to p_0, or adds it to the theory if t_0 currently does not have a proof. If any theorem in p_0 already depends on t_0, then a circularity will result (see Lemma 3 in Section 3.4); in this case the status s_0 of t_0 will be set to CIRCULAR and the overall theory status to HAS-CIRCULARITY. Otherwise, the theorem's status will be set to the default value UNPROVEN and the theory's overall status to INCOMPLETE, pending checks. The specification is given formally in Fig. 3.

$$
\begin{array}{l}
EditProof\,(t_0 : Thm, p_0 : Proof) \\
\textsf{ext wr}\ theory : Theory, status : Thm \xrightarrow{m} ThmStatus, overall : TheoryStatus \\
\textsf{post let}\ s_0 = \ \textsf{if}\ \exists\, t \in uses(p_0) \cdot depends\text{-}on(t, t_0, \overleftarrow{theory}) \\
\qquad\qquad\qquad \textsf{then CIRCULAR} \\
\qquad\qquad\qquad \textsf{else UNPROVEN} \\
\quad \textsf{in} \\
\quad theory = \overleftarrow{theory} \dagger \{t_0 \mapsto p_0\} \\
\quad \wedge\ status = \overleftarrow{status} \dagger \{t_0 \mapsto s_0\} \\
\quad \wedge\ overall = \ \textsf{if}\ \overleftarrow{overall} = \textsf{HAS-CIRCULARITY} \vee s_0 = \textsf{CIRCULAR} \\
\qquad\qquad\qquad \textsf{then HAS-CIRCULARITY} \\
\qquad\qquad\qquad \textsf{else INCOMPLETE}
\end{array}
$$

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

Note that this operation is conservative in the way it assigns statuses: for example, the proof p_0 may actually be complete, in which case PROVEN might be a more appropriate status. Given that proof checking can be time-consuming, the decision as to when to run the check should be under user control, and so thus the check has been allocated to the *CheckProof* operation instead. Note that *EditProof* errs on the side of caution, by assigning statuses in a way which is consistent with the requirements specified in Section 4.3.

5.2 Deleting the proof of a theorem

When the proof of theorem t_0 is deleted, the statuses of theorems which depend on t_0 may need to be changed: in particular, a theorem which was previously established may now no longer be established. Similarly, if the theory was previously complete, and the theorem was used elsewhere in the theory, then the theory will no longer be complete.

The specification of the $DeleteProof(t_0)$ operation is given formally in Fig. 4. Again, the operation is conservative in the way it assigns statuses. Note that the operation has a precondition: it can only be applied to a theorem with an existing proof. This is a (simple) example of how the Action Model can specify the conditions under which an operation is enabled.

$$
\begin{array}{l}
DeleteProof\ (t_0 : Thm) \\
\textsf{ext wr}\ theory : Theory, status : Thm \xrightarrow{m} ThmStatus, overall : TheoryStatus \\
\textsf{pre}\ \ t_0 \in \textsf{dom}\ theory \\
\textsf{post}\ theory = \{t_0\} \lhd \overleftarrow{theory} \\
\quad \land\ \forall\, t \in \textsf{dom}\ theory \cdot \\
\qquad \textsf{if}\ \overleftarrow{status}(t) = \text{ESTABLISHED} \land depends\text{-}on(t, t_0, theory) \\
\qquad \textsf{then}\ status(t) = \text{PROVEN} \\
\qquad \textsf{else}\ status(t) = \overleftarrow{status}(t) \\
\quad \land\ \ \textsf{if}\ \overleftarrow{overall} = \text{COMPLETE} \land \exists\, p \in \textsf{rng}\ theory \cdot t_0 \in uses(p) \\
\qquad \textsf{then}\ overall = \text{INCOMPLETE} \\
\qquad \textsf{else}\ overall = \overleftarrow{overall}
\end{array}
$$

Figure 4: Operation for deleting the proof of a theorem.

5.3 Checking the status of a theorem

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 that time the circularity may have been broken, say by deleting the proof of 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 subsequently been supplied.

The operation $CheckProof(t_0)$ will be provided to update the status of theorem t_0, if appropriate. The specification is given formally in Fig. 5, where $check\text{-}if\text{-}estab(t, T, m)$ looks recursively through the statuses of theorems on which t depends to see whether their proofs are complete:

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

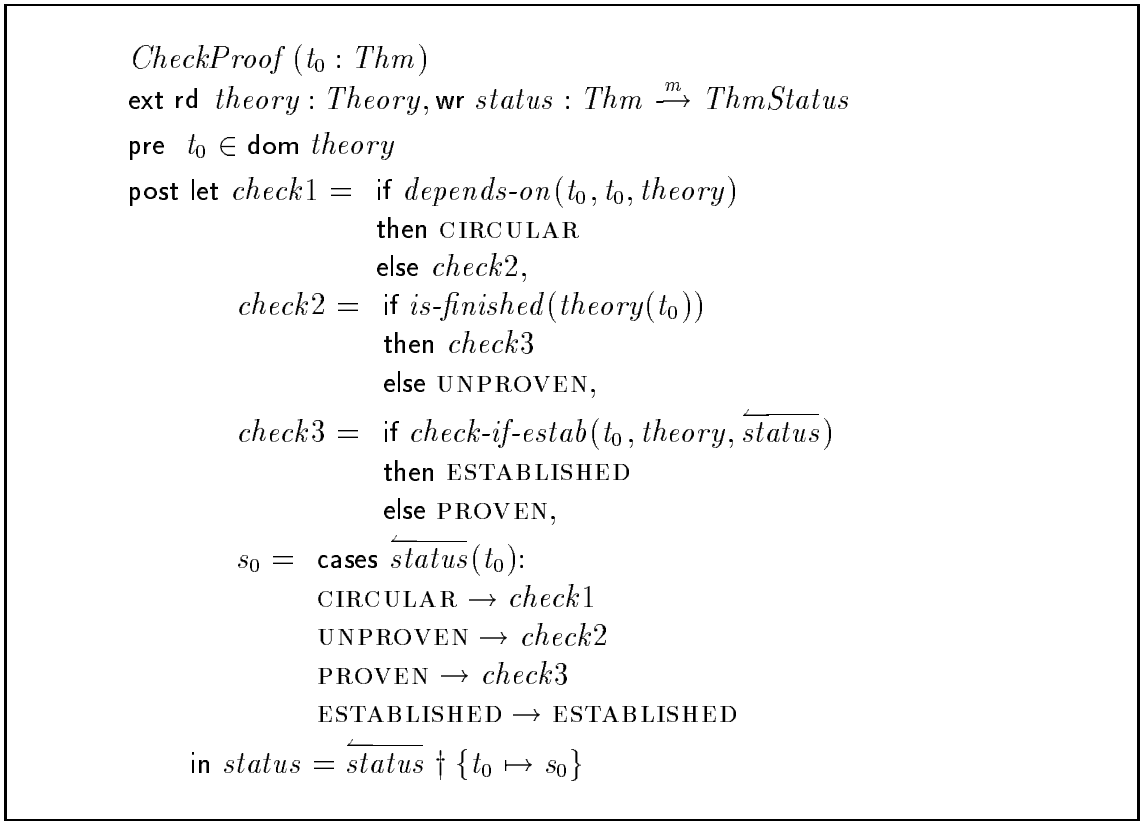

Figure 5: Operation for updating the status of a theorem.

It will follow from the properties given in Section 4.3 that, if *check-if-estab*(t, T, m) 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.) Note also that the definition of *check-if-estab* given here is a specification only; an implementation would need to ensure termination even in the presence of circularities.

Note that the *CheckProof*(t_0) operation applies the proof checker to just the proof of theorem t_0. It may thus be necessary for the user to apply the operation a number of times, on theorems on which t_0 depends, before the system will recognise that t_0 is fully established.

5.4 Checking the overall status of the theory

Finally, the operation *CheckTheory* updates the theory's overall status by checking the statuses of the theorems in the theory. The specification is given in Fig. 6. The fact that the theory is complete is all of its theorems have status ESTABLISHED follows from the properties asserted in Section 4.3 and Lemma 1 of Section 3.4.

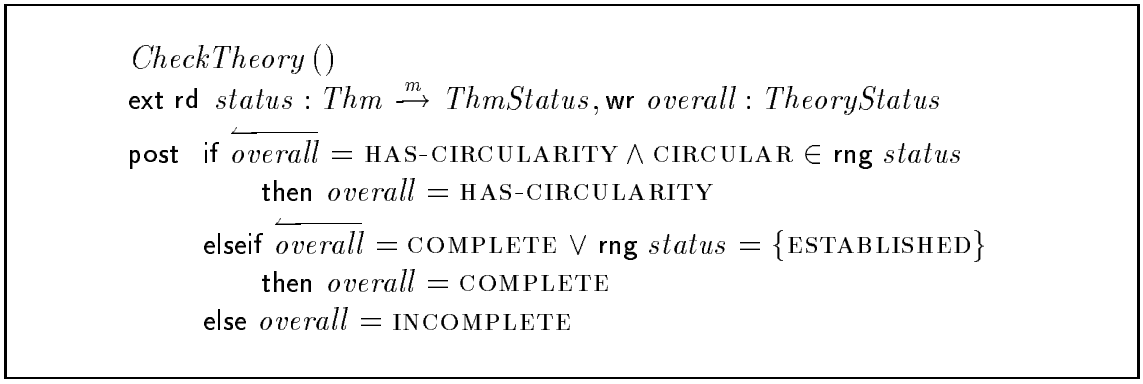

Figure 6: Operation for checking a theory.

6 Verification of the Task Manager design

This section illustrates the steps involved in the verification of the Goal Model and Action Model given above. The proofs are too lengthy to give here in full.

6.1 Well formedness and satisfiability of the models

To show that the models are mathematically meaningful it is necessary to check that all expressions are well formed [16]: i.e.,

- partial functions are applied only to arguments within their domains;
- recursive definitions are well founded.

It is also necessary to check that the operations are mathematically feasible, in the sense that their postconditions are satisfiable (can be achieved).

As an example of a well formedness proof, consider the term $\overleftarrow{status}(t)$ in the postcondition of *DeleteProof* (Fig. 4). The term is well formed in context since $t \in \mathsf{dom}\ theory$ and

$$theory = \{t_0\} \lhd \overleftarrow{theory}$$

According to the semantics of VDM, the state invariant is implicitly part of the pre- and post-conditions of all operations, thus

$$\mathsf{dom}\ theory = \mathsf{dom}\ \overleftarrow{theory} - \{t_0\} = \mathsf{dom}\ \overleftarrow{status} - \{t_0\}$$

and $t \in \mathsf{dom}\ \overleftarrow{status}$ as required. The proofs of the other cases are 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, which is easy to check.

6.2 Verification of the Action Model against the Goal Model

To show that the Action Model is consistent with the Goal Model, it is necessary to show that the task management requirements defined in Section 4.3 are preserved

by the Action Model. This can be done by showing that they are true in the initial state of the Task Manager and are preserved by all enabled operations.

From the initialization condition we know that $T = \{\mapsto\}$ initially, and hence that $\neg\, \mathit{has\text{-}circularity}(T)$, $\mathit{is\text{-}complete}(T)$ and $\mathsf{dom}\ T = \{\,\}$. It is easy to check that the five required properties are true in this case.

We show below that Requirement 1 is preserved by the *EditProof* operation specified in Fig. 3 above. The proofs for the other operations – and the proofs of the other properties – are similar.

Let $\tau = \mathit{mk\text{-}TaskManager}(T, m, \sigma)$ be the state in which the operation is invoked and $\tau' = \mathit{mk\text{-}TaskManager}(T', m', \sigma')$ the state immediately after. It follows that τ and τ' are related by the postcondition of $\mathit{EditProof}(t_0, p_0)$: thus e.g. $T' = T \dagger \{t_0 \mapsto p_0\}$). Suppose also that the requirements hold in τ (the "Induction Hypothesis"). We are required to prove that τ' satisfies Requirement 1: i.e. that

$$\mathit{has\text{-}circularity}(T') \;\Rightarrow\; \sigma' = \textsc{has-circularity}$$

The proof is by contraposition: i.e., we suppose that $\sigma' \neq$ HAS-CIRCULARITY and show that $\neg\, \mathit{has\text{-}circularity}(T')$. From the postcondition of $\mathit{EditProof}(t_0, p_0)$ it follows that $\sigma \neq$ HAS-CIRCULARITY and $s_0 \neq$ CIRCULAR, where s_0 is the assigned status of t_0. It also follows from the postcondition that

$$\neg\, \exists\, t \in \mathit{uses}(p_0) \cdot \mathit{depends\text{-}on}(t, t_0, \overleftarrow{\mathit{theory}})$$

From the fact that $\sigma \neq$ HAS-CIRCULARITY and the Induction Hypothesis, it follows that $\neg\, \mathit{has\text{-}circularity}(T)$. Finally, it follows from Lemma 3 in Section 3.4 that

$$\neg\, \mathit{has\text{-}circularity}(T \dagger \{t_0 \mapsto p_0\})$$

and hence that $\neg\, \mathit{has\text{-}circularity}(T')$, as desired.

7 Summary of the approach

This section summarizes the key elements of the approach in a general setting.

7.1 Formalizing the conceptual domain

The first step is to build a formal model of main concepts:

1. Define the object store: i.e., the kinds of object in the system and the possible relationships between them.

2. Specify the functionality of the tools which are available for manipulating the object store. Specifications should include the tools' 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).

3. Define the task goals, including intermediate goals and negative goals, such as situations to be avoided. Define the goals in terms of consistency and completeness conditions on the object store.

7.2 Determining task statuses

The next stage is to determine what task status information is to be communicated to the user:

1. Define requirements for the kind of task management information which is to be made available.

2. Determine which objects are to be tracked and their possible statuses. Status values should be chosen to correspond to important stages in the development of individual objects, and how they relate to task goals. Note that it may not be practical – or even desirable – to track the status of every object in the system.

7.3 Defining a Goal Model

The Goal Model indicates what properties of the underlying object store can be inferred from task status information. It consists of two parts:

1. A definition of the state of the Task Manager as consisting of the state of the object store itself, together with the status information.

2. A set of task management requirements, expressed as assertions which relate task statuses to object-store properties. The assertions typically take one or both of the following forms:

   (a) "if objects $x_1, \ldots, x_n$ have statuses $s_1, \ldots, s_n$ then property P holds" (sufficient conditions)

   (b) "if property Q holds then object x has status s" (necessary conditions).

In the example above we used a state-based modelling language (VDM-SL) to define the state of the task manager. Status information was modelled using mappings (finite partial functions) from objects to their status. A state invariant was used to record constraints on the object store, such as structural relationships that should always hold. Task management requirements were given as constraints on the state.

7.4 Defining an Action Model

The Action Model is a formal specification of the design of the Task Manager. Its purpose is to define the key user actions and say how they affect the object store and the status of objects in the store.

VDM-SL operations were used to formalize actions in the example above. Preconditions were used to specify constraints on inputs and the states from which operations can be invoked.

7.5 Verifying the models

Verification of the models has two parts:

1. Checking that the models are mathematically meaningful. Standard VDM proof obligations can be used for this [16].

2. Checking that the Action Model is consistent with the Goal Model – in other words, that the design of the Task Manager respects the meaning of the task status information as defined in the Goal Model.

Performing such verification leads to a high degree of assurance in the meaning and accuracy of task management in the resulting system. (Of course there is still a need to verify that the implementation satisfies the Action Model.)

8 Conclusions

This paper has been concerned with two aspects of useability of interactive systems: support for flexible, non-proscriptive work patterns, and provision of useful, accurate task management information. We presented a method for formally specifying and reasoning about task management, focussing on the meaning and accuracy of task status information. The method separates modelling of task status information from modelling of interactions. A proof method is described for checking that the two models are consistent.

The method is well suited to data-intensive applications in which the system is being used interactively to develop complex collections of interrelated objects, where the user takes primary responsibility for planning and carrying out the tasks while relying on the system to track the consistency and completeness of the object store. 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 [8].

The paper illustrated the method on a case study concerning interactive theorem proving. The 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 interactive systems. It is also an interesting application in itself, being to this author's knowledge one of the first documented designs of a theory management system that ensures logical soundness of the underlying theory store while allowing incomplete proofs and the possibility of circular reasoning. A broadly similar approach has been applied to a process model for a formal development support environment, to track the co-development of a formal specification and its verification [24].

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 constructive feedback from Jawed Siddiqi and Chris Roast which helped improve the form of this paper.

References

[1] U.K. Ministry of Defence. Safety Management Requirements for Defence Systems Containing Programmable Electronics. Second Draft Defence Standard 00-56, August 1996.

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] J. G. Turner and T. L. McCluskey. *The Construction of Formal Specifications: An Introduction to the Model-based and Algebraic Approaches.* McGraw Hill, 1994.

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

[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] 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.

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

[12] P.A. Lindsay. A formal basis for modelling process and task management aspects of user interface design. In *Proc. Formal Aspects of The Human Computer Interface*, Sheffield, UK, Sept 1996. BCS FACS, Springer Verlag Electronic Workshops in Computing. http://www.springer.co.uk/ewic/workshops/FAHCI/index.html. Available by ftp as SVRC TR 96-07.

[13] ISO/IEC International Standard 13817-1, December 1996. VDM Specification Language — Part 1: Base language.

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

[15] P.A. Lindsay. A survey of mechanical support for formal reasoning. *Software Engineering Journal*, 3(1):3–27, January 1988.

[16] 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.

[17] IEC. Functional safety: safety-related systems. Draft International Standard IEC 1508, June 1995.

[18] Information Technology Security Evaluation Criteria (ITSEC). Commission of the European Communities, June 1991. Provisional Harmonised Criteria.

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

[20] 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.

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

[22] J. Welsh, B. Broom, and D. Kiong. A design rationale for a language-based editor. *Software—Practice and Experience*, 21(9):923–948, 1991.

[23] 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.

[24] K.J. Ross and P.A. Lindsay. A precise examination of the behaviour of process models. In *Proc. 2nd Int. Symp. of Formal Methods Europe (FME'94)*, LNCS 873, pages 251–270. Springer Verlag, 1994. An extended version of this paper appears as SVRC TR 94-7.