Managing Standards Compliance

Wolfgang Emmerich, *Member, IEEE Computer Society*,
Anthony Finkelstein, *Member, IEEE Computer Society*, Carlo Montangero,
Stefano Antonelli, Stephen Armitage, *Member, IEEE Computer Society*, and Richard Stevens

Abstract—Software engineering standards determine practices that "compliant" software processes shall follow. Standards generally define practices in terms of constraints that must hold for documents. The document types identified by standards include typical development products, such as user requirements, and also process-oriented documents, such as progress reviews and management reports. The degree of standards compliance can be established by checking these documents against the constraints. It is neither practical nor desirable to enforce compliance at all points in the development process. Thus, compliance must be managed rather than imposed. We outline a model of standards and compliance and illustrate it with some examples. We give a brief account of the notations and method we have developed to support the use of the model and describe a support environment we have constructed. The principal contributions of our work are: the identification of the issue of standards compliance; the development of a model of standards and support for compliance management; the development of a formal model of product state with associated notation; a powerful policy scheme that triggers checks; a flexible and scalable compliance management view.

Index Terms—Software processes, software engineering standards, software development environments, compliance, consistency management.

1 Introduction

In this section, we outline the general problem of managing standards compliance in software development, motivate the development of automated support for this activity and describe the main elements of our approach.

1.1 Compliance

> "Standards are documented agreements containing technical specifications or other precise criteria to be used consistently as rules, guidelines, or definitions of characteristics, to ensure that materials, products, processes and services are fit for their purpose," [15].

Existing well-established software and systems engineering standards such as ISO 12207 [17], IEEE 1074 [14], and PSS-05 [23] set down the properties that both the process and its products must possess at given points in development. There is intense interest in adopting such standards in industry. This interest arises for a number of reasons: 1) as a means of transferring "good practice" in software engineering; as a result of the demands of clients or procurement agencies, 2) as a result of the demands of software process improvement (SPI) initiatives, ISO 9000 [16] certifications and ISO 15504 [19] trials, and 3) as a consequence of product certification requirements.

In each case, once a standard has been adopted it is important to manage compliance with the standard. By compliance we mean the extent to which software developers have acted in accordance with the "practices" set down in the standard. More narrowly we can think of this as consistency between the actual development process and the normative models embedded in the standard. The standards are both large and complex, and though they aspire to precision they are often incomplete and ambiguous. Determining the degree of compliance with specified practices, in particular as development progresses, is thus a challenging task. Compliance management is more difficult when you wish to use information about compliance to support remediation.

Significant resources are devoted to managing standards compliance. It is particularly critical in large systems engineering projects, such as in the defense, telecommunications and aerospace sectors. In such projects, much of the time of developers, managers, and quality assurance teams is occupied with identifying particular breaches in compliance and with tracking and managing the overall state of compliance of a project. Our treatment of this problem is thus strongly industrially motivated.

1.2 Approach

We take advantage of an important feature of the standards we have examined. They tend to express the requirements of the standard as constraints on the structure or contents of documents. Even the more "high-level" standards, such as the ISO 9000 series, are devoted to a considerable extent to requirements of this general form, though we have selected as a running case PSS-05, which is a particularly clear example.

- *W. Emmerich and A. Finkelstein are with the Department of Computer Science, University College London, London WC1E 6BT, UK. E-mail: {w.emmerich, a.finkelstein}@cs.ucl.ac.uk.*
- *C. Montangero is with the Dip. di Informatica, Università di Pisa, 56125 Pisa, Italy. E-mail: monta@di.unipi.it.*
- *S. Antonelli was with the Dip. di Elettronica e Informazione, Politecnico di Milano, P.za Leonardo da Vinci 32, 20133 Milano, Italy.*
- *S. Armitage and R. Stevens are with QSS Ltd., Northbrook House, Oxford Science Park, Oxford OX4 4GA, UK. E-mail: {steve.armitage, richard.stevens}@oxford.qss.co.uk.*

Manuscript received 15 Sept. 1997; revised 25 July, 1998.
Recommended for acceptance by C. Ghezzi.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 109064.

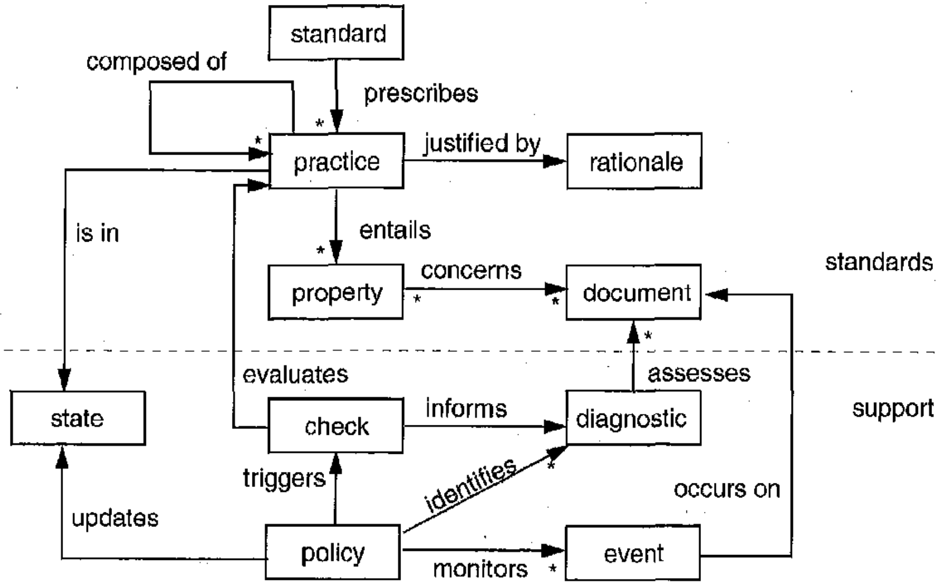


Fig. 1. Standards and compliance model.

Following from the standards themselves and from our experience with the development and use of software process technology [10], we adopt what might be termed a "tolerant" approach in which developers are free to organize for themselves the way they reach the goals set by project management. They are provided with ways to assess where they are with respect to their duties to conform to the practices. Policies set down the points at which different sorts of compliance should be established. Policies can however be overridden by an appropriately authorized developer who can postpone or even renounce compliance. We have a strong aversion to inflexible automated environments—early language based editors come to mind. We recognize that for significant periods of time developers leave work incomplete and inconsistent and that they may depart from normative practice for good reasons. Our approach assumes developers and managers are motivated to see the effective progress of the work.

It must be emphasized that our approach differs from conventional approaches to modeling software processes. We build on a starkly document-centered approach in which process is represented in the product and hence represented implicitly in the form of project plans and progress reports. We further assume that developers and managers are mutually committed to the maintenance of effective plans and reports. We argue strongly that this approach is entirely appropriate for most of the highly document-oriented industrial development processes with which we are familiar. The approach yields simple descriptions which are readily understandable by practitioners and amenable to inspection and improvement.

In our current work on compliance, we have focused on requirements management, and drawn our examples from this area. We have done so because it is a document-intensive activity of critical importance in software development. More significantly, because requirements processes cross-organizational boundaries, common standards and compliance play a particularly significant role. However, we believe that our findings are directly applicable to other stages of the software development process.

1.3 Outline

In Section 2, we describe our model of standards and compliance and illustrate this with an example. In Section 3, we outline notations used to specify the main elements of the model. In Section 4, we outline activities that are needed to formalize standard compliance using our notation. In Section 5, we describe a support environment that we have used to validate our approach. Section 6 sets out some important pieces of related work. In Section 7, we outline further work and conclude with a summary of our principal contributions.

2 Model

This section outlines the model of standards and compliance underlying our approach. Fig. 1 shows an entity-relationship diagram which summarizes the principal elements.

There are two parts to the model. The first, shown in the top part of the figure presents a simple view of standards and their use. The second, depicted in the bottom part of the figure, shows the main elements of our support for compliance management. In this section, a word in this font denotes entities or relationships in Fig. 1.

2.1 Standards

As discussed above, in order to express their requirements on the development process, software development standards tend to prescribe a number of practices to be followed. They usually leave ample room for tailoring of the actual processes, within the broad constraints they lay down. The distinction between mandatory and recommended practices, common to most standards, is one way of supporting this tailoring. For our purposes the distinction is irrelevant: We want to handle all the practices that the process owner demands compliance with.

PSS-05, for example, lists almost 200 practices, counting only mandatory practices. A typical practice taken from PSS-05 is the following:

> **UR04:** For incremental delivery, each user requirement shall include a measure of priority so that the developer can decide the production schedule.

Aside from the UR04 identifier, it is easy to recognize two parts to the practice: 1) a rationale: "so that the developer can decide the production schedule" and 2) a compliance requirement "for incremental delivery; each user requirement shall include a measure of priority."

Some standards, for example ISO-12207, distinguish between normative sections, which collect the practices and tend to exclude rationale in favor of conciseness and informative sections, which usually carry the rationale, albeit in an unstructured way. Since we assume that practices get into standards only after they have proven effective, we think it important, for user guidance, that practices are justified by a rationale which can motivate compliance: this suggests that the normative and informative sections be explicitly tied together. Standards, such as PSS-05 and ISO 12207, are both large and complex. They often prescribe several hundred practices. To cope with this complexity it is necessary to impose some structure on practices. The standard definitions are generally organized in a hierarchical manner, for instance according to different development stages or the document types that are to be produced. We reuse the structuring pattern of the standard to organize practices into a hierarchy. This leads to practices that are composed of other practices.

In PSS-05, for example, the practices related to the User Requirements Document (URD) can be subsumed in a composite practice. Apart from UR04 it contains another fifteen practices. Other composite practices in PSS-05 will be defined for the system requirements document, the architectural design document and so on.

A compliance requirement is an intrinsic part of any practice, and in many cases, as in UR04, it entails a given predicate on the product of the process that shall hold at some point. We highlight the static facet of a practice in the model, the property of interest. It may be convenient to break down a practice that we find in a standard into several properties. In our example we have the property:

> For incremental delivery, each user requirement includes a measure of priority.

We aim to provide support to the user to assess the current state of compliance with respect to this property. Some careful reading of the standard allows us to discover that the property entailed by UR04 concerns a specific document, namely the URD.

It should be noted that not all the practices obviously define compliance requirements with respect to the product. For instance, UR10 states:

> **UR10:** An output of the User Requirements phase shall be the URD.

This is, on the face of it, a constraint on the process. We believe that these constraints can be readily expressed as constraints on the product, by considering with more care those management documents, such as project plans and progress reports, that capture the essential features of the dynamics of the process. These documents, which actually constitute a large proportion of the documents produced during software development, have up to now received little attention in research on software process support and, on process technology in general.

As an example, UR10 might entail the following property:

> The Software Project Management Plan for the User Requirements phase includes a task or work package for the construction of the URD.

A similar argument applies to the conditional clause in UR04—for incremental delivery. This condition on the state of the process, which relates to the overall strategy of the project in PSS-05, can be transformed into a condition on the product, in a straightforward manner: the general description of the project, in the Software Project Management Plan (SPMP), shall include a "project mode" attribute, which may take as value, among others, "incremental" delivery. The property in UR04 then becomes:

> If the project mode in the project general description in the SPMP is "incremental" delivery, each user requirement in the URD shall include a measure of priority.

Once the properties that the practices entail have been characterized in terms of document states, they can be formalized to define a compliant process. So, we can characterize compliance precisely, with respect to the process state as it is embedded in a formal model of the product states. We are less concerned with modeling the dynamics of the process. However, the product needs to evolve, to reach a compliant state, and we capture this evolution by considering the actions that occur on documents, and may affect the value of a property. Our characterization of actions will be limited to what is needed to monitor them so as to advise the user about compliance before some critical step is performed.

Before considering the bottom part of Fig. 1, it should be clear that not all the relations in Fig. 1 are one-to-one: A standard usually recommends many practices and some practices may entail several properties. Properties may be defined that need to access information included in more than one document. Obviously, a document may participate in more than one practice, and is, therefore, required to satisfy many properties. Similarly, a composite practice may be composed of several component practices.

2.2 Support

The basic mechanism to support the user is the check, which evaluates a practice and identifies those elements of the documents which are noncompliant and the properties to which they fail to comply. In the case of UR04, this is the list of the requirements for which priority is undefined. Clearly, this is not always the most helpful information that could be provided. A check, therefore, informs a diagnostic, which could for example produce the percentage of the noncompliant document elements or a traversal which allows the relevant document elements to be accessed. This information would allow the engineer to assess the importance and the difficulty of making the document compliant. They may also indicate the range of possible repairs that can be performed. Though in line with our approach, we do not compute an exhaustive list of corrective actions. Such diagnostics should be provided by "canned" functions.

Even the best motivated user may fail to apply all the checks that are needed before some sensitive action, such as baselining. Also, given the scale and complexity of the practices, they may be uncertain of the best points to establish compliance. To ensure that no unintended breach of compliance occurs, we introduce policies that trigger the appropriate checks whenever some event or pattern of events occur. In other words, policies monitor events. An event occurs on a document when there is an attempt to perform an action on that document. We assume that users perform one action on one document at any one time, events occur on exactly one document. Events can be detected at any level of granularity. They can be distinguished for a document as a whole as well as for paragraphs and even individual attributes of paragraphs.

Policies have a mode that designates the extent of freedom to breach the compliance requirement. In each of the cases below, the user attempts to perform an action thereby generating an event. On the detection of this event:

- in the error mode, the check is immediately executed and the failure of the check prevents the action from being completed, in which case the problem should be fixed using the diagnostic as support;
- in the warning mode, the check is immediately executed and the failure of the check provides the user with the diagnostic but the user is permitted to perform the action and knowingly become noncompliant;
- in the guideline mode, the user is informed that it is advisable to execute a check but allows the user to perform the action, without executing the check if desired.

The most useful mode, given our tolerant approach, is the warning mode. The others open the door to more varied compliance management: for example, besides providing strict compliance enforcement, the error mode might be useful when the fix is so simple that there is no point in letting the breach occur, and the guideline mode allows the introduction of discretionary practices. In practice, we have found that developers wish to know when it is advisable to perform a check, but find the execution of the check disruptive or are already aware of compliance problems.

Different diagnostics may be appropriate in different circumstances. We, therefore, allow the diagnostic to be identified as part of the policy. For example, a policy in guideline mode may most appropriately be accompanied by a statistical diagnostic while a policy in warning mode may have a traversal diagnostic associated with it.

A practice can be in a state other than simply compliant or noncompliant. These states are displayed in the Statechart [12] in Fig. 2, which identifies composite states, such as defined and not checked, that subsume more primitive states, such as not required and unsafe.

For any standard, not all the practices are likely to be formally defined. This may be because of difficulties in the formalism, customization, or evolution of the standard. A defined practice can be in one of two states: checked and not checked. For a practice that is not checked, we distinguish whether the check is not required *at this point in the development*, from an unsafe state where a guideline to perform a check has been overridden by the developer. The execution of a policy or the manual execution of a check updates the state.

It is desirable to identify states for composite practices, too. These states are useful for providing a high-level perspective on how compliant the project is. They can be used to identify paths to hot-spots of noncompliant atomic practices that need the attention of developers. We, therefore, define the state of a composite practice in such a way that it is equal to the state of the component practice that requires most attention. Hence, we define an order between states:

compliant < check not required < undefined < unsafe < noncompliant

The state of a composite practice is then $max(i_1, \dots, i_n)$ where i_j are the states of the component practices. For example, if three of the practices relating to the URD are compliant, three are not required, three are undefined, three are unsafe, and three are noncompliant; the overall state of the composite practice would be noncompliant.

Notations are needed in order to specify the structure of documents, properties, practices, and their composition, policies, and events. These notations will be introduced in the next section.

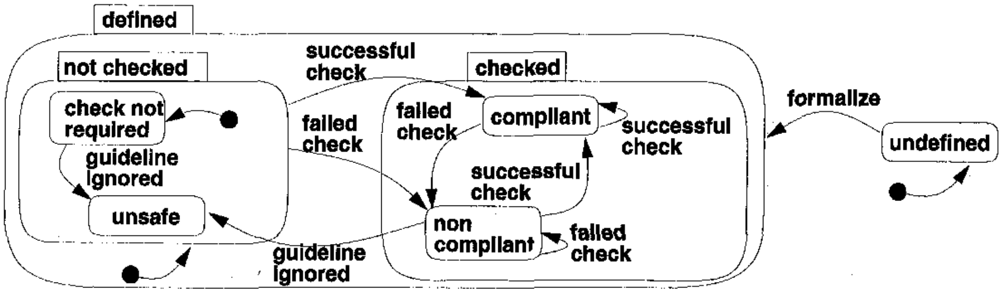


Fig. 2. States of practices.

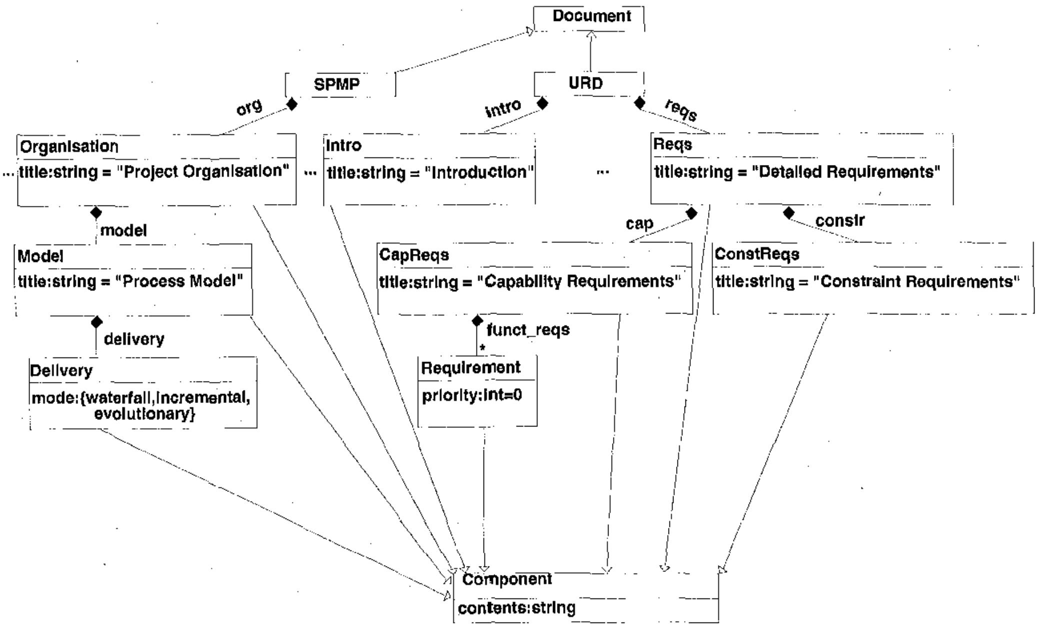

Document
SPMP
URD
org
intro
reqs
Organisation
title:string = "Project Organisation"
Intro
title:string = "Introduction"
Reqs
title:string = "Detailed Requirements"
...
model
cap
constr
Model
title:string = "Process Model"
CapReqs
title:string = "Capability Requirements"
ConstReqs
title:string = "Constraint Requirements"
delivery
funct_reqs
*
Delivery
mode:{waterfall,incremental, evolutionary}
Requirement
priority:int=0
Component
contents:string

Fig. 3. Document schema specification in UML.

3 Notations

3.1 Documents

The specification of properties is based on the structure of the underlying documents. The formalization of UR04 will assume, for instance, that paragraphs in the URD that state functional requirements have an attribute to which priorities are attached. It also assumes that the paragraph identifying the delivery mode in the SPMP has an attribute that expresses whether or not the delivery is incremental. As these assumptions are specific to a given standard, or to a company specific customization of a standard, the need arises to specify a schema for the underlying document structure.

This document schema specification serves various purposes. Standards provide a definition of the structure of documents. The document schema specification elaborates and formalizes these definitions so that properties can be checked against them. It is also used for creating instances of documents as templates that users of the support environment can then fill. It is exploited for the generation of checks as to whether the documents continue to conform to the type structures that are set down in the schema as development proceeds and changes are introduced.

We use a subset of class diagrams as specified in the Unified Modeling Language (UML) [26] for the document definition. Classes are used to model the documents and their components, such as sections, subsections and paragraphs. Attributes of classes model values we want to attach to components. Aggregation relationships are used to specify the decomposition of documents into components. Associations model links that exist between different components. Association classes are used to model the attributes of these links.

We assume that a number of classes are predefined. Among those are classes Document and Component. Document determines the common properties of a document, such as attributes for the document owner, the last modification date, the current version number and so on. Component determines common properties of any section, subsection, or paragraph.

Fig. 3 provides an example. It displays an excerpt of the document schema specification for PSS-05. The document schema includes two types of documents, URD and SPMP. Both document types contain an aggregation of sections, subsections, and paragraphs, that is partially displayed. The aggregation hierarchy was derived straightforwardly from the appendix of PSS-05 that gives "templates" for the different documents to be produced. We have added attributes to the component types taken from these templates. Let us now focus on type Requirement, instances of which will be used to define the users functional requirements, and type Delivery, instances of which define the delivery mode in the project management plan. An attribute priority was added to the type for Requirements and an attribute mode was added to the type for Delivery.

UML attribute initializations and aggregation relationships are used to specify the creation of document instances. They formalize the instance level of abstraction of the document templates defined in standards. In the example of UR04 given in Fig. 3, initializations define section titles that are assigned to title attributes as soon as section and subsections are created. The aggregation relationships are exploited to propagate the creation of components upon creation of a composite. For a section of type Reqs we know due to the aggregation relationship that two subsections should be created for the "Capability

Requirements" (cap) and "Constraint Requirements" (constr).

3.2 Properties and Practices

We use first-order logic in order to specify properties. The vocabulary that is used to form these logical expressions are operations of predefined attribute types, relationships, and attributes of document or component types identified in the document schema, names of instances of document types, operations of the Boolean algebra and universal and existential quantifiers.

Classes in the document schema have a type. The class type determines the attributes that instances of the type will have. Attributes also have a type. While we need to be able to define new classes, we can restrict ourselves to a limited set of attribute types that we can pre-define. This is because we only need to define type structures for software engineering document components rather than general-purpose objects. Hence, we predefine a number of attribute types, including Boolean, char, int, real, string, and enum. Each of these types have a number of straightforward predicates and functions, which can be used in expressions for the definition of properties.

As an example, consider attribute priority, which is of type int. We assume that the priority increases with the value of this attribute. We use 0 to indicate an undefined priority. Hence, the formalization of UR04 will have to compare values of requirement's priorities with 0. For that purpose we use the $\neq$ operator that is defined for type int.

One of the main purposes for defining the document schema above is to be able to make assumptions about the structure of documents when defining properties; the property specification language must be able to refer to, and use concepts of, the document schema. We now introduce a notation for access to attributes and traversing along relationships defined in the document schema. If a type t includes an attribute a we specify access to attribute a for an instance i of type t as i.a. The result of that expression is a value in the domain of the type of the attribute. Likewise, if the type t has a relationship r we denote traversal along the relationship as i.r. The result of that expression is a component of the type at the other end of the relationship (if the other relationship is 1:1) or a set of components of that type if the other end of the relationship is of cardinality many. Relationship traversals can be concatenated into path expressions formed by the relationship names delimited by ".". Only the last item in such path expressions may be an attribute name.

As an example, d.org.model.delivery.mode denotes the value of the mode attribute of a component of type Delivery that is included in software management plan document identified by constant d, where d is an instance of SPMP.

The document schema specification is at a type-level of abstraction. Attribute accesses, traversal along relationships and operations of predefined attribute types are at type-level, too. In order to determine whether or not a property holds we need to look at particular instances of documents and entities. We can denote instances either by quantifying over the universe of all instances of a particular type or by referring to named instances. Universal and existential quantifiers can be used for the former, but we need to introduce a notation for the latter. Standards generally limit the number of documents to be produced in a project. We assume that each document has a name and we allow these names to occur in formulae. Their names and types are declared at the beginning of the specification. The name for the user requirements document is introduced by urd:URD. Now we are in a position to specify the property UR04p1 entailed by practice UR04:

$$
\begin{aligned}
&\texttt{spmp} : \texttt{SPMP};\\
&\texttt{urd} : \texttt{URD};\\
&\texttt{UR04p1} :=\\
&\bigwedge\nolimits_{r \in urd.reqs.cap.funct\_reqs}\\
&(\texttt{spmp.org.model.delivery.mode} = \texttt{incremental})\\
&\Rightarrow \texttt{r.priority} \neq 0
\end{aligned}
$$

It should be relatively straightforward to define a static semantics for the first-order language sketched above. This static semantics would rely on the type system induced by the document schema and support consistency checks of the property specification. We could for instance detect the use of operations that are unavailable for an attribute type, the traversal along undefined relationships or the use of attributes that are undefined for a document or component type.

One might argue that first-order logic is insufficient to express compliance to standards that specify how activities should be ordered in time. We believe that we do not need the expressive power of temporal logic, at the level of checks, as the standards we have looked at assume that proper records are kept in project management reports about the temporal order of activities. This seems an entirely reasonable assumption. The structure needed for these records is expressed in the document schema and the primitives outlined above are appropriate to use this structure to specify properties. We do, however, need the expressive power of first-order logic as we have to use universal and existential quantifiers for relationships of cardinality *many* in property specifications and for properties that must hold for all, or at least one, instance of a type.

Practices are conjunctions of one or more properties. They are specified as a structured document by defining the practice identifier and enumerating all the properties that are part of it. For a composite practice, we give its name and enumerate all its component practices. We have not defined a notation for rationale. Each practice has associated with it a short piece of natural language text.

3.3 Policies and Events

Policies determine when practices are checked, the relevance of the result and the diagnostic provided to the user. A policy is given by a quadruple (E, P, M, D) where E is an event, P is a practice identifier, $\texttt{M} \in \{\texttt{ERROR, WARNING, GUIDELINE}\}$ identifies the policy mode and D identifies a canned diagnostic function. We have implemented three such functions in our prototype: LIST generates a list of the noncompliant items; STAT generates a simple statistical analysis (number, percentage) of the noncompliant items; TRAV generates a traversal of the underlying document base so as to retrieve a filtered document containing all the noncompliant items.

For a full appraisal of the expressive power of our policy language, we need to discuss the specification of events. Policies trigger checks on the occurrence of certain events recognizd for a document, a component or a component attribute.

We have found that the events which feature most frequently in policies are:

- `Open(<document>)`
- `Close(<document>)`
- `Update(<attribute>)`

The `Open` event is issued if the user is about to open a document. The document is identified by a constant in the same way as it was identified for properties. The `Close` event is issued if the user is about to close a document. Finally, the `Update` event is issued if the user is about to modify the value of a component's attribute. Attributes are identified by path expressions.

Let us revisit UR04 and look at some examples. Consider the policy:

```
(Update(spmp.org.model.delivery.mode),
 UR04, WARNING,STAT)
```

This policy might result in warning users about noncompliance of the user requirements document after the delivery mode attribute of a project plan was edited and shows as a diagnostic the percentage of noncompliant requirements. Another example is:

```
(Open(ddd), UR04, ERROR, TRAV).
```

It determines that users cannot work on the detailed design document (identified by constant ddd) if in an incremental delivery, the priority has not been specified. The given diagnostic is a traversal that enables the user to visit all noncompliant requirements.

While the atomic events given above are necessary for the specification of policies they are not sufficient for every policy that users might find appropriate. It is necessary to compose events, for example to subsume different events that all trigger the same check.

Event composition is a feature of FLEA, a formal language for expressing assumptions [8]. FLEA includes triggers with operators for temporal and logical event composition. We use FLEA's logical OR operator to express that a check will be triggered when either of the combined events is raised. An example is the following policy:

```
(Open(add) OR Open(ddd), UR04, ERROR, LIST)
```

It determines in just one policy that users can work on neither the architecture definition document (add) nor the detailed design document (ddd) if the project is noncompliant to UR04. As a diagnostic it provides the list of noncompliant requirements.

The temporal event composition operators of FLEA are THEN, THEN-EXCLUDING, IN-TIME, and TOO-LATE. THEN composes two events. The combined event is raised if the two events are executed after each other. IN-TIME raises the composite event if the second of the two events occurs within a specified period of time starting from the occurrence of the first event. Both, THEN and IN-TIME can be seen as temporal extensions of a logical AND operator. A logical AND cannot be applied for the composition of events because in both our model and in FLEA's triggers only one event can be detected at a specific point in time. THEN-EXCLUDING and TOO-LATE are temporal versions of the logical NOT operator. THEN-EXCLUDING combines three events and it is raised if the third event is not raised between the first and the second event. TOO-LATE raises the composite event if the second of the two events specified does not occur within a period of time starting from the occurrence of the first event. As an example of a temporal event composition consider:

```
(Update(spmp.org.model.delivery.mode)
THEN Open(urd),UR04, WARNING, STAT)
```

It determines that the user should be provided with a statistical analysis of noncompliant requirements if the delivery mode attribute has been edited and the user is about to open the URD.

We can now outline the semantics of policies. Each policy references a practice. If the policy is in guideline mode, the user is advised to execute the check. If the user declines to execute the check then the practice will be in state unsafe. For policies with warning and error mode, the check is executed transparently to the user. If the check passes, the practice will be in state compliant. If a check of a policy in error mode fails the diagnostics associated with the policy will be given and the action that triggered the event is aborted. In warning mode, the diagnostic is given to the user and the user can abort the action. If the user does not abort, the practice will be in state noncompliant. The state of all composite practices in which the checked practice is included will be recomputed when the state of the practice has changed.

Policies reference exactly one practice. If different practices have to be checked when an event occurs, different policies have to reference that event. They will then all be triggered when the event occurs. It is not possible to define more than one policy for a practice. If we had a policy in warning mode and another in guideline mode for the same practice, users would be confused when they are first given a guideline and then a warning. Hence, the static semantics of our policy definition language excludes these situations.

4 METHOD FOR DEFINING COMPLIANCE

Defining standards compliance is a complex activity. In this section, we outline a method that supports the systematic definition of standards compliance using the notation that we introduced in the previous section.

Fig. 4 shows a high-level Petri net that indicates the activities that constitute our method for defining standards compliance. The activities use information provided by standards, namely document templates, properties and practices, and policy statements, and produce a validated compliance definition. We now discuss each of these activities.

The first activity of our method is the definition of the UML class diagram. The class diagram is derived from templates for documents that are included in most

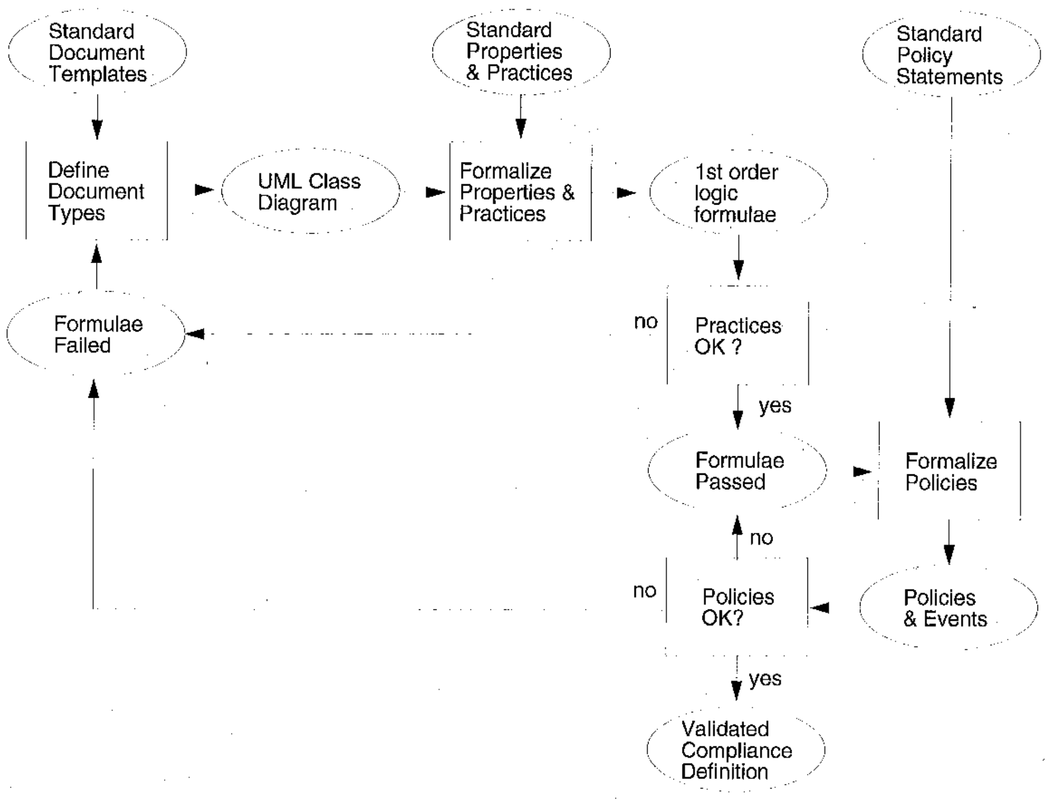


Fig. 4. Method for defining standard compliance.

standards. The appendix of ESA's PSS-05 includes 11 such templates. The IEEE standards 730, 828, 829, 830, 1012, 1016, 1058.1, and 1063 are further examples of standardized document templates. The templates provide suggestions for the chapters, sections, and subsection contents of documents. It is very straightforward to translate these into a set of classes that are interconnected through composition relationships. These composition relationships should be given expressive names that will be later used in path expressions.

The templates are translated into a considerable number of classes. To cope with the complexity involved, we suggest using the UML concept of packages for structuring the overall class diagram. We believe a template should be mapped into a single package. If necessary, nested packages should be used.

Based on the UML class diagram, path expression can be defined. In addition, attributes are needed. These should be added to the UML class diagram as appropriate. Developing the UML class diagram and the formalization of practices are an incremental and intertwined activity. In Fig. 4, this is suggested by the feedback cycle that leads to the document type definition activity.

If all properties of a (potentially composite) practice are formalized the practice can be tested. This should be supported by an environment in such a way that a check derived from a practice can be triggered manually. The environment should enable the instantiation of the classes identified in the document schema in documents. The formulae should then be interpreted by the environment in order to execute checks. By executing the check on different document test cases, the formulae that formalize practices and properties can be validated.

Standards include practices that lead to the adoption of policies. PSS-05 for example, includes a practice that dictates how compliance to practices should be checked. In PSS-05, compliance to practices of the phase should be established when the document produced in that phase is reviewed and noncompliance should be brought to the attention of management. Such policy statements will then be formalized by determining a policy mode, a practice identifier, a canned diagnostic and an event.

The support environment should support the incremental introduction of policies. The impact of introducing a new policy can then be tested incrementally. Moreover, this supports introductions of policies-on-the fly. The introduction of new policies, however, should be confined to authorized users.

5 Support Environment

In a project with many documents, evolving over a significant period of time and hence with a very large number of checks to be carried out, a support environment is needed that checks compliance, presents diagnostics and provides a means of obtaining an overall view of the current state of compliance. In this section, we describe the architecture of our support environment and the current status of our implementation.

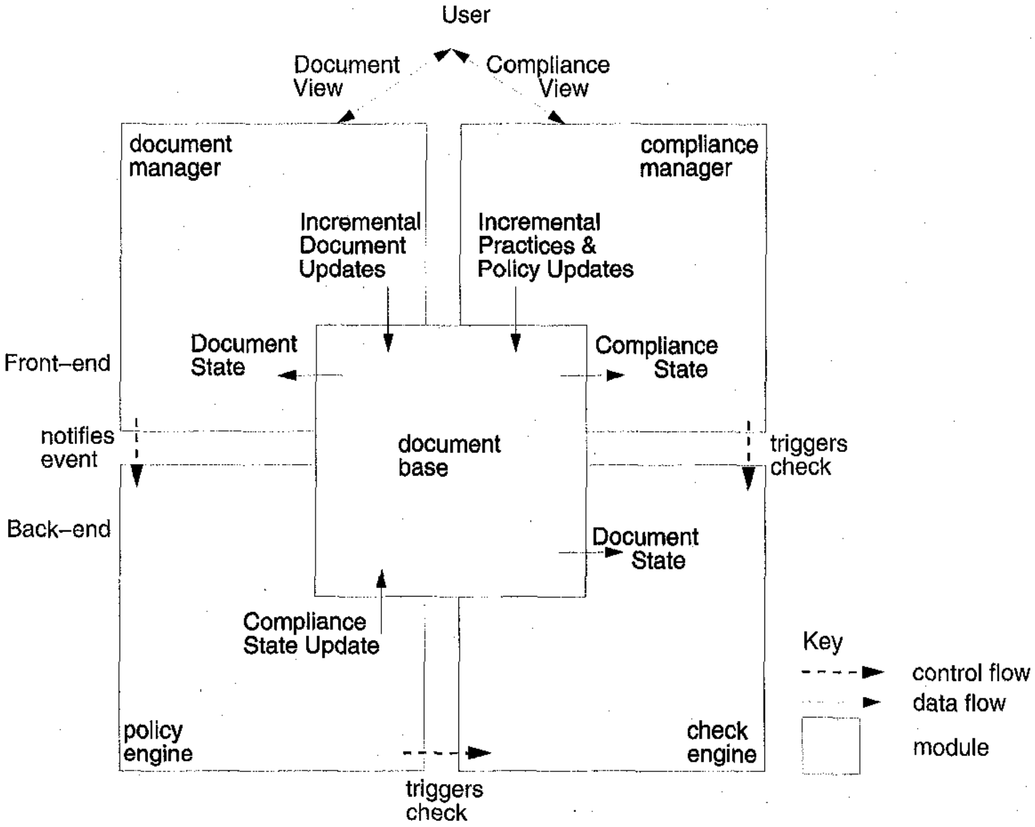


Fig. 5. Logical architecture of the support environment.

5.1 Architecture

The logical structure of our support environment is shown in Fig. 5. It consists of four main modules: a document manager and a compliance manager, comprising the front-end of the environment; and policy and check engines, comprising the back-end. These are integrated by way of a shared document base.

5.1.1 Document Manager

This module is a generic document management system with all the associated features such as navigation, folding and unfolding and so on.

The managed documents are hierarchically composed. Components are used to store information for sections, subsections, down to individual paragraphs (components). Every component in a document has attributes. Users can attach attributes to objects by defining a name and a type during editing sessions and from then create and/or display values of these attributes. The document manager also supports the concepts of links that can be used to relate one object to another. Links are used, for instance, in order to capture requirements traceability information.

Fig. 6 shows the document manager displaying a PSS-05 Systems Requirements Document.

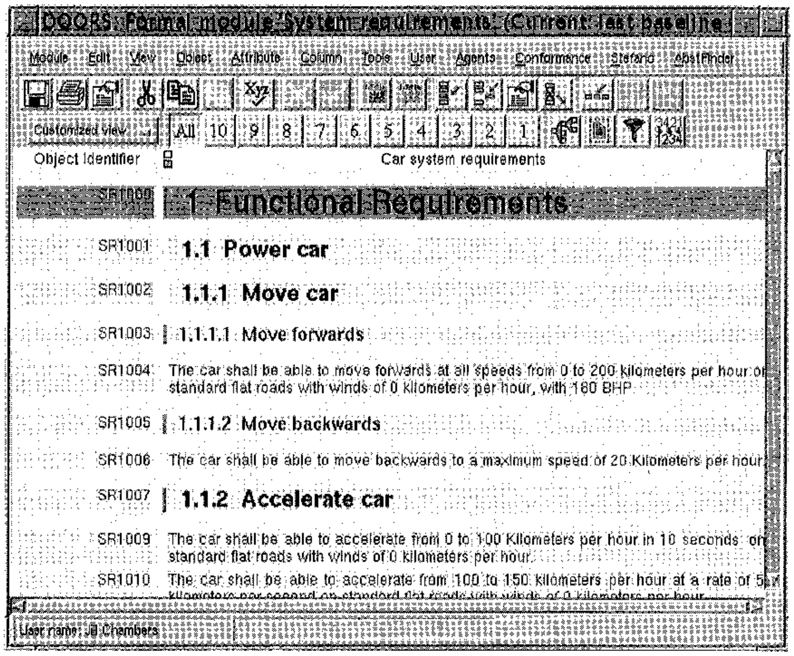
5.1.2 Compliance Manager

A standard is itself treated as a hierarchical document. PSS-05, which is a well organized standard, is divided into practices associated with process management and practices associated with products.

The compliance manager is based on a view of a hierarchical document as shown in Fig. 7. The practices with their associated rationale can be written within the compliance manager using the same editing facilities available for other structured documents. They can be viewed at any stage. The compliance manager provides a simple facility for manually triggering the checks associated with a particular practice. A separate document is provided to write the policies.

The practice states (Compliant, Not Required, Undefined, Unsafe, Noncompliant) described above are associated with a color. The color coding allows the manager or developer to understand the compliance of the project at a glance. Our scheme for providing a high level view of compliance, that is propagating the "worst state" up the tree, clearly fits with this approach. The overall state of compliance of a project with respect to a standard can be readily viewed at any level, with the tree folded, and more detail can be obtained by unfolding where there are obvious problems. Nodes can be opened in order to view the diagnostics for a noncompliant practice and for an unsafe practice information about the guideline.

Fig. 8 shows the practices from PSS-05 viewed as a textual document. In this figure, you can only see the natural language formulation, though the formalized properties can be viewed in an identical manner, as below



Object Identifier	Car system requirements
SR1000	1 Functional Requirements
SR1001	1.1 Power car
SR1002	1.1.1 Move car
SR1003	1.1.1.1 Move forwards
SR1004	The car shall be able to move forwards at all speeds from 0 to 200 kilometers per hour on standard flat roads with winds of 0 kilometers per hour, with 180 BHP
SR1005	1.1.1.2 Move backwards
SR1006	The car shall be able to move backwards to a maximum speed of 20 Kilometers per hour
SR1007	1.1.2 Accelerate car
SR1009	The car shall be able to accelerate from 0 to 100 Kilometers per hour in 10 seconds on standard flat roads with winds of 0 kilometers per hour.
SR1010	The car shall be able to accelerate from 100 to 150 kilometers per hour at a rate of 5 kilometers per second on standard flat roads with winds of 0 kilometers per hour.

Fig. 6. PSS-05 systems requirements document.

in Fig. 9. The shading in Figs. 7 and 8 denote the current practice state. The color key is given in the lower left corner of Fig. 7.

Fig. 9 shows a view of the properties that can be obtained from the compliance manager. Fig. 10 shows the crude diagnostics currently given by the compliance manager. The displayed diagnostic is the result of the policy discussed in Section 4, which demanded a statistical diagnostic. It identifies the policy through the composite event that triggered the check and the policy mode. It also indicates the practice that has been checked and the percentage of noncompliant components.

5.1.3 Policy and Check Engines

The document manager notifies the policy engine about the occurrence of events on documents and components. The policy engine monitors events and triggers the check engine. The check engine performs the check by evaluating the constituent properties and returns the results to the policy engine, which in turn updates the document holding the practice states, so that the compliance manager can show them to the user.

The section which follows explains how the front-end and back-end are implemented and communicate in the prototype.

5.2 Implementation

Rather than implementing an environment from scratch we are using and extending an existing system. We have chosen the Dynamic Object Oriented Requirements System (DOORS) [25]. It is widely used in industry to manage requirements and management documents that are produced during system engineering processes. DOORS has no process or work-flow engine. DOORS has a large user base with an expressed interest in problems of compliance.

Fig. 11 shows the physical architecture of the prototype support environment. The major elements are DOORS, FLEA, and AP5 [3]. They communicate through a set of files which are discussed in Section 5.2.4.

As shown in Fig. 11, we are exploiting FLEA not only for policy specification, but also in our implementation. We take advantage of AP5, the infrastructure on which FLEA is built, as a basis for implementing our check engine. The subsections which follow highlight the important facets of the three major implementation modules and their integration.

5.2.1 DOORS

From an implementation standpoint DOORS has powerful extension facilities that allow us to build relatively complex application layers and provides powerful and rapid data access. DOORS has a Dynamic eXtension Language (DXL) that can be used to automate tasks. DXL is an interpreted language. It includes imperative and rule-based language concepts. DXL functions can be attached to user interface primitives, such as pull-down menus. Functions are currently used to create template documents, whose structure and attributes correspond to those prescribed by certain standards. DXL provides control flow primitives, such as iterations, and simple means of attribute accesses and traversal across links.

DXL also provides the concept of triggers. Triggers are associations between events and actions. Triggers can be used to react to the occurrence of the event (posttriggers), or to guard the event which can then, if necessary be aborted (pretriggers). We have used pretriggers and associated aborts extensively as they allow us to prevent the developer performing actions forbidden by current policies.

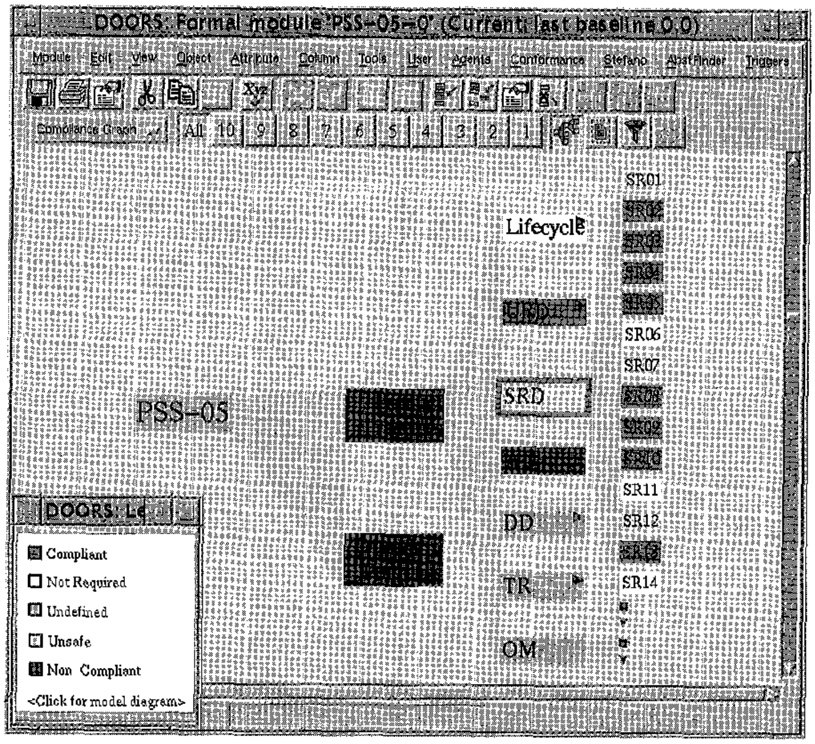
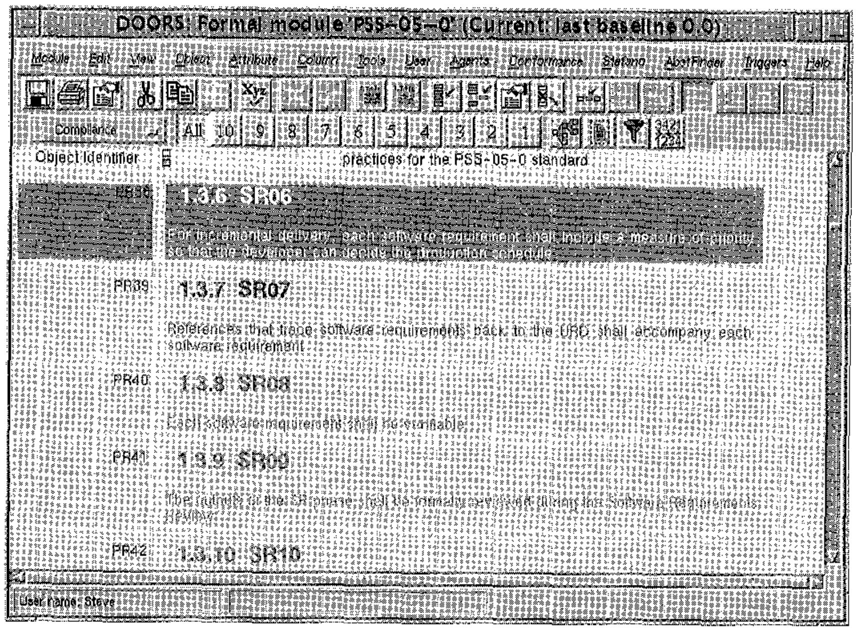


Fig. 7. PSS-05 in the compliance manager—hierarchical view.



Object Identifier | practices for the PSS-05-0 standard

PR38 **1.3.6 SR06**

For incremental delivery, each software requirement shall include a measure of priority so that the developer can decide the production schedule.

PR39 **1.3.7 SR07**

References that trace software requirements back to the URD shall accompany each software requirement

PR40 **1.3.8 SR08**

Each software requirement shall be verifiable

PR41 **1.3.9 SR09**

The outputs of the SR phase shall be formally reviewed during the Software Requirements Review

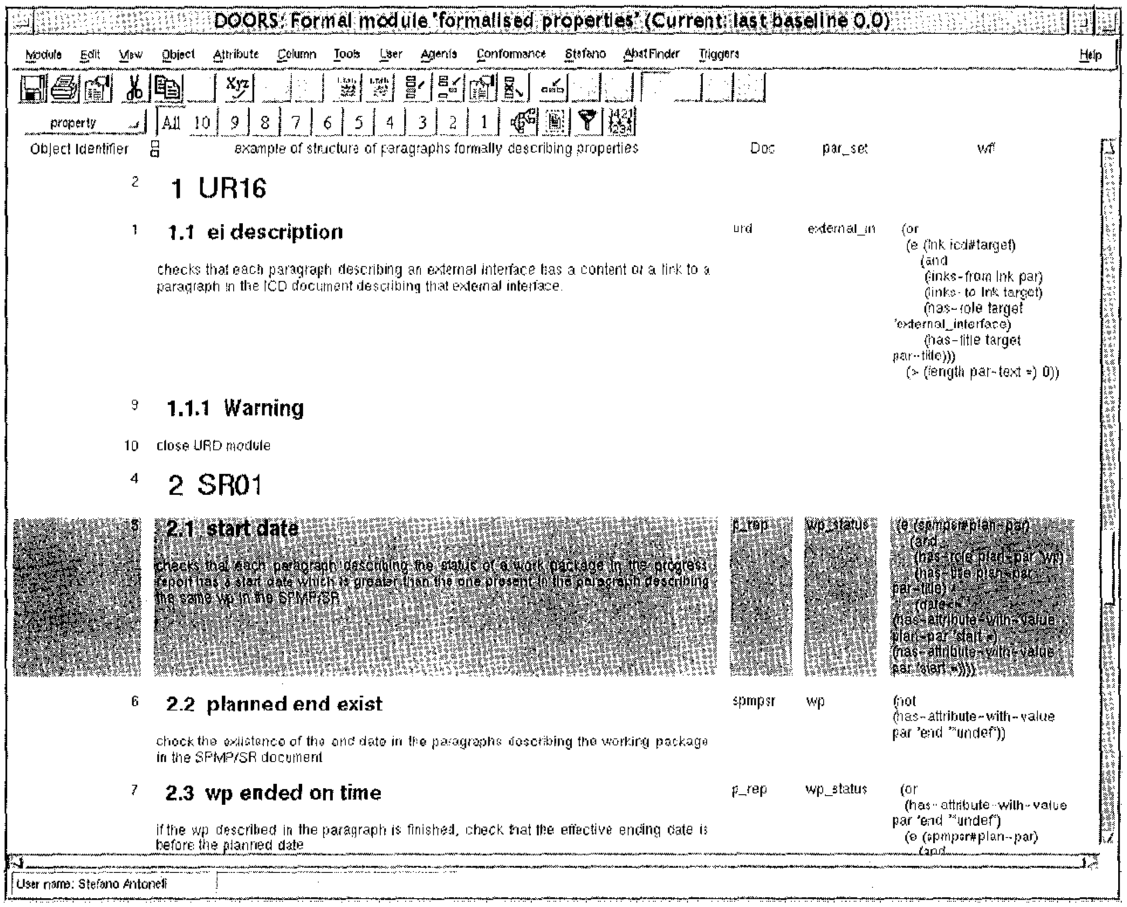
PR42 **1.3.10 SR10**

User name: Steve

Fig. 8. Fig. 8. PSS-05 in the compliance manager—standard view.

5.2.2 AP5

AP5 is an extension of Common Lisp that "allows users to program at a more 'specificational' level." "AP5 represents state (that is data) as a set of relationships among a set of objects, as in a model of first order logic. The language for accessing this data includes the language of first order logic" [3]. A relation between objects is represented by a tuple, containing the name of the relation and the list of objects related to each other: (relation-name obj1 obj2 ...). A tuple can be used in a well formed formula (WFF) as a predicate. Relation sets can be updated inserting tuples manually, or can be derived from the information already present in the database, using a WFF. The new relation will be updated as soon as the relations involved in its definition change. AP5 WFF are built from primitive relations, logical connectives (NOT, AND, OR, IMPLIES, EQUIV, XOR),


DOORS: Formal module 'formalised properties' (Current: last baseline 0.0)
Module Edit View Object Attribute Column Tools User Agents Conformance Stefano AbstFinder Triggers Help
property
All 10 9 8 7 6 5 4 3 2 1

Object Identifier	example of structure of paragraphs formally describing properties	Doc	par_set	wff
2	1 UR16			
1	1.1 ei description checks that each paragraph describing an external interface has a content or a link to a paragraph in the ICD document describing that external interface.	urd	external_in	(or (e (lnk icd#target) (and (links-from lnk par) (links-to lnk target) (has-role target 'external_interface) (has-title target par-title))) (> (length par-text =) 0))
9	1.1.1 Warning			
10	close URD module			
4	2 SR01			
5	2.1 start date checks that each paragraph describing the status of a work package in the progress report has a start date which is greater than the one present in the paragraph describing the same wp in the SPMP/SR	p_rep	wp_status	(e (spmpsr#plan-par) (and (has-role plan-par 'wp) (has-title plan-par par-title) (date< (has-attribute-with-value plan-par 'start *) (has-attribute-with-value par 'start *))))
6	2.2 planned end exist check the exilistence of the end date in the paragraphs describing the working package in the SPMP/SR document	spmpsr	wp	(not (has-attribute-with-value par 'end "undef"))
7	2.3 wp ended on time if the wp described in the paragraph is finished, check that the effective ending date is before the planned date	p_rep	wp_status	(or (has-attribute-with-value par 'end "undef") (e (spmpsr#plan-par) (and

User name: Stefano Antonelli

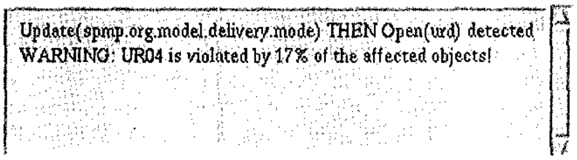
Fig. 9. Property descriptions in the compliance manager.

existential and universal quantifiers, and variables. AP5 also provides triggers. Every time a tuple is added to the database, it checks whether the conditions associated to all the defined triggers are satisfied. If they are, a lisp function, associated to each trigger, is executed.

The combination of AP5 (in the form of a library of LISP functions) and Common Lisp (in the form of the interpreter and compiler) provides an excellent vehicle for defining and experimenting with notations.

5.2.3 FLEA

The Formal Language for Expressing Assumptions (FLEA) is a monitoring system, which gathers the events occurring in an application and gives notification of certain combinations of these events. FLEA provides a small temporal logic-like language (discussed above) particularly suited to the expression of event combinations. It is a Common Lisp application, which uses the AP5 database. When a relevant event occurs in the monitored system the system itself notifies it by adding a tuple to the database (external event). A time tag will be automatically associated to each tuple. The description of an event combination is compiled into a query, which is executed every time the database is updated. If the query is successful, this is an event as well (definition event), and is added to the database. A definition event can be part of an event combination. The monitored system can also add information other than events to the database (relation), if it is useful to the specification of event combinations. The FLEA notation is an extension of the AP5 notation; once in the database, events are in fact primitive relations. There are a variety of approaches to implementing event monitoring and particularly temporal composition. For our prototype we have found the way in which FLEA makes the time/space tradeoffs entirely satisfactory.



```
Update(spmp.org.model.delivery.mode) THEN Open(urd) detected
WARNING: UR04 is violated by 17% of the affected objects!
```

Fig. 10. Diagnostic in the compliance manager.

5.2.4 Integration

Currently we have a very loose integration of DOORS with AP5 in which we create a mirror representation of the structure of a DOORS document in AP5. Properties and policies are written within the compliance manager. They

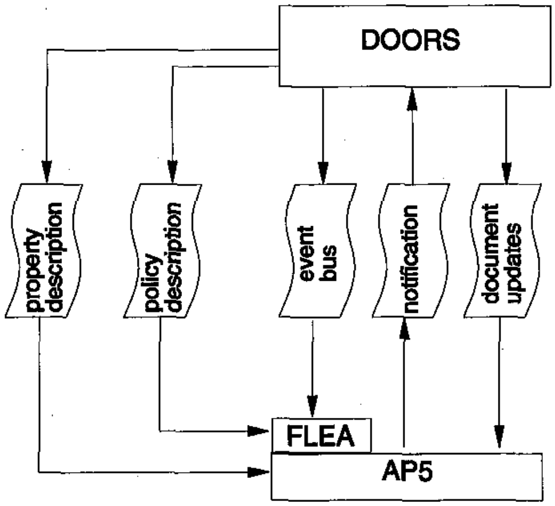


Fig. 11. Physical architecture of the prototype.

are exported to property description and policy description files, which are compiled by AP5 and FLEA, respectively.

DOORS generates events for significant activities such as opening a document or updating an attribute. We have made some minor modifications to the DOORS kernel to increase the range of actions that generate events. When an event occurs, DOORS writes a notification to the event bus file. FLEA reads this file periodically and updates its database, if necessary.

AP5 provides output in the form of messages to the user and writes to a notification file. DOORS monitors the notification file and the information is made available to the compliance manager so that the practice states are set appropriately. The occurrence of all checks and their results are written to the notification file. For policies in guideline mode the policy itself is written on the notification file. DOORS reads the file and sets the state of practice to unsafe until it is notified that the check specified in the policy has been performed. For policies in warning mode the check is triggered directly by the policy and the policy itself and the result (compliance or diagnostic) are written on the notification file. DOORS reads the file and updates the practices appropriately. For policies in error mode, that invoke a check which fails, the policy itself and the failure are written to the notification file. DOORS reads the file and generates a veto, a DOORS kernel facility that we use to implement aborts.

The work that was required to achieve this level of integration is not very substantial and leads us to believe that our overall architecture is sound. Much of the work was as a result of the prototype status of FLEA communication mechanisms. Modifications to DOORS were not strictly necessary but gave us a little more flexibility in writing policies.

5.3 Status

The current implementation is a prototype which has been assembled as a vehicle for experimentation and as a proof of concept. Though we intend to field a version of this prototype on a trial basis there are a number of significant changes that will be required before it can be more widely used.

Our current implementation has poor performance and the integration of DOORS and AP5 is clearly only suitable for our preliminary evaluation—it presents problems of synchronization and scale. We intend to develop a direct translator between our document and property notations and DXL. We intend to continue to use FLEA which provides a very flexible event monitoring service. However, notification of events from DOORS to FLEA is unsatisfactory and we plan to use a more effective communication mechanism than a shared file, probably sockets. The use of the notification file to update the compliance manager is the subject of current work.

In practice, we have found some need for the developer to be able to force the execution of particular checks outside the framework of the policies. We have prototyped a mechanism to do this but it is poorly integrated with our compliance manager interface.

As we gain more experience we are developing a better understanding of how to write policies and use of the different modes.

DOORS provides some support for multiple users; we have not considered in detail how this impacts our support environment. This is a relatively serious drawback and though we are moderately confident that our scheme is applicable in a multiuser setting, this issue requires attention. Our aim clearly is to validate the overall approach prior to dealing with multiple user support.

6 Related Work

Our work draws on a number of intertwined strands of research. The problem of compliance, as we have treated it, is closely related to inconsistency management in specification. Key contributions in this area are [9], [6], [11].

Our work concentrates on inconsistency detection and identification and leaves handling of inconsistency to the users of the tool. For some indication on how handling might be tackled see [13].

The use of process modeling techniques to control the application of consistency checks has been explored in [9] and in [22]. The approach described in the latter paper is similar to the one presented here and differs from that generally taken in the process modeling literature. There is no explicit representation of a global process, but rather a set of distributed local models, that may be inconsistent. Consistency checks are triggered on recognizing events by means of pattern-matching using regular expressions.

A similar approach to process support is taken in the non intrusive process centered software engineering environment Provence [20] which deploys the event-action specification tool Yeast [21]. An interesting feature of this work is the use of event contexts [1] to constrain event matching. We can reproduce this in FLEA.

The problem of process deviation has been analyzed in [4], who introduce the LATIN process modeling language and the SENTINEL support environment. A process model is defined in LATIN and enacted within SENTINEL. Deviations may occur since, for example, the user may force the execution of an action such as checking-out a module even if the current state of the process does not fullfill the conditions that the model requires for the execution of the action. LATIN defines the requirements for compliant performances, by the means of process constraints. The idea is that if a deviation does not result in a breach of the constraints, enactment may proceed. If on the other hand a deviation will breach the constraint a process of pollution analysis and repair is invoked based on reasoning over the performance traces.

Process constraints in LATIN are similar to properties in our approach. The requirement that deviations do not breach the constraints means that LATIN leads to fully compliant projects, while our approach is looser. We do not have a full process model to enact: we only have the product model, and a set of properties the product must satisfy if the performance is to be compliant. Actions are modeled very crudely; we are only really interested in them if they affect properties.

Cugola et al. [5] take a broader approach to the problem of process deviation, and present a formal framework for characterizing interactions between a "human-centered system" and automated support for that system. The approach encompasses process centered software engineering environments and work flow management systems. To formally capture the notions of inconsistency and deviation, the framework uses state machines (not necessarily finite) to model both the human-centered system and the associated support system. The machine modeling the human-centered system explicitly distinguishes between inconsistent and consistent states, and between expected and unexpected transitions, that is deviations. Linking the two machines by a pair of relations, between states and transitions, respectively, the framework formalizes the concepts of inconsistencies and deviations between the two systems and gives us a way of talking about the ability of the support system to provide effective support for the human-centered system.

The application of the framework to some process centered software engineering environments leads the authors to conclude that in order to minimize the problems associated with inconsistency and deviation it is necessary to enrich the semantics of the process modeling language, to facilitate the representation of a larger number of states and transitions; and, to enrich the architecture of the process support system with mechanisms that will distinguish all the events occurring in the human-centered systems, and map them onto the process model under enactment.

It is rather difficult to characterize our work in terms of this framework. Our minimalist and tolerant approach means we have no need to model deviations explicitly. Our treatment of events assumes the use of a support environment (DOORS) that provides an adequate set of events. We limit ourselves to those domains where a significant body of empirical knowledge about the human-centered system is available, in the rigorous form of standards. These standards also identify a set of significant events which we can use.

Methods for process validation, defined as the assessment of the discrepancies between the process actually followed and the normative processes defined in process models are discussed in [7]. The methods are based on string difference metrics. Characters in these strings represent process events. Strings which are captured from the performance of the actual process are compared with strings generated from the process model and a distance measure is derived using standard algorithms. Our approach differs in that we do not have an explicit process model but we use our product focus to provide more specific guidance about how to move from noncompliance to compliance. If we introduced an explicit process model we would be able to use this approach as we maintain an event trace.

7 Summary and Further Work

In this paper, we have introduced standards compliance as an issue of importance in software engineering and have developed a model which identifies the main elements of standards and of the support required to manage compliance. We have presented an environment which implements the model and described the structure of this environment.

The principal contributions of our work are:

- the identification of the issue of standards compliance,
- the development of a model of standards and support for compliance management,
- the development of a formal model of product state with associated notation; a powerful policy scheme that triggers checks,
- a flexible and scalable compliance management view.

Our environment is based on an industrial strength document management system. Our claim to scalability is justified both in our use of the services of this system and by our experiments with a real industrial standard. Our approach is lightweight, in the sense that it requires relatively simple augmentation of tools that are required in any case. The notations we have provided are simple to use and based on well-established and widely understood concepts. We have realized a "tolerant" approach which, we believe, fits well with the way in which complex software systems are built.

We hope that the details of our prototype do not distract from these contributions. We believe that much of what we have accomplished could be simply and cheaply engineered into similar document management systems.

We are building a second prototype that overcomes some of the difficulties that we experienced with the prototype described in this paper but uses the logical architecture set out above. This prototype will be geared towards industrial use. As such it will remove the dependency on FLEA, AP5, and CLISP and it will have a, probably simpler, event monitor built using DXL. This will make it simpler to install the prototype and allows it to execute on all hardware platforms DOORS operates on. We also have strong reasons to believe that this second prototype will execute more efficiently as the need for file-based communication between the event monitor and the document manager disappears.

We have scheduled a program of industrial evaluation for this second prototype in the immediate future. Several existing DOORS users have agreed to use the compliance manager to formalize their development standards and to evaluate the support for compliance that we provide.

Our immediate research agenda is set by the discussion in Section 5.2. However, some broader issues remain to be tackled. In addition to practices discussed above, many standards incorporate statements about the high-level goals of the development process. The question of how we can establish that the practices correctly implement these high-level goals is one which needs an answer. Some preliminary work on such correctness problems has been developed in [24].

We would hope that the ideas on which our work is based, can be fed back into the standards process itself and might assist in the formulation of new systems engineering standards, for example we are working on an emerging standard [18].

Customers who procure the development of a new system often demand compliance to a development standard. Now, our compliance manager displays the degree of compliance at one point in time. Customers may also be interested in the evolution of compliance throughout the development process. To achieve that we would need to measure how compliance develops over time. Such compliance measurements could also be used and integrated into an experience factory approach [2]. The integration would then support process improvement based on compliance monitoring of previous projects.

We are party to the shared research aim of building a better formal understanding of inconsistency, a contribution to this is [11]. In particular, we hope that our work will yield a better understanding of how to pull together many of the different research strands and also perhaps provide a test-bed for new tools and techniques.

Acknowledgments

The authors are grateful to Martin Feather for kindly providing us with FLEA and to Don Cohen for his technical advice on using AP5. The authors are also grateful for the comments of the anonymous referees who have allowed us to significantly improve this paper. This work was funded by the Teaching Company Directorate through Scheme No. 1884 and performed while Carlo Montangero was an EPSRC Visiting Fellow funded through Grant No. GR/L54561. Stefano Antonelli was a visiting research student at University College London. The authors acknowledge support from the ESPRIT Working Group 21185 Promoter.2. A preliminary position paper discussing some of the ideas detailed in this paper was presented at the ICSE-19 Workshop entitled *Living with Inconsistency* held at Boston, Massachusetts in May 1997.

References

[1] N.S. Barghouti and B. Krishnamurthy, "Using Event Contexts and Matching Constraints to Monitor Software Processes," *Proc. 17th Int. Conf. Software Eng.*, pp. 83–92, Seattle, Washington, IEEE CS Press, 1995.

[2] V.R. Basili, G. Caldiera, F. McGarry, R. Pajerski, G. Page, and S. Waligora, "The Software Engineering Laboratory—An Operational Software Experience Factory," *Proc. 14th Int'l Conf. Software Eng.*, pp. 370–381, Melbourne, Australia, IEEE CS Press, 1994.

[3] D. Cohen, *AP5 Manual*, Oct. 1992. ftp://ftp. isi. edu/pub/ap5/

[4] G. Cugola, E. Di Nitto, C. Ghezzi, and M. Mantione, "How To Deal with Deviations During Process Model Enactment," *Proc. 17th Int'l Conf. Software Eng.*, pp. 265–273, Seattle, Washington, ACM Press, 1995.

[5] G.P. Cugola, E. Di Nitto A. Fuggetta, and C. Ghezzi, "A Framework for Formalizing Inconsistencies in Human-Centred Systems," *ACM Trans. Software Eng. and Methodology*, vol. 5, no. 3, 1996.

[6] S. Easterbrook, A. Finkelstein, J. Kramer, and B. Nuseibeh, "Coordinating Distributed ViewPoints: The Anatomy of a Consistency Check," *Int'l J. Concurrent Eng.: Research & Applications*, vol. 2, no. 3, pp. 209–222, 1994.

[7] J.E. Cook and A.L. Wolf, "Software Process Validation: Quantitatively Measuring the Correspondence of a Process to a Model," *ACM Trans. Software Eng. & Methodology*, vol. 8, no. 2, pp. 147-176, 1999.

[8] M. Feather, "FLEA: Formal Language for Expressing Assumptions—Language Description," private communication, Apr. 1997.

[9] A. Finkelstein, D. Gabbay, H. Hunter, J. Kramer, and B. Nuseibeh, "Inconsistency Handling in Multi-Perspective Specifications," *IEEE Trans. Software Eng.*, vol. 20, no. 8, pp. 569–578, 1994.

[10] A. Finkelstein, J. Kramer, and M. Hales, "Process Modelling: A Critical Analysis," *Integrated Software Reuse: Management and Techniques*, P. Walton and N. Maiden, eds., pp. 137–148. Chapman & Hall and UNICOM, 1992.

[11] A. Finkelstein, G. Spanoudakis, and D. Till, "Managing Interference," L. Vidal, A. Finkelstein, G. Spanoudakis, and A. L. Wolf, eds., *Proc. Joint SIGSOFT'96 Workshops*, pp. 172–174, ACM Press, 1996.

[12] D. Harel, "On Visual Formalisms," *Comm. ACM*, vol. 31, no. 5, pp. 514–530, 1988.

[13] A. Hunter and B. Nuseibeh, "Analysing Inconsistent Specifications," *Proc. Third IEEE Symp. Requirements Eng.*, pp. 78–86, Annapolis, Md., IEEE CS Press, 1997.

[14] IEEE, *IEEE Standard for Developing Software Life Cycle Processes*, pp. 1,074-1,995, IEEE CS Press, 1995.

[15] Introduction to ISO, 1997. http://www. iso. ch/infoe/intro. html

[16] ISO/IEC "Quality Management and Quality Assurance Standards—Part 3: Guidelines for the Application of ISO 9001 to the Development, Supply and Maintenance of Software, ISO 9000-3," *Int'l Standardisation Organisation*, 1994.

[17] ISO/IEC, "Int'l Standard, Information Technology Software Life Cycle Process, ISO 12207," 1995.

[18] ISO/IEC, "Draft Systems Engineering Standard, ISO 15288," 1997.

[19] ISO/IEC, "Software Process Improvement and Capability Determination," *Int'l Standardisation Organisation*, 1997.

[20] B. Krishnamurthy and N.S. Barghouti, "Provence: A Process Visualization and Enactment Environment," I. Sommerville and M. Paul, eds., *Proc, Software Engineering—ESEC'93*, pp. 451–465, Garmisch-Partenkirchen, Germany, Lecture Notes in Computer Science 717, Springer-Verlag, 1993.

[21] B. Krishnamurthy and D.S. Rosenblum, "Yeast: A General Purpose Event-Action System," *IEEE Trans. Software Eng.*, vol. 21, no. 10, pp. 845–857, 1995.

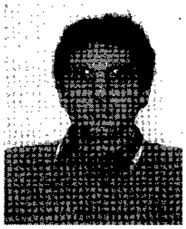
[22] U. Leonhardt, A. Finkelstein, J. Kramer, and B. Nuseibeh, "Decentralised Process Enactment in a Multi-Perspective Development Environment," *Proc. 17th Int'l Conf. Software Eng.*, pp. 255–264, IEEE CS Press, 1995.

[23] C. Mazza, J. Fairclough, B. Melton, D. De Pablo, A. Scheffer, and R. Stevens, *Software Engineering Standards*. Prentice Hall, 1994.

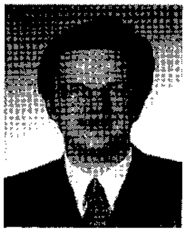
[24] C. Montangero and L. Semini, "Applying Refinement Calculi to Software Process Modelling," *Proc. Fourth Int'l Conf. Software Process*, pp. 63–74, Brighton, U.K., IEEE CS Press, 1996.

[25] Quality Systems & Software Ltd., Oxford Science Park, Oxford, U.K., "DOORS Reference Manual (V3. 0)," 1996.

[26] Rational Software Corp., Santa Clara, Calif., *UML Semantics*, version 1. 1 alpha r6 ed., July 1997.



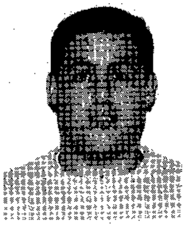
Stefano Antonelli. This paper is dedicated to the memory of Stefano Antonelli. Stefano completed his studies to obtain the "Laurea" degree in Informatics Engineering at Politecnico di Milano in 1997. He was writing his dissertation on the subject of this paper while working at University College London as a visiting research student. Shortly after the completion of the first draft of this paper, he died suddenly and in tragic circumstances. He is much missed by his friends and colleagues.



Wolfgang Emmerich received a diploma degree in Informatics from the University of Dortmund, Germany (1990) and a doctor of science degree in mathematics and informatics from University of Paderborn (1995). He then joined City University, London, United Kingdom as a lecturer. In November 1997, he took his current position as a lecturer at University College London, United Kingdom. His research interests are in software processes, requirements, and software architectures for distributed systems. He has particular interests in managing distributed documents using XML and the paradigm of distributed objects. Dr. Emmerich is a partner and cofounder of Zühlke Engineering GmbH, an IT consultancy that specializes in object technology. He has published extensively in software engineering and is currently writing a textbook on *Distributed Objects* (John Wiley & Sons). Dr. Emmerich is a chartered engineer, a member of the IEEE Computer Society, and the ACM. He is secretary of the Requirements Engineering Specialist Group of the BCS and a Member of the IEE.

Anthony Finkelstein holds a BEng degree in systems engineering, an MSc degree in systems analysis, and a PhD degree in design theory. He is professor of software systems engineering at University College London, a post held in the Department of Computer Science. He is also a director of the UCL Centre for Systems Engineering. Formerly, he was professor of computer science at The City University, London and head of the Department of Computer Science. Prior to that, he was a member of the academic staff at Imperial College of Science, Technology & Medicine. His research interests are in the area of software systems engineering and, in particular, in requirements engineering. He has contributed to software specification methods, software development processes, tool, and environment support for software development. Recent work has included contributions to work on specification from multiple viewpoints and to requirements traceability. He has published more than 150 papers in these areas and held research grants totaling in excess of 6m. He is a chartered engineer, a member of the IEE and BCS. He is a founding member of IFIP WG 2.9 Software Requirements Engineering.

Carlo Montangero received a degree in electronic engineering from the Politecnico di Milano in 1969 and is a professor of programming at the Department of Informatics of the University of Pisa since 1981. His research interests have always been in software development methods and processes, and the related support environments. Currently, his major interest is in refinement calculi to specify and develop coordination architectures for distributed systems. He is a member of the ACM, IEEE Computer Society, AICA, and EATCS.



Stephen Armitage received a BSc degree in computer science from the University of Hertfordshire, United Kingdom, in 1996. He was a teaching company associate, working on a collaborative research project between University College London and Quality Systems & Software Limited. The project involved checking project compliance to standards. His research interests include systems engineering, requirements engineering, and process modeling. He currently works for QSS Ltd. as an Internet and application developer. He is a member of the British Computer Society, ACM, IEEE, and IEEE Computer Society.

Richard Stevens received a BSc (honors) in physics from the University of Wales in 1967, an MSc degree in solid state physics from the University of Bath in 1969, and a PhD degree in solid state physics from the University of Wales in 1972. Dr. Stevens is the chief technical officer and founder of QSS Ltd. He was formerly head of methodology, technology, and quality in the IS Division of the European Space Agency, where he was instrumental in the development of the PSS-05 standard. He is now leading the development of the DOORS requirements management tool and accompanying training courses, plus the theory and practice of Innovation. Dr Stevens is the author of several textbooks on systems engineering and standards and has written numerous academic papers and articles for publications such as *Byte* magazine and *The Times and Guardian* newspapers. In 1998, he was awarded the INCOSE fellowship in recognition of his work in systems engineering.