Formal and Informal Aspects of Requirements Tracing

Francisco A. C. Pinheiro

Universidade de Brasília
Departamento de Ciência da Computação
facp@cic.unb.br

Abstract. We discuss some formal and informal issues regarding requirements tracing. Our view is that requirements tracing needs are guided by necessity and not by any pre-defined structure. On the other hand the use of formal techniques and formalization of requirements tracing itself is essential for an efficient automation and has many benefits such as allowing verification. We present an abstract formal model for tracing with some capability to meet the informal needs of requirements tracing such as the use of informal or unstructured information and the need to trace information not thought of in advance.

Keywords: Requirements traceability, requirements models, requirements processes

1 Introduction

Regardless of its essential technical aspects, the development of software systems is a social activity involving, from the start, the user's needs for the system and a whole web of social interactions as well as individual and group perception of what is being done. Therefore, software development mixes both the formal and informal aspects of information. These aspects are called the 'dry' and the 'wet' in (Goguen 1992). They are also discussed in (Goguen 1996) with respect to requirements engineering. Taking the view that the informal aspect is more dominant in the initial phases of software development we would expect it to have a great influence on many of the requirements engineering activities. This paper discusses some formal and informal issues regarding requirements tracing, and presents an abstract formal model for tracing with some capability to meet the informal needs of requirements tracing such as the use of informal or unstructured information and the need to trace information not thought of in advance.

First we present a view of requirements traces as being naturally produced during software development, and of requirements traceability as being the ability to capture those traces. Then, we discuss the concepts of trace definition, production, and extraction in the context of software tracing models. This sets up the context for our discussion on the formal and informal aspects of requirements tracing. Finally, we present an abstract tracing model and argue for its adequacy to deal with some informal needs of requirements tracing.

2 Requirements Tracing

A prominent aspect of requirements traceability we wish to highlight is the natural occurrences of traces. We trace a requirement back to a document because the requirement carries traces of the document. It may be that the requirement is directly extracted from the document, it may be that the document contains statements supporting the requirement, or it may be something else. In any case, the existence of some *influence* of one on another is necessary if they are to be viewed as *related* objects. In the same way, we trace a requirement forward to a design module because the module carries traces of the requirement. Again, it may be that the module is designed to meet the requirement, it may be that the module specifies a test procedure for the requirement, or it may be something else. In this view we have to identify which traces we want to control, how they may be captured, and how they may be followed afterwards. Our working definition will be:

> **Requirements Traceability** *refers to the ability to define, capture, and follow the traces left by requirements on other elements of the software development environment and the traces left by those elements on requirements.*

This definition depends on previous notions. For us, a software development environment involves not only the technical, but also the social aspects of software development. Its elements comprise not only the technical artifacts such as specifications, diagrams, and code, but also people, policies, decisions, and even less tangible things like goals and concepts.

The idea is that *traces are naturally produced* as a result of activities, actions, decisions, and events happening during software development. If a requirement is a result of some discussion, traces of the discussion are present on the requirement at the very moment of requirements elaboration. This is just as footprints are present on a surface at the very moment someone walks on it. The definition of what traces should be considered, what elements should be involved, how trace production may be captured, and how traces should be retrieved is central to any tracing model.

In ordinary usage a trace is defined as *a mark, track, sign, etc. showing what has existed or happened* (Murray et al. 1989). The 'mark' does not need to be a material thing. It can be, for example, *a non-material indication or evidence of the presence or existence of something, or of a former event or condition.* In the natural world to trace something amounts to look for, or to follow the course, development, or history of traces, or occurrences of traces, left by whatever is being traced.

In computer science terms the picture is slightly different because this definition is blurred by the need for representation. Things should be defined and represented in machine processable ways in order to be accessible by computer programs. Moreover, whatever is defined in that way should also be produced in machine processable terms, i.e., it should be registered or stored in a medium that can be accessed by computer programs. Also, the ways of accessing those representations should be machine effective. Therefore any tracing model is circumscribed with respect to trace definition, trace production, and trace extraction.

2.1 Trace Definition

When one software object leaves traces on another software object, those objects may be viewed as related in some way. The nature of the trace is represented by the particular relation indicating how the two objects interact. For example, consider that a requirement is derived from another one indicating that although they may be different requirements one induced the existence of the other. If we trace this requirement back to the one from which it was derived then we may view the derived requirement as carrying traces of the other.

In order to be able to register traces of software objects a tracing model should define what its *trace units* are and what its *traces* are, i.e., how they are represented in the model. A tracing model should also define what *sorts* of traces may exist, indicating what the trace is intended to represent, what are the units that may be involved, and under what conditions traces may be registered.

A tracing model may define just one sort of trace to cover all possibilities, but this would leave the interpretation of the trace to take place outside the model. For example, a tracing model may have only one kind of link to indicate the there is a trace of an object on another one. However, as all links are alike it is not possible to determine from the information in the model what are the different types of traces. Only in very specific cases is it possible to fully define a trace so that its interpretation would be formally precise. Nevertheless, even if the interpretation is left to be done outside the model, it is still possible to incorporate in the model indications of how the trace should be interpreted. For example, one may use named links to give additional information of what the trace is meant to represent.

2.2 Trace Production

In the natural world a trace is produced as a result of actions, events, or conditions which naturally impress or leave a mark of their effect. Software traces are also naturally produced as a result of the working practices of software development. However, the use of automated tracing models may not capture traces as they are produced.

The trace production itself can sometimes be automated. In this case the software development activities are carried out using tools that are integrated with the tracing model in a way that allows the automatic registration of traces.

Trace production is an important aspect of tracing models not only because one can trace only what is available, but also because it may interfere directly with the activities of developing software. It may impose an overload on people carrying out these activities. The less intrusive the trace production, the more efficient and accurate the use of the tracing model is.

2.3 Trace Extraction

In order to follow a trace it is necessary to extract the registered representation of the trace. The trace may be extracted in several ways and the trace extraction features of a given tracing model depend on how the trace is defined and on how the trace is produced.

Trace extraction depends on trace definition and production to the extent that one can extract only what has been registered. Trace extraction also needs to be defined in the model in terms of an effective procedure to perform the extraction. A tracing model should provide different and flexible ways to extract information registered in it, so that the most appropriate one can be chosen for each occasion.

3 Formal and Informal Needs for Tracing

One aspect of registering the relevant information is to define the adequate referential: relevant with respect to what? The referential in question is usually set by the methods and techniques employed by the people developing systems. Those methods and techniques determine to a great extent what kind of information is needed in the next steps and how they relate within the framework prescribed by them. However, not all needs for tracing may be encompassed by using methods and techniques. Certainly not when what is sought refers to the very use of them. For example, the answer to what data-flow is input to process X in a certain data-flow diagram involves only elements from the method itself, while asking why a particular process in the same diagram is described the way it is can only be answered with recourse to a meta-model, where the *use* of the model can be assessed. In this case the referential involves a wider context that may include the social environment in which the development is carried out.

The use of formal tracing models helps in automating the traceability process. Not only is the automation of the model facilitated, but also the automatic generation of traces and the definition of procedures to verify consistency and correctness of traces. In short, using a formal tracing model, it is easier to enforce the appropriate registration and extraction of traces and trace units according to a given interpretation of appropriateness – the interpretation given by the model. But formalization may apply not only to formal models of traceability. The use of formal definitions for the contents of a trace unit or formal descriptions for software processes may also bring benefits for requirements tracing. For this allows the definition, and automation, of traces in a more fine-grained way. For example, using formal specification languages to describe the system being developed makes possible the definition of traces involving not only the specification as a whole, but also its elements. This may increase traceability efficiency since it provides a more focused way of discerning which traces to follow if the need for tracing is identified with a particular element of the specification.

However, not all activities in software development are amenable to formalization and for those which are, we constantly face an applicability problem.

The automation of some activities may be perceived as non-natural, so that their enforcement would hamper traceability rather than help it. At least, enforcing non-natural practices causes problems that easily overcome the possible traceability benefits. Another point is that it may not be possible to correctly register all relations prescribed by a given tracing model. This point has two aspects. First, even if the relations are regarded as meaningful they may not be perceived as such by all people in charge of registering them and, therefore, they will probably be left out. Second, one cannot devise all *necessary* relations for future traceability needs. This point is explored in more detail in the next section.

The information that should be made traceable is in many cases inherently informal, e.g., texts and natural language statements, diagrams, graphs, etc. The reduction of this type of information to some formal structure may facilitate their manipulation but it is likely not to fulfill traceability needs since what relates a text, graph, or diagram to a requirement is not their formal features, if any. They are related instead based on many different grounds, e.g., relevance, translation, causality, policies and even organizational politics. Also, the reason to relate this type of information to other artifacts in software development is in itself an important information to be made traceable. The rationale behind each decision is in many cases what is sought when tracing software objects. But it is not sufficient to capture the argumentation in a frozen way; sometimes one wants to reassess the line of reasoning followed in achieving a given decision. This may be subject to formalization in terms of a model capturing the discussion process in some reasonable structure, but more important than that is the possibility to capture part of the context in which the discussion occurred. This points to the use of multiple representation and hyper-media artifacts: playing back a video or tape recorded interview allows not only the reassessment of what happened but also a re-interpretation of the events.

Informality is also needed to deal with the fundamentally unstructured way in which information is gathered and used. Information itself is in many cases unstructured or, more particularly, has a complex structure which is dependent on the (social) context. Moreover, the production and posterior use of this kind of information is also complex and context sensitive. Therefore, no predefined structure will contain the necessary elements to satisfy the real needs for traceability. *The search for related information is guided by necessity not by pre-defined structures.*

4 Traceability Support

The diversity and huge amount of information dealt with when developing large software systems points to the need for automated tools to support development practices, including traceability. To some extent requirements traceability is already supported by existing tools and environments to automate the development process. Thus, configuration management systems support the definition of project baselines and control of software versions, process centered environments allow the definition and control of software processes and the enactment of associated tools, CASE tools provide support for several phases of the development, etc. All these tools and traceability tools in particular offer ways for registering and retrieving related information.

Current traceability support provides useful ways to manage large amounts of information. When developing large systems it is hard to remember all links that were made to relate information and it may even be impossible, in a distributed or multi-team development, to know that they exist. Moreover, the matter is complicated when immediate links to a given piece of information are not what is required. One usually wants to follow a chain of links that goes far away from the starting point of tracing. Current traceability support comes in handy for these cases, allowing the retrieval of important related information which would otherwise be difficult to find. Moreover, the fact that increasingly large and complex software systems are constantly being built with varying degrees of success implies that traceability is somewhat achieved in the process. After all, for a system to be satisfactorily completed implies that people have performed the natural going back and forth of the development process.

However, we suggest that the existing traceability support is not as adequate as it should be (or at least, as it can be). We argue that current traceability tools do not provide full support, and that traceability is achieved through personal contact and searches that go beyond automation, e.g., group discussions and inquiries. There is some evidence for this view, which explains the existence of many proposals to enhance traceability through models aimed at capturing requirements related discussion (Ramesh and Dhar 1994, 1992, Ramesh and Luqi 1994), design rationale (Potts and Burns 1988), personnel based traceability (Gotel 1995,Gotel and Finkelstein 1995), etc.

Most of the time the information should be traced beforehand in order for it to be (automatically) traceable in future. This circularity is enforced by current traceability tools, which are based on the existence of explicit relations between objects (through links, attributes, etc.). To register related information in these tools amounts to establishing a trace; one links documents to requirements only if there is a perceived trace between them. In the same way requirements are linked to design modules, and those to code and so on. As one traces only what has been related, the net result is that problems that require forward or backward traceability to or from any given object are supported only to the extent that (traceability) links already exist. The same can be said of almost all traceability techniques: requirements are tagged with information known to be relevant; cross references are made with objects known to be related; traceability matrices are filled with known connections, etc. Therefore, the current support for traceability is restricted, to a great extent, to the extraction of information already made traceable.

However, tracing existing links can hardly be considered as satisfying all needs for tracing. Two problems can easily be identified. On one hand, in an environment characterized by dynamic change and in which the unfolding process of developing information is open to negotiation and to several solutions, it is not possible to foresee all links that are going to be needed in future. On the other hand, even for those links that may be thought of in advance, it may not be adequate to register them all. For this second point there is an interesting illustration. In a study for the NASA Goddard Space Flight Center Mission Operations and Data Systems Directorate (MO&DSD)[1] a commercially successful requirements tool is said not be used because people "felt they spent more time than they wanted on traceability". What is interesting is the fact that a requirements traceability tool is left out precisely because it requires too much traceability. In addition, an existing link may not give the necessary information to trigger the desire of a person to follow it in all relevant cases.

As for the impossibility of complete automation, we agree that much of the information used in system development, mainly in its initial phases, is of an unstructured nature or, at least, little is gained from imposing one particular structure. We believe that this is true not only of information viewed as a unit, but also of information viewed as relations. Although it is our view that tracing cannot be fully automated (e.g., in the extreme case it is even possible that the information wanted is not stored in a computer at all; which does not mean that such information was not used at some point in the development), we believe that the basic framework provided by existing traceability tools may be enhanced to cope with some unstructured information. Although the arguments may not be the same, our conclusions agree with the conclusions of a recent workshop on requirements engineering (Pohl and Peters 1996) which identified a need for improvements in tool support for traceability to automatically record the traces and provide suitable interfaces for using trace information, and in tool support for structuring and documenting requirements as well as support for several kinds of representation such as video, text, graphics, diagrams, etc. Before presenting our requirements tracing model we review, in the next subsections, some of the proposed support for requirements tracing.

4.1 Tracing Models

Tracing models provide a representation for traces and trace units. They establish the structures containing the elements and the relations used in tracing, usually specifying their types as well as the constraints under which the elements of the model can be related. They also provide the means to interpret the structure obtained.

- The IBIS (Conklin and Begeman 1998, Rein and Ellis 1991) related models intend to capture design rationale by providing automated support for discussion and negotiation of design issues. Basically, they implement the *Issue Based Information Systems* model which consists of four concepts and nine

[1] The report is part of the MO&DSD tools capability inventory and is available on the net http://joy.gsfc.nasa.gov/MSEE/doors.htm. It only reflects a personal judgement but illustrates our point that traceability information is not always dealt with in advance.

kinds of links to relate them. The basic concept of the IBIS model is that of *Issue*. An Issue may be any problem, concern, or question that may require discussion (and resolution). Each Issue may have many *Positions* ascertaining a possible solution for it. Each Position may in turn have *Arguments* supporting or rejecting it.

- The REMAP model (Ramesh and Dhar 1992, 1994, Ramesh and Luqi 1993, 1994) enlarges the IBIS model. It adds to the original set of IBIS concepts (Issue, Position, and Argument), the new concepts of *Assumption*, *Decision*, *Requirement*, *Constraint*, and *DesignObject*. The links among IBIS concepts remain the same and new ones are created involving the new concepts. Some examples of these new links are: a Requirement *Generates* an Issue, an Assumption *Qualifies* an Argument, an Argument *Depends-on* an Assumption, and a Decision *Resolves* an Issue.
- Contribution Structures is a model that addresses personnel-based traceability (Gotel 1995, Gotel and Finkelstein 1995, 1996), making traceable the human sources of requirements, requirements related information, and requirements related work. It consists of a web of relations among contributors (the agents of a contribution) and the artifacts resulting from their contributions. It also encompasses relations among contributors themselves, revealing organizational structure, and relations among artifacts, revealing semantic dependencies, as well as (derived) relations indicating notions such as social role, commitment, temporality, etc.
- Document centered models usually represent traces as relations between documents of different types. An example is SODOS (Horowitz and Williamson 1986b, 1986a) which also includes concepts for software life-cycle. Other examples are hypertext models like RETH (Kaindl 1993) and HYDRA (Pohl and Haumer 1995). Hypertext models are particularly suitable for capturing informal or originally unstructured information. Also, using hypermedia features they allow the assessment of information in its original format.
- Database guided models are used to register trace information on databases for later retrieval. The work of Toranzo and Castro (1999) presents a quite complex database model. The idea is for the model, tuned by user needs, to be used by great diversity of users and environments, selecting the more useful entities for each occasion.

4.2 Tracing Methods

Tracing methods define an organized set of activities and establish the procedures necessary to make artifacts related to each other according to some model. The model need not be explicit. Instead the procedures may be centered around some general concept of related elements and make use of traceability techniques.

- The RADIX method (Yu 1994) is centered around a document based traceability model. Trace units are marked parts of texts inside documents. There are a pre-defined number of marks to delimit parts of the text specifying their nature, e.g., requirement, explanation, keyword. Traces are defined by explicitly marking a trace unit with references to other texts and documents. The method itself consists of several steps to organize the production, use, and verification of these documents.

4.3 Tracing Techniques

We classify as tracing techniques the specific activities and their resulting products used for requirements tracing.

- Traceability Matrices. They are used to relate requirements to other software development artifacts. Usually requirements are listed along rows and the other artifacts like specification, design modules, and programs along columns. The order may be reversed. At each crossing of a row and a column a mark is made if the respective requirement and artifact are related. It is possible to envisage more sophisticated ways of using traceability matrices, e.g., using different marks to indicate different kind of relationships.
- Cross References and Indexing Schemes. They are references made across several artifacts to indicate links between them, or lists of indices containing the related artifacts for each one. Cross references are used, for example, as part of the RADIX method and indexing is implemented in READS (Smith 1993).

Tracing techniques may easily be incorporated into several methods and may also be used in conjunction with different models. Indeed, some form of traceability matrices is implemented in almost all current traceability tools.

4.4 Tracing Languages

There are a number of specification languages with features allowing references to requirements. For example, the statements `traces-to` and `traces-from` in the requirements specification language RSL (Alford 1977) and the statement `by-requirements` in the prototype system description language PSDL (Luqi and Steigerwald 1991). However, they are not languages primarily intended for requirements tracing.

- Database query languages. At the moment, most of the current requirements tracing tools use conventional database query languages to inspect and retrieve trace information from their databases.
- Regular expressions. They are used as part of the tracing model implemented by TOOR (Pinheiro and Goguen 1996). The following section describes their use in more detail.

4.5 Tracing Processes

Tracing processes are models of the software development process incorporating ways to trace the process elements. This is a very promissing way to deal with the informal needs of requirements tracing. Jarke (1998) identifies the adaptability to project specific needs as a critical issue and Dömges and Pohl (1998) cite integration into the process, adaptation to the situation, and support for organizational knowledge creation as desirable features of tracing environments. Other types of process information that could be traced are assignment of tasks to individuals and organizational hints to specific expertise (Rose 1998).

A recent work in this direction is the requirements baseline model proposed by Leite (1995). In this model a baseline allows the control of the evolution of software artifacts and the development phases. The model is able to distinguinsh among releases and configurations and has proved to be adaptable: a work on scenario based software development has been successfully carried out using the structure prescribed by it (Breitman and Leite1999, Leite et al. 1998).

4.5.1 Tracing as a By-Product Many of the tracing information related to software artifacts may be captured as a by-product of the development process. Therefore, tools designed to automate the development are good sources of traceability information. CASE (Dawson and Dawson 1995) and IPSE (Tombros and Geppert 1995, Finkelstein et al. 1995) tools are particularly suitable for this purpose, naturally integrating and maintaining information related to the activities and processes they support. However, traceability is not their primary concern and, therefore, better support is still given by specific traceability tools. This is reflected in the existence of interfaces to CASE tools from almost all industrial requirements management systems. A good and well documented example of the automatic production of tracing information as a result of development practices is given by the NATURE project (Jarke et al. 1993, Pohl and Jacobs 1994).

5 TOOR's Approach

This section describes the TOOR's approach to traceability. TOOR is a tool implementing an abstract traceability model that may be viewed as consisting of three languages: a language to define structures of related objects, a language to express relationships, and a language to define module structures. A consequence of the abstract nature of the model is that its use in a particular situation requires instantiation by a concrete tracing model. Using our model to implement an IBIS-like structure of related objects is different from using it to implement a structure of related objects following the Contribution Structures approach. In addition to its many distinguishing features our model address the two points discussed above: it makes possible from inside a single environment to trace information according to relations not thought of in advance, and it provides structuring facilities that can be made as general or specific as is convenient.

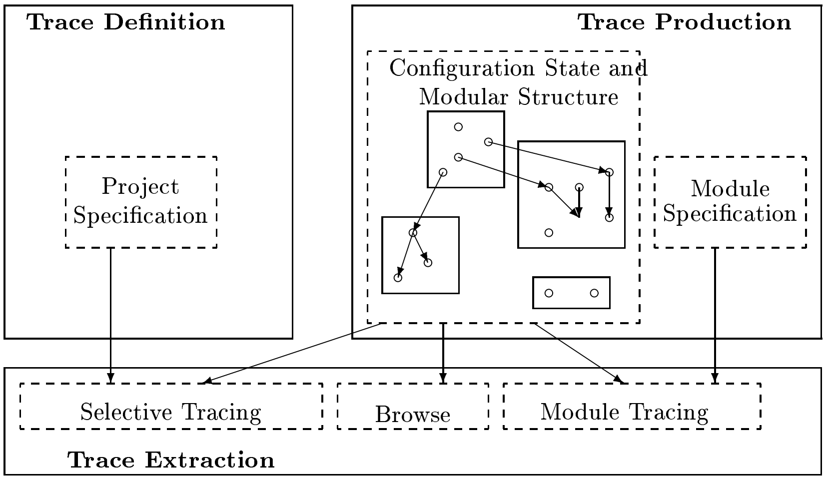

Fig. 1. TOOR 's abstract tracing model components

Figure 1 illustrates the model. The definition language is basically the FOOPS (Socorro 1993, Rapanotti and Socorro 1992) specification language with constraints for the way traces and trace units should be defined. Each concrete instantiation of the model, i.e., each specification defining particular traces and trace units, is called a *project specification*. The configuration state consists of the actual objects (trace units) and relations (traces) registered using TOOR . Projects are developed in a modular way and objects are created and used in the context of modules. The module specification language is represented as part of *trace production* to suggest that a project structure need not be defined in advance, rather it may be the result of the development itself with modules being dynamically created and imported. The extraction features comprise a browse mode to inspect the configuration state, an interactive (or modular) tracing mode to trace according to the project structure, and a selective mode, using regular expression to express patterns of related objects. The following sections discuss the model features in an informal way. A more technical presentation is given in (Pinheiro and Goguen 1996) and the details are elaborated in (Pinheiro 1997).

5.1 The Project Specification

The first step in using TOOR 's traceability model is to define a project specification. The project specification instantiates the TOOR 's abstract traceability model with a concrete one. It is a formal specification written in FOOPS containing the definitions of the objects that can be traced during the development of a project and the definitions of the relations intended to capture traces of one object on another. The project specification also defines an environment for the project, which consists of the menus and templates that TOOR supplies to register the actual objects used in the project development. Each class for the objects we wish to trace, such as *People* and *Statement*, should be specified and each relation we require, such as *Assert* between *People* and *Statement* objects, should also be specified as a class in the project specification.

5.1.1 The Use of Abstract Data Types

TOOR allows the specification of abstract data types for the value of objects' attributes. Thus, for instance, a data type for files may be defined such that a file associated to an attribute is not just a reference pointer; it is an actual value for that attribute that may be accessed and modified via the operations specified for the corresponding data type. Moreover, since the operations specified for a data type can be used in regular expressions, external information can be actively used when tracing objects.

> The following module defines a class of stored documents whose contents in TOOR are taken directly from the file in which the document is stored. The module declares a class `StoredDoc` as a subclass of `Document`, which is defined in the imported module `DOCUMENT`. The `content` attribute is redefined to get values of sort `T$File` and the other attributes are not modified . The sort `T$File` together with the operations to manipulate it are defined in the imported module `FILE`

```
omod STOREDDOC is
  class StoredDoc .
  extending DOCUMENT .
  protecting FILE .
  subclass StoredDoc < Document .
  at content : Document -> T$File [redef] .
endo
```

> Using this specification, every time an object of class `StoredDoc` is registered in TOOR , its `content` attribute should be filled with an existing file name.

5.1.2 Specifying Relations

R elations in TOOR are declared as classes containing at least the attributes `t$source` and `t$target` to hold the objects being related. Attributes `t$source` and `t$target` should be object valued attributes taking their values from objects of the classes representing the domain and codomain of the relation.

The parameterized module below defines a generic relation class `Relation`. It is intended to be instantiated with the modules containing declarations for the classes representing the domain and codomain of the relation being defined.

```
omod RELATION[X :: CTRIV, Y :: CTRIV] is
 class Relation .
 extending TOORLINK .
 subclass Relation < ToorRel .
 at t$source : Relation -> CTriv.X .
 at t$target : Relation -> CTriv.Y .
 at t$type   : Relation -> RelType .
endo
```

The parameter module `CTRIV` is a theory specifying syntactic and semantic constraints actual modules should satisfy to be used as arguments. In this case `CTRIV` contains only a single declaration for a class `CTriv`. Therefore, any module declaring a class may be used as an argument for `RELATION`. The module `TOORLINK` contains the declarations necessary to compute the mathematical properties of relations such as image, counter-image, etc., according to the relation type defined in the attribute `t$type`.

> The module below declares a relation `Extract` on objects of class `Document` to objects of class `Requirement`.
>
> ```
> omod EXTRACT is
>   extending RELATION[DOCUMENT,REQUIREMENT] *
>                                     (class Relation to Extract) .
>   var E : Extract .
>   ax t$type(E) = ordinary .
> endo
> ```
>
> The symbol `*` is a rename operator. In this case it renames the class `Relation` to be `Extract`. There is an automatic mapping that associates the class `CTriv.X` to the principal class of `DOCUMENT` and the class `CTriv.Y` to the principal class of `REQUIREMENT`. Therefore, the `t$source` attribute of `Extract` takes values from `Document` and its `t$target` attribute takes values from `Requirement`. The relation type is set to `ordinary` meaning that objects are related if there is a link between them.

5.1.3 Specifying Relation Axioms

T he basic constraint two objects should obey to be related under a particular relation is that the source object should be of the class specified for the relation domain and the target object should be of the class specified for the relation codomain. This constraint is automatically defined by the coarity of the `t$source` and `t$target` attributes, i.e., by the class of the objects that may be used as their values. A second common constraint is determined by the type of the relation. For example if a relation `R` is specified as antisymmetric and the objects `o1` and `o2` are already related under `R`, then it is not possible to relate them in the reverse order. However, it is possible to further restrict the way two objects may be related. For a relation named `R`, this further constraint is given by a method `may-R` with signature `may-R : ToorObj ToorObj -> Bool`, where `ToorObj` is a superclass for all classes in TOOR and `Bool` is the type for booleans. This method should be specified by the user if necessary.

The following module declares a relation called `State` on objects of class `People` to objects of class `DbReq` (for database requirements) which is considered to be declared in the module `REQUIREMENT`. In this case there is an explicit `view` mapping the class `CTriv` of argument `Y` to the class `DbReq` of `REQUIREMENT`.

```
omod STATE is
  extending RELATION[PEOPLE,view to REQUIREMENT is
                            class CTriv to DbReq . endv] *
                                          (class Relation to State) .
  me may-State : People DbReq -> Bool .
  var S : State .  var P : People .  var R : DbReq .
  ax t$type(S) = ordinary .
  cax may-State(P,R) = department(P) == technical .
endo
```

The relation name is `State` and, therefore, the method used to verify if two objects may relate is named `may-State`. The constraint axiom for `may-State` specifies that only people from the technical department can state database (`DbReq`) requirements.

Axioms and object's properties are evaluated each time an object is created. In this way two objects may be considered related even if there is no direct link between them. For example, if a relation `R` is declared as transitive and the object `o1` is related to object `o2` under `R` and object `o2` is related to object `o3` under `R`, then the object `o1` is considered as related to object `o3` even if there is no link between them, i.e., even if the user does not explicitly relate them.

Another type of indirect link by which objects may be considered related even if there is no link between them is given by composition of relations. For example, one may want to declare that a requirement r is *Derived* from a document d if it is derived from a document d' and d' is part of d, i.e, considering $\mathit{PartOf} \subseteq \mathit{Document} \times \mathit{Document}$ and $\mathit{Derive} \subseteq \mathit{Document} \times \mathit{Requirement}$ one may want to specify

$$(\forall d, d' \in \mathit{Document})(\forall r \in \mathit{Requirement})$$
$$((d,r) \in \mathit{Derive} \ \text{ if } \ (d',r) \in \mathit{Derive} \ \text{ and } \ (d,d') \in \mathit{PartOf}).$$

To specify that objects are related under a relation `R` through composition of other relations it is necessary to explicitly declare a method `is-R`, with signature `is-R : ToorObj ToorObj -> Bool`, together with its corresponding axioms.

The situation described above may be specified in the following way:

```
omod DERIVE-PARTOF is
  extending DERIVE .
  extending PARTOF .
  me is-Derive : Document Requirement -> Bool .
  var D : Document . var R : Requirement .
  cax  is-Derive(D,R) = member(image('Derive,D),R) or
                 inter(image('PartOf,D),image-1('Derive,R) =/= empty .
endo
```

As usual, the modules `PARTOF` and `DERIVE` import the generic module `RELATION` to declare the relations `PartOf` on `Document` to `Document` and `Derive` on `Document` to `Requirement`, respectively. The method `is-Derive` is specified to result in `true` if there is a common element in the image of `D` under `PartOf` and in the counter-image of `R` under `Derive`. The functions `member`, `inter`, `image`, and `image-1` for membership and intersection of sets and image and counter-image of relations, respectively, are all defined in the module `TOORLINK` which is imported by `RELATION`.

5.1.4 Evolving and Adapting Project Specification

O nce a project specification is defined the user may start a specific project by creating objects and relating them. Of course, the project specification may be expected to evolve. A new class may be considered necessary to better register the relations between objects or an existing class may be modified. For example, on reaching the design phase there may be a need to register technical manuals as a specific subclass of `Document` or to create a new class to hold entity-relationship diagrams. Also, after some time it may be noticed that some classes have attributes that are never used and removing these attributes would facilitate the registration of objects. Relations may also be perceived as unnecessary for a particular project, or not detailed enough. Also, there may be a need for new relations. Therefore, the use of a project specification makes possible not only the definition of different traceability models, but also a given traceability model may evolve over time to cope with particular traceability needs and to adapt to changing situations.

5.2 The Tracing Language

Objects in TOOR may be traced in different ways. One of powerful way is to use regular expressions to describe patterns of related objects. These patterns are expressed in terms of object and relation identifiers combined by means of regular operators. For example, the regular expression

```
(req1 | req2) Derive req3
```

consists of the object identifiers `req1`, `req2`, and `req3`, and the relation class identifier `Derive`. The regular operator `|` expresses an alternative and concatenation of identifiers is expressed by a space between them. Given a regular expression, TOOR searches its configuration of object and relations for objects related in the way described by the pattern. In the example above an object `req1` related to `req3` by `Derive` would match the pattern.

5.2.1 Regular Definitions for Properties

T he pattern matching procedure of TOOR does not use only object and relation identifiers. Regular expressions are extended to consider object's properties and the methods and axioms defined for each class. A sample of some possible regular expressions is given below:

```
<*-input> Derive [Requirement]                          (1)
Doc1 Extract ; Derive [Requirement]                     (2)
Doc1 ToorRel+ reqA                                      (3)
Doc1 Extract [[Requirement x if priority(x) == high]]   (4)
```

The regular expression (1) is matched by any object whose identifier matches the regular expression `*-input` and is related by `Derive` to an object of class `Requirement`; (2) is matched by an object identified by `Doc1` related to an object of class `Requirement` by a composition of relations `Extract` and `Derive`; (3) is matched by object `Doc1` related to object `reqA` by a chain consisting of one or more relation objects (every relation in TOOR is a subclass of `ToorRel`); (4) is matched by object `Doc1` related by `Extract` to objects of class `Requirement` having a high priority.

The use of axioms in TOOR 's regular expressions allows the tracing of information according to relations not thought of in advance. This is possible because, as exemplified in Example 5.1.3, relations may be specified by axioms that may be added or removed from a project specification in order to conform to a particular situation or line of reasoning. Another instance of this 'tracing in the fly' is given below.

5.2.2 Active Use of Hypermedia Information for Tracing

T he definition of abstract data types and the use of operations and methods associated to them as part of regular expression queries promote the active use of hypermedia information for tracing purposes. For example, a data type `Video` for files containing video scenes may be defined together with an operation `is-there-scene` that given two `Video` files `A` and `B` returns `true` if the scenes in `A` are contained in `B` and `false` otherwise. Thus, if requirements objects are defined to have an attribute `video-evd` of sort `Video` the user may associate a video file as a value for this attribute. Moreover, if a certain requirement `req-A` is being analyzed the user may want to retrieve all other requirements possessing similar video evidence, i.e., all requirements for which the video file used as a value for `video-evd` contains the scenes of the video file associated to `req-A`. This would simply be expressed as

```
[[Requirement x if
 is-there-scene(video-evd(req-A),video-evd(x)) == true]]
```

Note that the retrieved requirements may not be related in the sense of having a physical link between them. The relation between `req-A` and the retrieved requirements is that they share a common piece of video evidence; in this example this relation was thought of, and expressed as a regular expression query, at the very moment of analyzing `req-A`. Of course, the above query may be enlarged to consider other kinds of chains to, for instance, retrieve all people related to `req-A` or even people related to each matched requirement.

```
[People] ToorRel [[Requirement x if
 is-there-scene(video-evd(req-A),video-evd(x)) == true]]
```

This sort of direct use of hypermedia information to trace objects is not possible with current industrial tools. Most of them have mechanisms to import/export graphics into the documents they produce but they do not use this type of information to actively trace objects.

5.3 Structuring Projects

TOOR modules are used to structure projects by providing scopes for the creation and use of objects in a project development. They are not intended to structure the system being developed. The main purpose of TOOR modules is to establish a space for object creation and a policy for object use. Objects in TOOR are registered inside modules and may be used inside the module in which they are registered. Also, module importation allows an object registered in one module to be used in another one. TOOR modules allow the user to structure the project development by defining particular signatures for each module, thus restricting the type of objects that may be registered in them. For example, a module `REQ-ELICIT` may be specified to have a signature consisting of classes `People`, `Document`, `Requirement` and the relation classes involving them. In this way an object of class `Spec` cannot be registered in `REQ-ELICIT` and, consequently, the user is forced to register it in the appropriate module, say `SYSTEM-SPEC`. However, `SYSTEM-SPEC` may well import `REQ-ELICIT` so that relations between requirements and specifications can be made.

TOOR has an *interactive tracing* mode that, given an object to be trace, is used to traverse the project module structure looking for objects related to the given one. The user may control the traversal by choosing the objects and modules to be used in each step and even backtracking to previous points. Another way of using modules for tracing is explained below.

5.3.1 Using Modules to Restrict the Tracing Space

A given module or set of modules may be used to restrict the tracing space. For example, when tracing with regular expressions the objects and relations that are considered to match a certain pattern are only those registered or being used in the selected modules. This is useful when one wants to restrict the tracing to certain parts of the project. For instance, if an error is found, during the coding phase, in a certain source code object it may be advisable to trace it first looking for relations to other source code objects, going back at most to specification objects. If this is the case and if the project is structured in a way that these objects are registered in specific modules, then the appropriate modules may be selected to restrict the tracing. Another example would be the tracing of requirements objects excluding those objects registered in modules created only to hold tentative requirements. The choice of which modules to use to restrict a trace depends on the purpose of tracing and, of course, on the way a project is structured.

Restricting the trace to selected modules has the advantage of giving additional freedom in the way regular expressions are written. Since the class of any object in TOOR is a subclass of `ToorObj` and the class of any relation is a subclass of `ToorRel`, the regular expression below is matched by any object related by any chain of relations to the object identified by `programA`.

```
[ToorObj] ToorRel+ programA
```

By selecting the appropriate modules one can use such a general expression and be sure that no object outside the selected modules will match the pattern.

6 Conclusions

The presentation of TOOR's abstract tracing model highlighted some of its features intended to meet the informal needs of requirements tracing as discussed in the earlier sections of this paper. One question that may be asked is to what extent traceability can be automated in a useful way. After all, design and requirements are recognized as 'wicked problems' (Bubenko 1995, Yeh 1991, Sommerville 1989) where the problem domain and the solution domain overlap and solution for one aspect of the problem usually results in subtle ramifications, unfolding a chain of previously unsuspected problems. In such an intangible and inherently inconsistent environment it may be argued that traceability will always be achieved through non-automated initiatives. It may be the case that, as Wieringa (1995) points out "the world is the ultimate traceability tool". Despite the platonic connotations of such a statement, this view is in accordance with some of the traceability practices highlighted in the preceding sections. According to our view it is not possible to fully automate requirements traceability. However, we believe that progress can still be made, and that the answer lies in an adequate mix of the formal and informal aspects of software development and in a smooth transition between the two.

References

Mack W. Alford. A requirements engineering methodology for real-time processing requirements. *IEEE Transactions on Software Engineering*, SE-3(1):60–69, January 1977.

Breitman, K.K. and Leite, J.C.S.P.: Processon de Software Baseado em Cenários. *II Workshop on Requirements Engineering (WER99)*, pp. 95-105, Buenos Aires, September 1999 (in Portuguese).

J. Bubenko. Keynote address. In *Proceedings, Second IEEE International Symposium on Requirements Engineering*, York, England, March 1995. IEEE Computer Society Press.

Jeff Conklin and Michael L. Begeman. gIBIS: A hypertext tool for exploratory policy discussion. *ACM Transactions on Office Information Systems*, 6(4):303–331, October 1988.

C. W. Dawson and R. J. Dawson. Towards more flexible management of software systems development using meta-models. *Software Engineering Journal*, 11(3):79–88, May 1995.

Dömges, R., Pohl, K.: Adapting Traceability Environments to Project Specific Needs. Communications of the ACM **41**:12 (1998) 54–62

Anthony Finkelstein, Jeff Kramer, and Bashar Nuseibeh, editors. *Software Process Modelling and Technology*. Research Studies Press / Willey, 1995.

Joseph Goguen. The dry and the wet. In Eckhard Falkenberg, Colette Rolland, and El-Sayed Nasr-El-Dein El-Sayed, editors, *Information Systems Concepts*, pages 1–17. Elsevier North-Holland, 1992. Proceedings, IFIP Working Group 8.1 Conference (Alexandria, Egypt).

Joseph A. Goguen. Formality and informality in requirements engineering. In *4th International Conference on Requirements Engineering*, pages 102–108. IEEE Computer Society Press, April 1996.

Orlena Gotel. *Contribution Structures for Requirements Traceability*. PhD thesis, Department of Computing, Imperial College of Science, Technology, and Medicine, University of London, August 1995.

Orlena Gotel and Anthony Finkelstein. Contribution structures. In *Proceedings, Second IEEE International Symposium on Requirements Engineering*, pages 100–107. IEEE Computer Society Press, March 1995.

Orlena Gotel and Anthony Finkelstein. Revisiting requirements production. *Software Engineering Journal*, 11, 1996.

E. Horowitz and R.C. Williamson. SODOS: a software documentation support environment - its definition. *IEEE Transactions on Software Engineering*, 12(8):849–859, 1986.

Ellis Horowitz and Ronald C. Williamson. SODOS: A software documentation support environment – its use. *IEEE Transactions on Software Engineering*, 12(11):1076–1087, November 1986.

Leite, J.C.S.P and Oliveira, P.: A Client Oriented requirements Baseline. *Proceedings of the Second International Symposium on Requirements Engineering (RE95)*, pp 108-115, IEEE Computer Society Press, York, March 1995.

Leite, J.C.S.P. et al.: Enhancing a Requirements Baseline with Scenarios. *Requirements Engineering Journal*, vol. 2, pp. 184-198, Spring-Verlag, December 1998.

Jarke, M.: Requirements Tracing. Communications of the ACM **41**:12 (1998) 32–36

M. Jarke, J. Bubenko, C. Rolland, A. Sutcliffe, and Y. Vassiliou. Theories underlying requirements engineering: An overview of NATURE at genesis. In Stephen Fickas and Anthony Finkelstein, editors, *Proceedings, IEEE International Symposium on Requirements Engineering*, pages 19–33, California, January 1993.

H. Kaindl. The missing link in requirements engineering. *ACM SIGSOFT Software Engineering Notes*, 18(2):30–39, 1993.

Luqi and Robert Steigerwald. CAPS as a requirements engineering tool. *Communications of the ACM*, pages 75–83, 1991.

James A.H. Murray, Henry Bradley, W.A. Graigie, and C.T. Onions, editors. *The Oxford English Dictionary*. Oxford University Press, Oxford, UK, 1989.

Francisco A. C. Pinheiro. *Design of a Hyper-Environment for Tracing Object-Oriented Requirements*. PhD thesis, Programming Research Group, Oxford University Computing Laboratory, Oxford, UK, (1996 forthcoming).

Francisco A.C. Pinheiro and Joseph A. Goguen. An object-oriented tool for tracing requirements. *IEEE Software*, 13(2):52–64, March 1996.

Klaus Pohl and Peter Haumer. HYDRA: A hypertext model for structuring informal requirements representations. In *Second International Workshop on Requirements Engineering: Foundations of Software Quality (REFSQ'95)*, Jyvaskyla, Finland, June 1995.

Klaus Pohl and S. Jacobs. Traceability between cross-functional-teams. In *1st International Conference on Concurrent Engineering, Research and Application*, Pittsburgh, USA, August 1994.

Klaus Pohl and Peter Peters. Workshop summary – second international workshop on requirements engineering: Foundations of software quality (REFSQ'95). *ACM SIGSOFT Software Engineering Notes*, 21(1):31–34, January 1996.

C. Potts and G. Burns. Recording reasons for design decisions. In *Proceedings, 10th International Conference on Software Engineering*, pages 418–426, April 1988.

Balasubramaniam Ramesh and V. Dhar. Supporting systems development using knowledge captured during requirements engineering. *IEEE Transactions on Software Engineering*, 18(6):498–510, June 1992.

Balasubramaniam Ramesh and V. Dhar. Representing and maintaining process knowledge for large scale systems development. *IEEE Expert*, 9(2):54–59, 1994.

Balasubramaniam Ramesh and Luqi. Process knowledge based rapid prototyping for requirements engineering. In Stephen Fickas and Anthony Finkelstein, editors, *Proceedings, IEEE International Symposium on Requirements Engineering*, pages 248–255, California, January 1993.

Balasubramaniam Ramesh and Luqi. An intelligent assistant for requirements validation for embedded systems. Submitted to *IEEE Expert*, 1994.

Lucia Rapanotti and Adolfo Socorro. Introducing FOOPS. Technical Report PRG-TR-28-92, Programming Research Group, Oxford University Computing Laboratory, 1992.

Gail L. Rein and Clarence A. Ellis. rIBIS: a real-time group hypertext system. *International Journal of Man-Machine Studies*, 34(3):349–367, March 1991.

Rose, T.: Visual Assessment of Engineering Processes in Virtual Enterprises. Communications of the ACM **41**:12 (1998) 45–52

Thomas J. Smith. READS: A requirements engineering tool. In Stephen Fickas and Anthony Finkelstein, editors, *Proceedings, IEEE International Symposium on Requirements Engineering*, pages 94–97, California, January 1993.

Toranzo, M. and Castro, J. The Multiview++ Environment: requirements traceability from the perspective of stakeholders. *II Workshop on Requirements Engineering (WER99)*, pp. 95-105, Buenos Aires, September 1999.

Adolfo Socorro. *Design, Implementation and Evaluation of a Declarative Object-Oriented Programming Language.* PhD thesis, Oxford University Computing Laboratory, 1993.

Ian Sommerville. *Software Engineering.* Addison-Wesley, 1989.

Dimitrios Tombros and Andreas Geppert. A survey of database support for process-centered software development environments. Technical Report ifi-95-28, Computer Science Department, University of Zurich, 1995.

Roel J. Wieringa. An introduction to requirements traceability. Technical Report IR-389, Faculty of Mathematics and Computer Science, University of Vrije, Amsterdam, September 1995.

R. T. Yeh. System development as a wicked problem. *International Journal of Software Engineering and Knowledge Engineering*, 1(2):117–130, 1991.

Weider D. Yu. Verifying software requirements - a requirement tracing methodology and its software tool - RADIX. *IEEE Journal on Selected Areas in Communications*, 12(2):234–240, 1994.