Application of Information Technology ■

The Syntax and Semantics of the PRO*forma* Guideline Modeling Language

David R. Sutton, BA, PhD, John Fox, BSc, PhD

Abstract PRO*forma* is an executable process modeling language that has been used successfully to build and deploy a range of decision support systems, guidelines, and other clinical applications. It is one of a number of recent proposals for representing clinical protocols and guidelines in a machine-executable format (see <www.openclinical.org>). In this report, the authors outline the task model for the language and provide an operational semantics for process enactment together with a semantics for expressions, which may be used to query the state of a task during enactment. The operational semantics includes a number of public operations that may be performed on an application by an external agent, including operations that change the values of data items, recommend or make decisions, manage tasks that have been performed, and perform any task state changes that are implied by the current state of the application. *Disclosure*: PRO*forma* has been used as the basis of a commercial decision support and guideline technology Arezzo (Infermed, London, UK; details in text).

■ **J Am Med Inform Assoc.** 2003;10:433–443. DOI 10.1197/jamia.M1264.

There is a growing body of research on the representation of clinical guidelines in forms that are interpretable by computer systems. The longest established representation is the *Arden Syntax*,[1,2] which sets out to provide a standard for capturing condition-action rules and has been widely taken up by industry. Although the Arden Syntax has been important and influential, it is recognized that to formalize complex decisions and care pathways or clinical workflow, a more expressive formalism will be needed.

To address this, a number of newer languages have been developed that typically embed logical rules in higher order structures that represent tasks such as decisions, plans, and actions, which can be composed into time-oriented networks to represent protocols and guidelines. The most developed examples of these "task network" formats are *Asbru*,[3,4] *EON*,[5] *GLIF*,[6–8] *GUIDE*,[9,10] *PRODIGY*,[11,12] and PRO*forma*.[13,14] These and other representations are reviewed at <www.openclinical.org>, and their features and expressiveness have been compared by Peleg et al.,[15] who have placed them in the context of existing standards for workflow such as those developed by the Workflow Management Coalition.[16]

This new family of languages is showing potential for developing a range of different types of applications and for standardizing representations of clinical processes such as guidelines and protocols. However, there are, as yet, few practical tools available for authoring and implementing applications using the languages, and where these exist they have only been developed by the groups who designed the languages. This is clearly unsatisfactory if we are to achieve wider dissemination and independent assessment by the medical informatics community.

To promote wider take-up of these promising new technologies, it would be highly desirable to provide a precise public definition of the syntax and semantics for each of the representations they use. A partial definition of the semantics of Asbru has been created using Structured Operational Semantics,[17,18] and a formal semantics for an early version of PRO*forma* is discussed in detail by Fox and Das.[33] However, neither of these efforts provides sufficient information to permit others to implement tools. A detailed public definition would permit other groups to:

- Assess what each representation can and cannot capture. The Peleg et al. study[15] made an important start on this, but their study was based on a limited review of syntactic features of the representations without considering execution semantics.
- Carry out independent evaluations and comparisons of the representations on their own clinical applications.
- Implement software tools for building applications based on publicly defined formats (e.g., guideline authoring tools) and field them in clinical use ("enactment engines").
- Investigate the value of "open source" knowledge, in which clinical guidelines, protocols, and the like can be shared and enacted on software platforms from alternative suppliers.

This article attempts to address this requirement for PRO*forma* by providing a full syntax in Backus Naur Form (BNF) and an operational semantics for the language. Due to

Affiliations of the authors: Oxford Brookes University, Oxford, England (DRS); Advanced Computation Laboratory, Cancer Research, London, UK (JF).

PRO*forma* has been used as the interchange format and guideline specification language for a commercial decision support technology: Arezzo from *Infer*Med Ltd. The first author has no connection with this company; the second is a stock-holder in the company but has no day-to-day involvement in its commercial activities.

Correspondence and reprints: David R. Sutton, Oxford Brookes University, Oxford, England; e-mail: <david.r.sutton@ntlworld.com>.

Received for publication: 10/04/02; accepted for publication: 05/13/03.

limitations of space, we present only the core concepts and interpretation of the syntax and semantics here, but the full definition is available in a companion document, which can be downloaded from our website.[19] It is hoped that this publication will facilitate use of the language as widely as possible and proposals for continuing development.

Background

The PROforma Language and Method

PROforma has been used to develop a range of clinical applications, including:

- CAPSULE for supporting general practitioners (GPs) in prescribing medications for common conditions.[21]
- ERA, which assists GPs in complying with urgent (two-week) cancer referral guidelines (currently on trial in the UK National Health Service, see <www.infermed.com/era>).
- RAGs, a system for risk assessment in cancer genetics.[22,23]
- LISA, a decision support system embedded in a clinical database system for helping clinicians comply with the dosage rules for a trial protocol for children with acute lymphoblastic leukemia.[24]
- A system that integrates decision support with *Clinical Evidence* (in collaboration with BMJ Publishing).[25]

Two main implementations of a PROforma engine are currently available, the Arezzo implementation, which is available commercially from *Infer*Med Ltd. (London, UK) and the Tallis implementation from Cancer Research UK. The Arezzo implementation's behavior is closely similar, although it is based on a somewhat earlier PROforma language model (e.g., a larger set of control states, see text).

A commercial toolset for authoring and enacting guidelines written in PROforma is available from *Infer*Med Ltd.[26] This company has also developed clinical applications using this, including Arno, a system to advise on the management of pain in palliative care, and MACRO, a Web-based clinical trial management system.

In PROforma, a guideline is modeled as a set of tasks and data items. The tasks are organized hierarchically into *plans*. The PROforma task model divides tasks into four classes: *Actions* represent some procedure that needs to be executed in the external environment (e.g., administering a drug or updating a database). *Enquiries* represent points in a guideline at which information needs to be acquired from some person or external system. *Decisions* are points at which some choice has to be made, either about what to believe or what to do. *Plans* are collections of tasks that are grouped together for some reason, perhaps because they share a common goal, use a common resource, or need to be done at the same time.

PROforma processes may be represented diagrammatically as directed graphs in which nodes represent tasks and arcs represent *scheduling constraints*. By convention, plans are represented as rounded rectangles, decisions as circles, enquiries as diamonds, and actions as squares. A guideline itself contains a single root plan, which may be recursively divided into sub plans.

Figure 1 illustrates a PROforma application in which an enquiry "Presentation" gathers information about the patient and is followed by a decision, "Overdose Type," which may, in turn, be followed by one of two plans: "Staggered Overdose" or "Normal Overdose." These two plans could themselves be displayed graphically using the conventions outlined here.

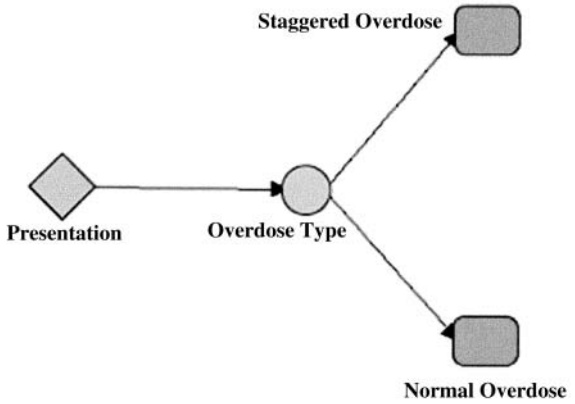

Figure 1. A simple PROforma application.

Properties of Tasks and Data Items

Each task can have a number of properties whose values determine how it is to be interpreted. The value of a property may be a scalar value (e.g., an integer), an expression, or it may be an object, which, itself, has properties. We do not have space here to describe all the properties that tasks or data items may have. However, we present a number of examples. First, all tasks and all data items have the following properties: *Caption and description:* In PROforma anything that can have any properties at all can have a text caption and a description, which are used to provide short or long comments (e.g., explanations of the intended purpose of a task).

All tasks have the following properties: *Precondition*: a truth-valued expression that must be true when a task is started. *Task scheduling constraints:* logical constraints that prevent one task from starting before another task or set of tasks has been completed.

Particular subclasses of tasks have distinctive properties. Decisions have the following properties: *Candidates:* objects representing the options to be considered when a decision is to be taken. Candidates have properties of their own, including a *recommendation rule*, which is a PROforma expression used to express the conditions under which it would be appropriate to commit to that candidate. *Arguments:* objects representing arguments for, against, or simply relevant to a particular candidate. An argument consists of a truth-valued expression and a caption and description describing that expression.

In addition, plans have the following control properties: *Termination and abort conditions:* truth-valued expressions that represent sufficient (though not necessary) conditions for successfully terminating the current plan and continuing enactment of successor tasks, or aborting the plan and canceling downstream tasks.

PROforma Software

A number of software components have been written to create, visualize, and enact PROforma guidelines. The *Tallis* software suite developed by this group includes a set of components written entirely in Java. The suite includes two

applications: the *Composer*, which supports creation, editing, and graphical visualization of guidelines, and the *Tester*, which enables a developer to step through and debug a guideline. These applications access various other Tallis components via their APIs. These other components include a *Parser*, which reads and writes guidelines expressed as text files, and an *Engine*, which enacts guidelines and allows them to be manipulated by other applications. We also have developed various Java Servlets that allow guidelines to be published and enacted over the Web. Note that the PRO*forma* language definition does not address any data security issues that may be raised by communication between components using HTTP (e.g., patient data transmission); these are regarded as implementation matters.

The parser and engine can be viewed respectively as concrete implementations of PRO*forma*'s syntax and operational semantics.

Figure 2 illustrates how the Tallis components may be deployed so as to allow guidelines to be developed and tested on a standalone PC or workstation. The Composer application is used to graphically create and edit the guideline. This tool calls on the Parser to save the guideline as a text file and to subsequently reload such text files. The guideline is tested and debugged using the Tester application, which calls on the Engine to enact the guideline.

Figure 3 shows how the Tallis components may be deployed so as to allow guidelines to be enacted over the Web. A user sitting at a client PC or workstation starts the guideline and interacts with it through a Web browser. A Web server processes HTTP requests from the browser by creating a Tallis Engine and then using the Engine API to load and enact the guideline. The Web server may obtain patient data from a database server using a database connectivity protocol such as JDBC.

We have developed a representation format that allows the relationship between data items in a guideline and records in a patient record system to be expressed as an XML file. However, no implementation of this format currently exists. Hence, Java code must be written to map the guideline onto the patient record system.

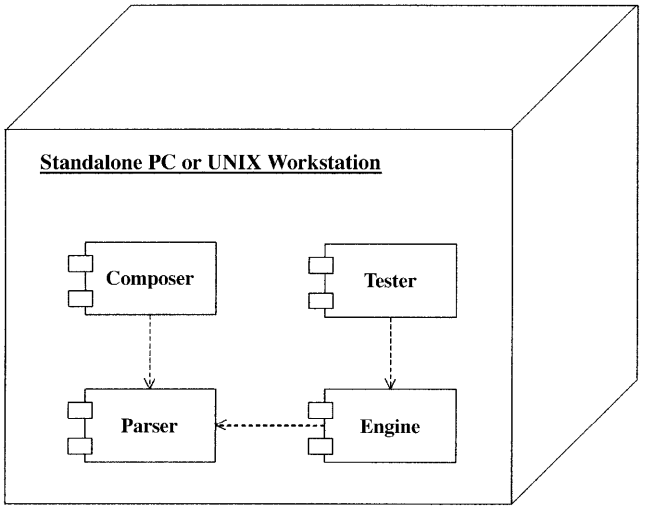

Figure 2. Deployment of Tallis components in the guideline development environment.

An Example Guideline

We can illustrate the use of PRO*forma* by looking at the simple guideline shown in Figure 4. This guideline is an implementation of a UK Government directive to the effect that all patients presenting to their GPs with symptoms indicative of breast cancer should be seen by a specialist consultant within two weeks. The guideline consists of five tasks. The enquiry *Clinical Information* describes the data needed by the guideline, these data include the patient's age and sex and various true/false data items such as whether the patient is experiencing intractable pain. The decision, *Referral Decision*, defines the arguments in favor of and against urgent referral (within two weeks), nonurgent referral, and no referral at all. The actions, *Two week referral*, *Nonurgent referral*, and *No referral* have properties whose values describe what should be done in these three possible cases.

Referral Decision has three *candidates*, which we shall imagine have the same names as the three actions in the diagram. Each candidate will be associated with a number of *arguments*, which are logical conditions that influence whether the candidate will be recommended. These arguments are combined into a *recommendation rule* for the candidate.

The arrows in the diagram indicate *scheduling constraints*, for instance, the constraint that *Referral Decision* must wait until *Clinical Information* has been completed. A task also may have *preconditions*, which are examined after its scheduling constraints have been met and are used to determine whether to proceed with the task. In this example only the three actions have preconditions. These test whether *Referral Decision* recommended the candidate corresponding to that action. For instance, the precondition of *Two week referral* is result_of('Referral Decision') = 'Two week referral'.

Figure 5 describes a concrete scenario illustrating the use of the guideline. The diagram is relatively high level and shows the user interacting directly with a PRO*forma* engine, whereas in reality this interaction would be mediated by some sort of user interface.

During the enactment of a guideline, each task may undergo state transitions. A task may be in one of four states: *dormant*, *in_progress*, *completed*, or *discarded*. The meanings of these states and the allowable transitions between them are set out in "Task States" under "PRO*forma* Components" in the System Description, below.

At the start of the scenario illustrated in Figure 5 the guideline has been loaded, and all five tasks are in the *dormant* state. The engine then repeatedly examines the properties of the tasks to determine what state changes and other actions should occur. The sequence of events is as follows (numbers in brackets refer to the numbering of the messages in the figure):

(1) The engine infers that the *Clinical Information* enquiry can enter the *in_progress* state. (2) The enquiry then marks all of its data sources as being *requested*; this indicates that values need to be supplied for these values. The asterisk before the requestData() message indicates that the message is sent several times because several data items are requested. (3) The user then asks which data items are requested and (4) supplies appropriate values for those items. (5) The engine infers that *Clinical Information* can enter the *completed* state, because all of its sources have been supplied with values. (6)

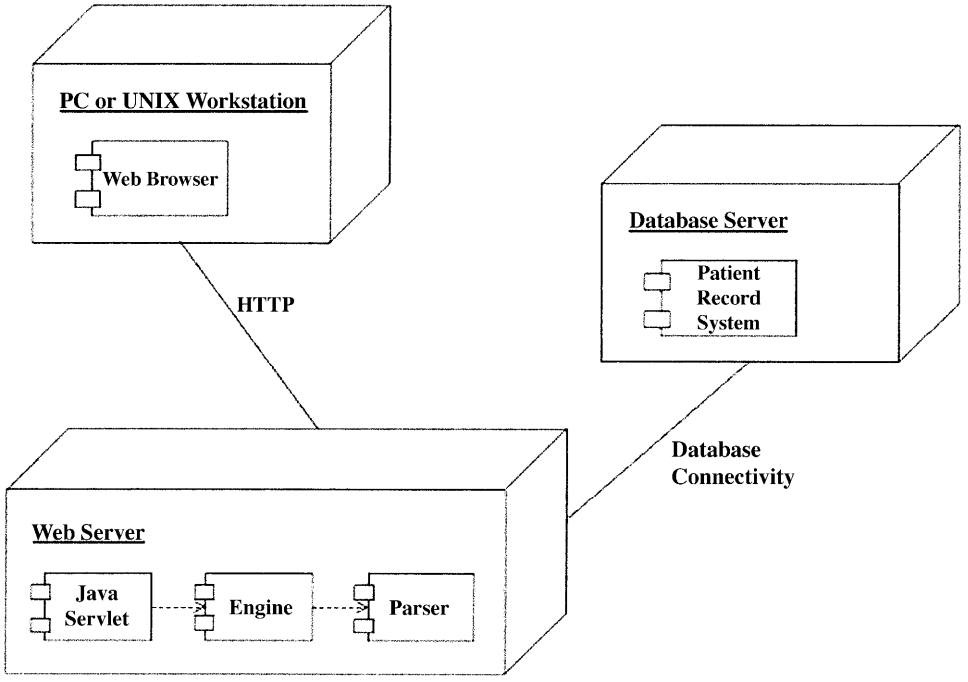

Figure 3. Deployment of Tallis components when enacting a guideline over the Web.

Referral Decision now becomes *in_progress* because its scheduling constraint has been met. (7, 8) The user than asks for the recommended candidate of *Referral Decision*. This candidate is calculated by evaluating the decisions arguments and combining them in ways described by the candidates' recommendation rules. (9, 10) In this scenario, we imagine that the recommended candidate was *Two week referral*. The user commits to this candidate and (11) *Referral Decision* enters the *completed* state. At this point the preconditions of the actions *Two week referral*, *Non urgent referral*, and *No referral* are examined and, as a result (12, 13, 14) the first of these actions enters the *in_progress* state, and the other two enter the *discarded* state. Finally, (15, 16) the user confirms that the referral has been made, and *Two week referral* enters the *completed* state.

Objectives

Why Does PRO*forma* Need a Syntax and Semantics?

Our aim is to provide a means by which centers of expertise can exchange information about medical processes and to permit different software producers to develop tools that are capable of reading and processing applications written by others. We see great potential for developing guidelines in an "open source" manner. To do this, PRO*forma* must have a defined, publicly available syntax and semantics so that the meaning of guidelines is unambiguous, and it is possible to determine whether any given piece of software is correctly reading and processing PRO*forma*.

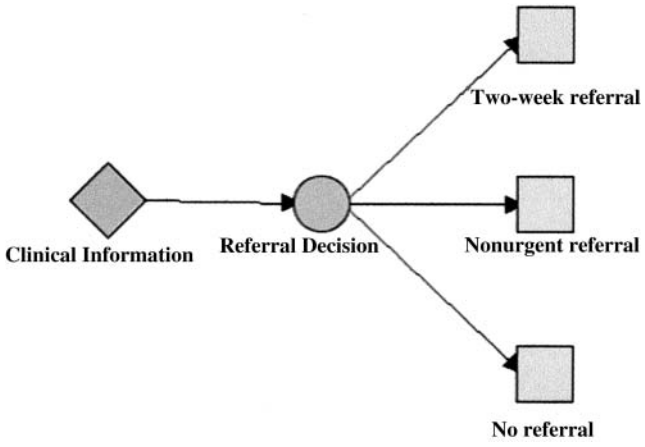

Figure 4. A Simple PRO*forma* guideline.

What Criteria Should the Syntax and Semantics Meet?

Before setting out the syntax and semantics of PRO*forma* we will set out some criteria that have guided their development.

1. The Syntax should, as far as possible, conform to or use existing standards. These may be formal standards such as XML or informal standards such as the standard meaning and precedence of arithmetic operators.
2. The semantics should be specified in such a way that any operation performed on a guideline has a precisely defined and unambiguous effect. It should be noted that this is not a "motherhood" statement. Many languages contain deliberate and explicit ambiguities in their semantics. For instance, the C language[27] explicitly leaves undefined the order in which most operators evaluate their arguments.
3. The syntax should take into account that guidelines may be viewed graphically in ways that hide certain details of the guideline text. For instance, the order in which tasks appear in the guideline text may not be apparent to a user and consequently should not affect the semantics of the guideline.
4. Where possible, the semantics should allow operations to be performed in parallel without ambiguity. For instance, it should be possible for a user to request that two data items be updated without worrying that the effect of these updates will depend on the order in which they occur.

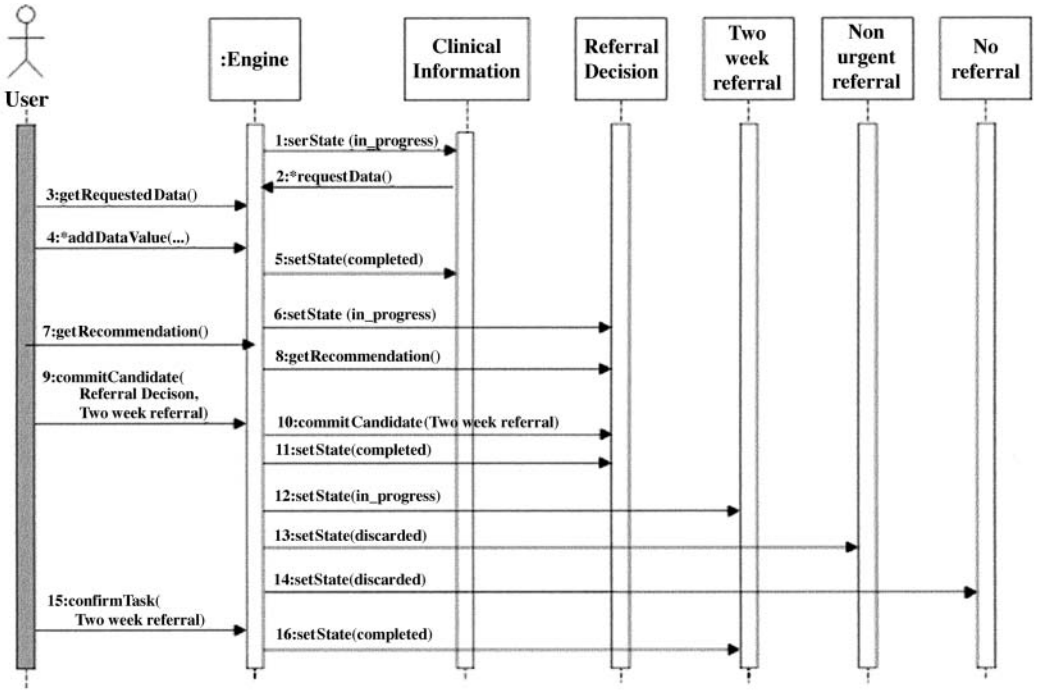

Figure 5. Sequence diagram shows the use of the guideline shown in Figure 4.

5. The syntax should include an expression language. The semantics should define the value of expressions and allow such expressions to be used to query the guideline. The semantics also should guarantee that evaluation of expressions does not have any side effects, i.e., that it does not change the state of the guideline.
6. The description of the syntax and semantics should be sufficiently formal to leave no ambiguity but should not require the reader to have any more background knowledge of mathematics than is necessary and should not introduce any more specialist notation than necessary.
7. The semantics should make it easy to reason about the behavior of a guideline, for instance, whether it will terminate under some given set of conditions.

Approaches to the Definition of a Semantics

There are a number of possible ways in which the semantics of a language may be defined. Ullman and Aho[28] identify the following approaches:

Mathematical (or denotational) semantics: in this approach, a mapping is defined between sentences in the language and mathematical objects that these sentences are said to denote.[29]

Axiomatic definition: rules are defined that relate the values of data before and after the execution of each language construct.[30]

Extensible definition: the semantics is defined in terms of a set of primitive operations.

Translation: the semantics of a language are defined through rules that specify how it may be translated into some other language whose semantics are already known, such as the lambda calculus.[31]

Operational semantics: an abstract machine is described and the enactment and rules are provided for enacting programs on this abstract machine.

The PRO*forma* Specification provides an operational semantics for the language. An abstract PRO*forma* engine is described (see The Abstract PRO*forma* Engine Section) along with a set of "public operations" (see Public Operations of the Abstract Engines Section) that may be performed on the engine by an external system. Rules are set out that describe how the public operations change the state of the abstract engines.

The style and notational conventions used in the PRO*forma* specification are in some ways similar to Z.[32] However, we have attempted to provide a definition of the semantics that can be understood without the extensive background reading that would be required to understand a definition written in Z.

System Description

The Syntax of PRO*forma*

The syntax of PRO*forma* is set out as a Backus Naur Form (BNF) on our website.[20] Guidelines that do not conform to this syntax cannot be enacted. The BNF syntax can be divided into two parts. The syntax of PRO*forma* expressions defines the forms that PRO*forma* allows logical conditions and mathematical expressions to take. This part of the syntax is likely to be enhanced in the future as additional mathematical and logical operators are added to the language. The rest of the BNF defines how the definitions of tasks and other guideline components should be arranged and separated. It is likely that this part of the syntax will be redefined in XML.

The PRO*forma* specification also sets out a type inference algorithm for PRO*forma* expressions and imposes type restrictions on expressions that are values of properties. For instance, the value of the *precondition* property of a task must be a truth-valued expression. The current implementation of PRO*forma* warns if an expression cannot be typed or has the

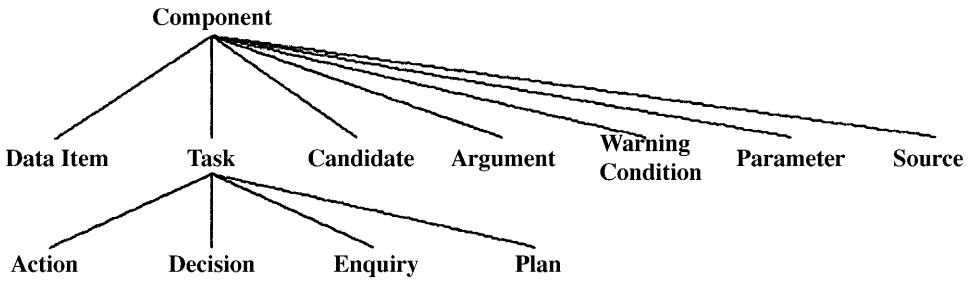

Figure 6. The PRO*forma* component set.

wrong type but does not prevent guidelines containing such errors from being loaded or enacted.

PRO*forma* Components

The semantics of PRO*forma* treats a guideline as a set of objects, referred to as PRO*forma components*. The classes that are used to instantiate these objects are arranged in an inheritance hierarchy as illustrated in Figure 6.

The semantics of PRO*forma* does not impose any real world interpretation of the classes shown in Figure 6. However, the intended interpretation of most of these classes has been sketched out above. The only classes not mentioned in that section are warning conditions (which describe conditions that should be checked when a value is supplied for a data item), parameters (which are used to hold data specific to a particular instance of a task), and sources (which are used to describe the information that should be gathered by an enquiry task).

As we previously mentioned, each class has a number of named *properties*, and each class inherits all the properties of its superclasses. An instance of a class will have values for each of the properties of that class (including those inherited from its superclasses). For example, when adding an instance of the *Task* class to a guideline, we must specify what values that instance has for each of the properties of the *Task* class including those properties that it inherits from the *Component* class, for example, the *name* and *caption* properties.

Each instance of any component has a *Component Identifier* that is unique to that instance, i.e., no two instances have the same identifier even if they are instances of different classes.

The value of a property must be a PRO*forma Value*, that is, it must be one of the following:

- A number (either real or integer).
- A text string.
- A PRO*forma* expression (see The Semantics of PRO*forma* Expressions).
- A Component Identifier.
- One of a number of constants defined in the PRO*forma* specification. We shall not list all of these, but they include the values *unknown*, *true* and *false*, as well as the task states *dormant*, *in_progress*, *discarded*, and *completed* (see Task States, below).
- A finite sequence of PRO*forma* values. We use angle brackets to denote sequences, e.g. ⟨1,2,3,4,5⟩.

Task States

All tasks have a *state* property, which can take four different values: *dormant*, *in_progress*, *discarded*, and *completed*. All tasks initially are in the *dormant* state. The *PROforma* semantics does not impose any real-world interpretation of task states, but the usual interpretation would be (loosely speaking) that a task is *dormant* if it has not been started, and it is not yet possible to say whether it will be started, *in_progress* if it has been started, *discarded* if the logic of the guideline implies either that it should not be started or that it should not be completed, and *completed* if it has been done.

Figure 7 illustrates the allowed transitions between task states. The reason there are transitions out of the *completed* and *discarded* states is that tasks may be cyclic, that is, that they may be enacted many times during enactment of a plan. The transition from *completed* to *in_progress* occurs when the task itself cycles, and the transitions from *completed* and *discarded* to *dormant* occur when its parent plan cycles.

The state transition diagram shown above is simpler than that described by Fox and Das,[33] which has 11 states. This is the result of a deliberate attempt to reduce the number of states to the minimum necessary to provide the required behavior while keeping the semantics of the language as simple as possible. Thus, statements about tasks that might be of interest to the user are not described as "states" if they can be *inferred* from the values of that task's existing properties.

The Abstract PRO*forma* Engine

The semantics of PRO*forma* are defined in terms of an *Abstract PROforma Engine*. An abstract PRO*forma* engine is responsible for the enactment of, at most, one guideline. Note that this does not make it impossible to enact two or more guidelines at the same time, it just means that we would have to describe these guidelines as being enacted in separate engines.

The state of an abstract engine is defined by the following four variables.

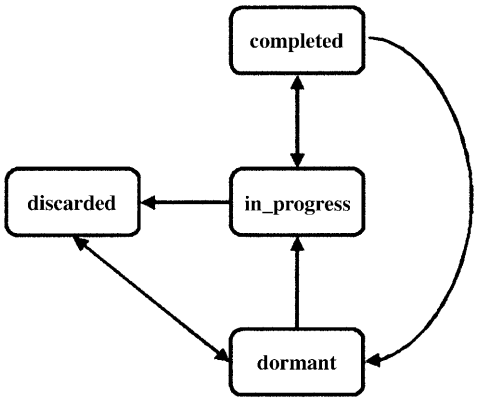

Figure 7. Task state transitions. An arrow between two states indicates that a transition between those states is possible. The circumstances in which the transition would actually occur are too complex to show in the diagram.

- The *Properties Table*. This is a three-column table containing the current values of all the properties of all the components in the guideline. Each row in the table is of the form *(C,P,V)* where *C* is a unique identifier for some component in the guideline, *P* is an identifier for some property of that component, and *V* is the current value for that property. The Properties Table cannot contain two rows with identical values for *C* and *P*.
- The *Changes Table*. This is a table containing new values that are to be assigned to properties of components as a result of operations that have been performed on the guideline. The rows of the Changes Table have a similar form to those of the Properties Table. Each row in the table is of the form (*C,P,V*) where *C* is a unique identifier for some component in the guideline, *P* is an identifier for some property of that component, and *V* is a new value that should be assigned to that property. The Changes Table may contain multiple rows with the same values for *C* and *P*. However, when the engine attempts to process these rows it will generally set the Exception flag (see below) to true.
- A logical flag *Exception*, which is true if an abnormal event has occurred in the processing of a guideline operation and false otherwise.
- A real number known as the *EngineTime*. The value of the EngineTime may only be changed by the Abstract Engine's *setEngineTime* operation (see Public Operations of the Abstract Engine). The semantics does not require the EngineTime to correspond to any measure of time in the real world. However, the Tallis implementation of PROforma attempts to make the EngineTime correspond as closely as possible to the number of milliseconds that have elapsed since midnight, January 1, 1970 UTC.

Figure 8 illustrates the state of an Engine running the guideline shown in Figure 4 at the point just after a candidate for the Referral Decision has been committed. The tasks named *No referral* and *Two week referral* are currently in the dormant state but will, respectively, enter the discarded and completed states once the changes in the Changes Table are enacted.

Public Operations of the Abstract Engine

The PRO*forma* specification defines a number of *public operations* that an external system may request the engine to perform. An operation may receive certain values as inputs, may yield certain values as outputs, and may change the contents of the Properties Table or Changes Table or the values of the Exception or EngineTime variables.

An implementation of the PRO*forma* engine should implement all of the public operations and should *not* allow external agents to change its state in ways that cannot be achieved by performing the above operations in some order.

The reason for imposing these restrictions is that they will enhance our ability to reason about what changes in the state of the engine can occur under what conditions. Table 1 gives informal descriptions of the engine's public operations. A more precise definition may be found in the Appendix.

Note that an implementation is free to implement any other operations as long as they do not change the state of the guideline. In general, an implementation should implement methods allowing complete read access to the Properties Table.

The Semantics of PRO*forma* Expressions

The PRO*forma* specification also defines the value of any expression when evaluated over any guideline. The value of an expression is defined recursively over the syntax of expressions. The full definition may be found on our website.[20]

It is an important feature of the evaluation of expressions that it is completely *side effect free*. By this we mean that evaluating an expression does not in any way change the state of the guideline over which it is evaluated.

Status Report

In a previous section we set out a number of criteria that the syntax and semantics of a guideline modeling language might be expected to meet. The current PRO*forma* specification meets these criteria in most respects. However, there are certain points that would merit further work.

Conformance with Existing Standards

During the period in which the PRO*forma* language and associated tools have been developed, a number of standards have emerged for the storage and interchange of structured information of various kinds. These include HL7 RIM,[34] XML,[35] RDF,[35] and DAML.[36] Currently, the specification of PRO*forma* does not make use of these standards. However, an

Properties Table

Component ID	Property	Value
C1	name	No referral
C2	name	Two week Referral
C1	state	dormant
C2	state	dormant
.	.	.
.	.	.

Changes Table

Component ID	Property	Value
C1	state	discarded
C2	state	in_progress
.	.	.
.	.	.

Exception = false
Engine Time=1024057142760

Figure 8. Example of engine state.

Table 1 ■ Public Operations

Operation	Description
loadGuideline	Load a guideline represented by a text string conforming to the PRO*forma* syntax.
evaluateExpression	Determine the value of a given PRO*forma* expression
addDataValue	Change the value of a given data item.
confirmTask	Inform the engine that a given task has been performed.
commitCandidate	Commit to a particular candidate or candidates of a decision.
sendTrigger	Send a trigger to the engine. Triggers provide a means of explicitly starting tasks without waiting for their scheduling constraints to be satisfied.
setEngineTime	Set the engine time. As mentioned above the semantics does not require the engine time to correspond to any measure of "real world" time, but implementations of the engine will in general establish some such correspondence.
runEngine	Informally, the effect of the runEngine operation is to update the Properties Table so as to reflect the consequences of any previous operations that have been performed. For instance, if a task has been confirmed then the runEngine operation may cause other tasks to enter the in_progress state as a result of their scheduling constraints being satisfied. A more precise definition of this operation is given in the appendix.

XML specification of PRO*forma* is being developed, and the latest generation of PRO*forma* software has been constructed in such a way as to minimize the effort required to adapt to emerging standards.

Precision and Unambiguity

The semantics of PRO*forma* provides an unambiguous definition of the response of the abstract engine when any of its public operations are performed. However, because it is abstract, the specification of the engine does not impose any restrictions on how data must be represented in a concrete implementation. For instance, it does not describe how many bytes should be used to represent integers or real numbers. It currently is unclear whether these decisions should form part of the PRO*forma* specification or whether they should be left to implementers of the language.

Discussion

The primary goal of defining the semantics of PRO*forma* is to facilitate wider discussion of the language and to allow it to be refined in light of the experience and different viewpoints of others in the medical informatics community. We see this as conferring a number of benefits, including building on the experience of others to improve the language and associated technologies; contributing experience with PRO*forma* to general discussions of clinical knowledge interchange formats and standards; and facilitating discussion of technologies for disseminating knowledge, particularly discussion of open source knowledge repositories.

Technical issues on which we seek input from the medical informatics community include, but are by no means limited to:

- Approaches to bringing PRO*forma* into line with relevant standards established by prominent standards organizations, such as HL7, the workflow management coalition, and others.
- Integration with other knowledge representation technologies, such as medical language and ontology representation systems.
- The definition of standard interfaces to external data and knowledge sources.
- Requirements for extending PRO*forma*'s expression language, e.g., the standardization of PRO*forma*'s first order logic features, and the introduction of operators allowing more flexible reference to guideline state.
- Improved temporal reasoning, including the ability to represent temporal constraints and abstractions about the state of tasks (e.g., when a task was completed or a data value was acquired) and the evolution of values, e.g., the ability to define temporal predicates such as "increasing" or "decreasing."[37]

As with other work on modelling clinical guidelines (e.g., Arden Syntax, GLIF) we see an important potential role of PRO*forma* as an interchange format for capturing guidelines and care pathways in a form that may be enacted on different software platforms or customized using different authoring and editing tools. We believe there is an indefinite number of different classes of application in which standard PRO*forma* guidelines or other applications may need to be edited and enacted in different ways for different environments. Examples from our own work include a project (REACT)[38] which is developing planning tools for customizing generic PRO*forma* guidelines into individualized patient care plans, the adaptation of cancer treatment protocols to permit different tasks within the protocol to be enacted by a number of distributed PRO*forma* agents,[39] and tools for integrating applications written in XML PRO*forma* with multimedia documents and other XML resources.[40]

Finally, we see the new generation of task-oriented guideline modeling technologies as a major vehicle for disseminating medical knowledge.[41] We believe that an important future development will be the creation of repositories of enactable guidelines, which complement the repositories of conventional (natural language) guidelines to be found at such sites as the National Guideline Clearing House in the United States and the National Electronic Library for Health in England. In effect, these will be repositories of "open source" knowledge of best clinical practice, which will permit guideline developers to download a published guideline and adapt it for local use.

This idea is being explored by the Institute for Knowledge Implementation based in Boston, MA <www.imki.org> and OpenClinical <www.openclinical.org> who are both developing repositories, focusing on rule-based representations at IMKI and task-based representations at OpenClinical. The two organizations are working together to explore methods for open source knowledge publishing in a format-independent manner, looking particularly at the need to develop editorial methods for this new form of knowledge publishing,

which support high standards of knowledge quality, operational safety, and ethical practice in the creation and dissemination of medical knowledge.[42,43]

We are strongly committed to making the results of this work available to others to use and to build on the experience of other communities in the development of PRO*forma* technology. For this purpose, the Tallis toolset (PRO*forma* and HTML authoring environment, enactment engine, and publishing package for making applications available over the Web) and documentation are available for research purposes and may be obtained by application to the authors.

References ■

1. Hripcsak G, Ludemann P, Pryor TA, Wigertz OB, Clayton PD. Rationale for the Arden Syntax. Comput Biomed Res. 1994;27:291–324.
2. Hripcsak G. Tutorial on how to use the Arden Syntax. Writing Arden Syntax medical logic modules. Comput Biol Med. 1994;24:331–63.
3. Miksch S, Shahar Y, Johnson P. Asbru: A task-specific, intention-based and time-oriented language for representing skeletal plans. In: Motta E, van Harmelen F, Pierret-Golbreih C, Filby I, Wijngaards NJE (eds). Proceedings of the 7th Workshop on Knowledge Engineering: Methods and Languages (KEML'97), Open University, Milton Keynes, January 22–24, 1997.
4. Shahar Y, Miksch S, Johnson P. The Asgaard project: a task-specific framework for the application and critiquing of time-oriented clinical guidelines. Artif Intell Med. 1998 Sep–Oct;14(1–2):29–51.
5. Musen MA, Tu SW, Das AK, Shahar Y. EON: A component-based approach to automation of protocol-directed therapy. J Am Med Inform Assoc. 1996;3:367–88.
6. Ohno-Machado L, Gennari JH, Murphy SN, et al. The guideline interchange format: a model for representing guidelines. J Am Med Inform Assoc. 1998;5:357–72.
7. Peleg M, Boxwala A, Ogunyemi O, et al. GLIF3: the evolution of a guideline representation format. Proc AMIA Symp. 2000:645–9.
8. Peleg M, Ogunyemi O, Tu S, et al. Using features of Arden Syntax with object-oriented medical data models for guideline modeling. Proc AMIA Symp. 2001:523–7.
9. Quaglini S, Stefanelli M, Cavallini A, Micieli G, Fassino C, Mossa C. Guideline-based careflow systems. Artif Intell Med. 2000; 20(1):5–22.
10. Dazzi L, Fassino C, Saracco R, Quaglini S, Stefanelli M. A patient workflow management system built on guidelines. Proc AMIA Annu Fall Symp. 1997:146–50.
11. Purves IN, Sugden B, Booth N, Sowerby M. The PRODIGY project—The iterative development of the release one model. Comput Methods Programs Biomed. 1997 Sep;54(1–2):59–67.
12. Johnson PD, Tu S, Booth N, Sugden B, Purves I. Using scenarios in chronic disease management guidelines for primary care. Proc AMIA Symp. 2000:389–93.
13. Fox J, Johns N, Rahmanzadeh A. Disseminating medical knowledge—The pro*forma* approach. Artif Intell Med. 1998;14:157–81.
14. Bury J, Fox J, Sutton D. The PRO*forma* guideline specification language: progress and prospects. Proceedings of the First European Workshop, Computer-based Support for Clinical Guidelines and Protocols (EWGLP 2000), 2000.
15. Peleg M, Tu S, Bury J, et al. Comparing computer-interpretable guideline models: a case study approach. J Am Med Inform Assoc. 2003;10:52–68.
16. The Workflow Reference Model. <www.wfmc.org>. Accessed July 11, 2003.
17. Balser M, Duelli C, Reif W. Formal semantics of Asbru. Technical Report. <http://www.protocure.org/Documents/Deliverables/D2-formal-semantics.pdf>. Accessed Sept 2002.
18. Aceto L, Fokkink W, Verhof C. Structural operational semantics. In: Bergstra JA, Ponse A, Smolka SA, (eds). Handbook of Process Algebra. Amsterdam: Elsevier, 2001.
19. Syntax and Semantics of PROforma. <http://www.acl.icnet.uk/lab/proformaspec.html>. Accessed July 11, 2003.
20. Reference deleted.
21. Walton RT, Gierl C, Yudkin P, Mistry H, Vessey MP, Fox J. Evaluation of computer support for prescribing (CAPSULE) using simulated cases. BMJ. 1997;315:791–5.
22. Emery J, Walton R, Murphy M, et al. Computer support for recording and interpreting family histories of breast and ovarian cancer in primary care: comparative study with simulated cases. BMJ. 2000;321:28–32.
23. Coulson A, Glasspool D, Fox J, Emery J. RAGS: a novel approach to computerised genetic risk assessment and decision support from pedigrees. Methods Inf Med. 2001;40:315–22.
24. Bury JP, Hurt C, Bateman C, et al. LISA: a clinical information and decision support system for collaborative care in childhood acute lymphoblastic leukaemia. Proc AMIA. 2002:988.
25. <http://www.openclinical.org/BMJDemo/jumpstart.htm?bmj,bmj.pf>. Accessed July 11, 2003.
26. <www.infermed.com>. Accessed July 11, 2003.
27. Kernighan BW, Ritchie DM. The C Programming Language (ed2). Englewood Cliffs, NJ: Prentice Hall, 1988.
28. Aho AV, Ullman JD. Principles of Compiler Design. Menlo Park, CA: Addison-esley, 1977.
29. Stoy J. Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. Cambridge, MA: MIT Press, 1977.
30. Hoare CAR. An axiomatic basis for computer programming. Comm of the ACM. 1969;12:576–80.
31. Peyton Jones SL. The Implementation of Functional Programming Languages. Englewood Cliffs, NJ: Prentice Hall, 1987.
32. Potter B, Sinclair J, Till D. An Introduction to Formal Specification and Z. Prentice Hall, 1996.
33. Fox J, Das S. Safe and Sound: Artificial Intelligence in Hazardous Applications. Cambridge, MA: MIT Press, 2000.
34. <http://www.hl7.org/library/data-model/RIM/modelpage_non.htm>. Accessed July 11, 2003.
35. <www.xml.org>. Accessed July 11, 2003.
36. <www.daml.org>. Accessed July 11, 2003.
37. Shahar Y, Musen MA. Knowledge-based temporal abstraction in clinical domains. Artifi Intell Med. 1996;8:267–98.
38. Glasspool DW, Fox J. REACT—a decision-support system for medical planning. Proc AMIA Symp. 2000:911.
39. Black E, Fox J. Breast cancer referral system with queuing strategies. ACL Technical Report no. 385, Advanced Computation Laboratory, Cancer Research UK, 2002.
40. Steele R, Fox J. Enhancing conventional web content with intelligent knowledge processing. ACL Technical Report no. 384, Advanced Computation Laboratory, Cancer Research UK, 2002.
41. Fox J, Johns N, Lyons C, et al. PRO*forma*: a general technology for clinical decision support systems. Comput Methods Programs Biomed. 1997 Sep;54(1–2):59–67.
42. Fox J, Thomson R. Clinical decision support systems: a discussion of quality, safety and legal liability issues. Proc AMIA Symp. 2002:265–9.
43. Fox J, Bury J, Humber M, Rahmanzadeh A, Thomson R. Publets: clinical judgement on the web. Proc AMIA Symp. 2001:179–83.

Appendix: Public Operations

In this appendix we give more detailed definitions of the public operations that were introduced in Public Operations of the Abstract Engine. For each operation we specify its input, its output, and the change that occurs in the engine state when it is performed. The purpose of this Appendix is to give an explanation of the semantics that is sufficiently detailed to show its most important properties.

- *loadGuideline.*
  - **Input:** a text string conforming to the syntax of PRO*forma* guidelines.
  - **Output:** none.
  - **Engine State:** The Properties Table is updated so as to contain those components defined by the input guideline. The Changes Table is set to empty. A fuller description of this operation explaining how the properties of components are initialized can be found on our website.[20]
- *evaluateExpression.*
  - **Input:** a text string conforming to the PRO*forma* expression syntax (see The Syntax of PRO*forma*).
  - **Output:** The value of the input expression, as defined by PRO*forma*'s expression semantics (see The Semantics of PRO*forma* Expressions).
  - **Engine State:** unchanged.
- *addDataValue.*
  - **Input:** a component identifier D, which must identify a data item, and a PRO*forma* value V, which must be a type correct value for that data item (see The Syntax of PRO*forma*).
  - **Output:** none.
  - **Engine State:** the operation adds row (D,value,V) to the Properties Table after removing any other row whose first two columns have the values D and value.
- *confirmTask.*
  - **Input:** a Component Identifier C, which must identify a task.
  - **Output:** none.
  - **Engine State:** the operation adds the row (C,confirmed,true) to the Properties Table after removing any other row whose first two columns have the values C and *confirmed*.
- *commitCandidate.*
  - **Input:** a Component Identifier D, which must identify a decision, and a nonempty list of component identifiers $\langle C_1, \ldots, C_n \rangle$ which all identify candidates of the decision D.
  - **Output:** none.
  - **Engine State:** the operation adds the row (D,result, $\langle C_1, \ldots, C_n \rangle$) to the Properties Table after removing any other row whose first two columns have the values D and result.
    It also adds the row (D,confirmed,true) to the Properties Table after removing any other row whose first two columns have the values D and confirmed.
- *sendTrigger.*
  - **Input:** a text string T.
  - **Output:** none.
  - **Engine State:** The engine state is updated as follows:
    For each component identifier C such that the Properties Table contains the row (C,trigger,T) add the row (C,trigger_active,*true*) to the Properties Table after removing any other row whose first two columns have the values C and trigger_active.
- *setEngineTime.*
  - **Input:** a real number V.
  - **Output:** none.
  - **Engine State:** the value of the EngineTime variable is set to V.
- *runEngine.*
  - **Input:** none.
  - **Output:** none.
  - **Engine State:** The Engine State is updated by performing the following operations:
    1. Perform the *burst* operation (see The Burst and Enactchanges Operations in this appendix)
    2. If the Changes Table is empty then stop, otherwise, perform the *enactChanges* operation (see The Burst and EnactChanges Operations in this appendix) and repeat from step 1.

    Informally, the effect of the runEngine operation is to update the Properties Table so as to reflect the consequences of any previous operations that have been performed. For instance, if a task has been confirmed, then the runEngine operation may cause other tasks to enter the in_progress state as a result of their scheduling constraints being satisfied.

    Burst and *enactChanges* are not public operations. That is, an implementation of PRO*forma* does not allow them to be performed at the direct request of an external system. In fact, a PRO*forma* implementation need not perform these operations at all, it simply has to ensure that the runEngine operation updates the engine state in the same way as it would *if* these operations were performed.

The Burst and EnactChanges Operations

The runEngine operation updates the Properties Table so as to reflect the consequences of any previous operations that have been performed. It is defined in terms of a number of *Private Operations*. A private operation is one that is introduced to set out the semantics of PRO*forma* but which a PRO*forma* implementation will not make available to external systems.

As described above, the public runEngine operation is defined in terms of the private operations burst and enactChanges. These operations are defined as follows.

- *burst.*
  - **Input:** none.
  - **Output:** none.
  - **Engine State:** the Engine State is updated as follows:
    1. Let $T_1, \ldots, T_n$ be the identifiers of all the tasks in the guideline.
    2. For $i = 1$ to n
       - Perform the *reviewTask* operation (See The Review Task Operation in this appendix) with identifier T_i as input.

    As we shall see in The Review Task Operation Section, *reviewTask* is defined in such a way that it does not matter in what order tasks are reviewed.

    Informally speaking, the burst operation examines each task in turn and determines whether any changes to the guideline are implied by the state of that task.

- *enactChanges.*
  - **Input:** none.
  - **Output:** none.
  - **Engine State:** The Engine state is updated as follows:
    1. Let *n* be the number of rows in the Changes Table.
    2. For *i = 1 to n*
       - Let (*C*,*P*,*V*) be the contents of the *i*th row in the Changes Table.
       - If the Changes Table contains a row (*C*,*P*,*V′*) where $V' \neq V$ then set the Exception flag to *true*.
         Otherwise add the row (*C*,*P*,*V*) to the Properties Table having first removed any other row whose first two columns have the values *C* and *P*.
    3. Empty the Changes Table.

The ReviewTask Operation

The *burst* operation is defined in terms of the operation *reviewTask*. The definition of *reviewTask* refers to the following private operations:

- *initialiseConditions*, *startConditions*, *discardConditions*, and *CompleteConditions* are operations in which input is a component identifier identifying a task, in which output is either *true* or *false*, and which do not change the state of the engine.

  We shall use the expression *initialiseConditions(C)* to denote the output from the *initialiseConditions* operation when it is passed the component identifier *C* as input. We shall use the same convention to denote the output from the other operations mentioned above.
- *initialise*, *start*, *discard*, and *complete* are operations in which input is a component identifier identifying a task, which have no output, and which may add rows to the Changes Table but may not change the Engine State in any other way.

The *reviewTask* operation then may be defined as follows:

- *reviewTask.*
  - **Input:** a Component Identifier *C* that identifies a task.
  - **Output:** none.
  - **Engine State:** The engine state is updated as follows:
    - If *initialiseConditions(C) = true* then perform the *initialise* operation with input *C*.
    - Else if *startConditions(C) = true* then perform the *start* operation with input *C*.
    - Else if *discardConditions(C) = true* then perform the *discard* operation with input *C*.
    - Else if *completeConditions(C) = true* then perform the *complete* operation with input *C*.

ReviewTask in More Detail

In this report we do not give complete descriptions of the private operations that are used in the definition of the *reviewTask* operation. There is not enough space for such definitions to be set out here and, in addition, it is these operations whose definitions are most likely to change as the PRO*forma* semantics is refined.

However, we will make the following remarks:

1. As one might expect, the *complete* operation with input *C* will add the row (C,state,completed) to the Changes Table. Similarly, the *initialise*, *start*, and *discard* operations, respectively, add the rows (*C*,state,dormant), (*C*,state, in_progress), and (*C*,state,discarded) to the Changes Table. Note that these are not the only rows added to the Changes Table by these operations.
2. The *initialise*, *start*, *discard*, and *complete* operations may alter the contents of the Changes Table but do not cause any other change in the engine state.
3. The *initialiseConditions*, *startConditions*, *discardConditions*, and *completeConditions* operations do not cause any change in the engine state.
4. The value output from the *initialiseConditions*, *startConditions*, *discardConditions*, and *completeConditions* operations is independent of the contents of the Changes Table.
5. When defining the *burst* operation, we asserted that it does not matter in what order tasks are input to the *reviewTask* method. We justify this assertion as follows:
   - Let *T1* and *T2* be component identifiers that identify tasks.
   - The operation *reviewTask* when performed with *T1* as input may alter the contents of the Changes Table but cannot alter the contents of the Properties Table. This follows directly from points 2 and 3 above and from the definition of the *reviewTask* operation.
   - The effects of the *reviewTask* operation when performed with task *T2* as input are independent of the contents of the Changes Table. This follows directly from point 4 above and from the definition of the *reviewTask* operation.
   - Consequently, the changes in the Engine State that result from applying *reviewTask* to *T1* can make no difference to what happens when *reviewTask* is applied to *T2*.
   - It can be seen readily from this argument that when performing the *burst* operation it does not matter in what order we apply *reviewTask* to the tasks in the guideline.

Motivation

The reason point 5 above matters is that if the effects of the *burst* operation did depend on the order in which tasks where passed to *reviewTask* then the definition we have given for *Burst* would be ambiguous. Furthermore, to resolve the difference, we would have to impose some sort of ordering on the tasks in the guideline, which would inevitably depend on some "trivial" property of those tasks such as the order in which they appear in the guideline text or the lexicographic ordering of their names. We would thus have departed from the criteria that we suggested for the semantics in the section on What Criteria Should the Syntax and Semantics Meet?

The observations leads us to suggest a criterion that we should observe when considering possible modifications to the detail of the PRO*forma* semantics:

We may (and frequently do) consider modifying the definitions of the *initialise*, *start*, *discard*, *complete*, *initialiseConditions*, *startConditions*, *discardConditions*, and *completeConditions* operations. When doing this we should take care to ensure that points 2, 3, and 4, and hence point 5 above remain true. In this way we will ensure that the semantics retains the desirable characteristics mentioned above.