A scoped approach to traceability management

Patricia Lagoa,*, Henry Muccini, Hans van Vliet

*VU University Amsterdam, Department of Computer Science, De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands

Dipartimento di Informatica, Università dell’Aquila, Via Vetoio 1, 67100 L’Aquila, Italy

Article info

Keywords:
Traceability paths
Software product line
Traceability issues
Software process management

Abstract

Traceability is the ability to describe and follow the life of a software artifact and a means for modeling the relations between software artifacts in an explicit way. Traceability has been successfully applied in many software engineering communities and has recently been adopted to document the transition among requirements, architecture and implementation. We present an approach to customize traceability to the situation at hand. Instead of automating tracing, or representing all possible traces, we scope the traces to be maintained to the activities stakeholders must carry out. We define core traceability paths, consisting of essential traceability links required to support the activities. We illustrate the approach through two examples: product derivation in software product lines, and release planning in software process management. By using a running software product line example, we explain why the core traceability paths identified are needed when navigating from feature to structural models and from family to product level and backward between models used in software product derivation. A feasibility study in release planning carried out in an industrial setting further illustrates the use of core traceability paths during production and measures the increase in performance of the development processes supported by our approach. These examples show that our approach can be successfully used to support both product and process traceability in a pragmatic yet efficient way.

© 2008 Elsevier Inc. All rights reserved.

1. Introduction

The notion of traceability has been thoroughly discussed in many software engineering communities, and the problem of tracing information is certainly not new. In the requirements engineering field, Gotel and Finkelstein (1994) defined traceability as the “ability to describe and follow the life of a requirement, in both a forward and backward direction, i.e., from its origins, through its development and specification, to its subsequent deployment and use”. In this area, a lot of effort has been invested on analyzing how requirements, architectures and implementation can be mutually related (Berry et al., 2003; Nuseibeh, 2001; Grünbacher et al., 2004). The introduction of multi-view modeling (Kruchten, 1995) has required a broader definition of traceability, as a means for modeling the relations between any software artifact in an explicit way.

Traceability support is required whenever large and/or complex software systems are to be maintained (Watkins and Neal, 1994; Gotel and Finkelstein, 1994; Rigaux and Eymans, 2003; Spanoudakis and Zisman, 2005; Asuncion et al., 2007). It may become critical to project success (Watkins and Neal, 1994). In general, whenever we need to maintain systems over time and space we expect to be able to reuse them after their release. This means that we implicitly rely on up-to-date documentation, i.e., on the ability to “trace back”, e.g., in which design modules a given requirement is realized, which technology is required to deploy a certain component, which rationale led to certain architectural decisions. Being able to trace such information in a reliable and efficient way is essential for reuse and evolution. If explicit traceability is absent, we rely on the experience and memory of the responsible developers who may become unavailable or must spend additional time and effort in recalling decisions taken in the past.

On the other hand, traceability is very expensive, both to reconstruct and to maintain. It is expensive to reconstruct: to recover lost information is very time consuming and difficult, if at all possible. Also, the complete picture of traces that exist for a complex system is quite often very large and intricate. Besides, the advantages of reconstructing these traces are not immediate and are possibly perceived only by others. Traceability is also expensive to maintain: similar to the more general problem of software documentation, if traceability links are kept manually, they are simply not updated or just forgotten as soon as some development deadline approaches.

This trade-off between necessity and cost is not resolved yet. Automation can definitely contribute (Egyed, 2003; Cleland-Huang et al., 2005; Asuncion et al., 2007). Nonetheless, this is possible only partially and in very specific contexts. What and how to
automate depends very much on the domain, the current practice within the developing company, and the knowledge and experience of developers.

In this paper, we present an approach to customize traceability to the situation at hand. Instead of automating tracing, or representing all possible traces, we scope the traces to be maintained to the activities stakeholders must carry out. We show its general applicability by means of two examples, one in academia in Software Product Line (SPL) engineering, and a second one in industry in process management. These examples illustrated that our approach can be successfully used to support both product and process traceability.

Before providing a more informative description of the paper goals, we introduce some definitions clarifying the terminology used hereafter.

1.1. Definitions

Generalizing the well known definition of requirements traceability given by Gotel and Finkelstein (1994), traceability is the ability to describe and follow the life of a software artifact through its life cycle in both a forward and backward direction.

Traceability is described by the links that connect related artifacts. For example, we can document requirements (R), design (D) and its implementing system components (SC), and the links between R and D, and between D and SC. Examples of such links are given in Fig. 1, which is based on the well known work of Ramírez and Jarke (2001).

By using these links we can trace the relations that exist among the documented artifacts. For example, the dependency between system components and requirements is represented by the link "depend-on" and modeled as follows:

\[ R \rightarrow D \rightarrow SC \]

This gives us a traceability path from SC to R.

The same pair of software artifacts can be related by multiple links at the same time. For example, next to "depend-on" Fig. 1 shows that between requirements and design there also are links labeled "drive", "modify" and "influence". This means that between the same two artifacts, say again requirements and system components, there can simultaneously be multiple traceability paths, each using different links. For example, a second traceability path is

\[ R \rightarrow D \rightarrow SC \]

The decision about which of these links to record depends on the anticipated use of traceability. For example, if our goal is to support the evolution of the existing systems, we record links like "drive" and "modify" that help us to select the design solutions that fulfill a certain list of requirements. If instead we want to support assessment activities, we model the "influence" link that helps us to reason about the impact of a certain set of requirements on the design.

In other words, the set of links we need to record depends on the specific usage we aim at.

As illustrated in Fig. 1, there are in general multiple links between the same two artifacts which serve the same use. For example, the following links can be used for evolution purposes; their combinations identify multiple traceability paths between requirements and system components:

\[ R \rightarrow D \rightarrow SC \]

Our discussion is not about specific links as such, but only about the set of links that suit a specific purpose. We therefore collapse such a set of links to one "generic" link, and consequently the set of traceability paths for a specific usage also collapses to a single path.

By knowing in advance the use we aim at, we can consciously decide on which traceability links to document, and can exclude those that are not necessary. In this way, we can focus on a core set of traceability paths, i.e., the necessary set of paths that can be used for our specific aim.

Core traceability paths are defined to solve some related traceability issues, i.e., matters to be decided upon and for which traceability support is needed.

1.2. Paper goal

In this paper we address the identified traceability issues in two different domains: SPL engineering and process management.

In SPL engineering, traceability support is required both between different software models (e.g., feature and structural models), and between different development phases (e.g., from domain engineering to application engineering). In SPL engineering, families of products are developed within a specific context. This means that, as favored in Dönges and Pohl (1998), we are in the position to filter the traces that are relevant for that context, i.e., the specific company, the application domain and the usage. As compared to "general-purpose" software engineering, this decreases the number of traces we need. Further, we focus on two software representations often used in SPL engineering, a feature-oriented view and a structure-oriented view. In this way we involve only two "core" views as principal documentation for traceability. Additional views are considered ancillary, e.g., used as decorations of these core views. Again, this decreases the number of traces we need.

In process management, traceability support is needed among the different activities, roles, and artifacts characterizing a software development process. This paper defines a conceptual process traceability framework enabling the support of explicit traceability among software process artifacts. An industrial study applies the framework to the software development process of NU-Biko, a small and dynamic Dutch IT company delivering software development, concept development and consultancy in the insurance, media and entertainment domains. By making explicit the different types of software process artifacts and by defining a focused and explicit traceability support, we show that new projects supported by the core traceability paths are accomplished in shorter time than old projects, i.e., the performance of new processes is better than the performance of old processes.

With our scoped approach we make traceability manageable and we build the basis for future research on how to support traceability. Core traceability paths stem from important issues that have to be dealt with in SPL and process engineering.

The paper is structured as follows: Sections 2 and 3 present the two illustrations of our approach. Each includes background information about the specific domain (SPL engineering and process

---

1 www.nubiko-development.nl.
management, respectively), followed by the identification of the traceability issues we encountered and the supporting traceability paths. Lastly, we illustrate how the traceability paths can be used to address the identified issues. The SPL engineering example in Section 2 uses an example from the literature to show the feasibility of our approach for answering typical traceability questions in the SPL-context. The example in Section 3 involves a study in an industrial context to illustrate the feasibility of our approach for addressing traceability in process management. Section 4 discusses related work. Section 5 summarizes the points made and discusses next steps to be taken.

2. A study in SPL engineering

In SPL engineering, traceability support is required both between different software models (e.g., feature and structural models), and between different development phases (e.g., from domain engineering to application engineering). Since we should trace the relations between product members and the product line architecture, as well as those among the product members themselves, a large set of traces should be kept. By taking a trace-all approach it may become impractical and unaffordable to maintain a consistent and aligned view on traceability links. For example, Jirapanthong and Zisman (2005) identify 10 different types of traceability relations among feature, sub-system, process, module, use case, class diagram, state-chart diagram and sequence diagram models, which may lead to hundreds of traceability paths.

In the study illustrated hereunder, we scope the traces to be maintained, to two software representations often used in SPL engineering, namely a feature-oriented view and a structure-oriented view. In this way, we involve only two “core” views as principal documentation for traceability, thus decreasing the number of traces we need.

2.1. Background on software product lines

Software product lines (SPL) represent a strategic approach to reuse, where components and other assets are identified in relation to specific application domains, architectures and scopes (as opposed to software reuse schemes that try to create assets as general as possible).

Quoting Clements and Northrop (2002), a software product line is “a set of software-intensive systems sharing a common managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way”. The idea behind a family-system approach is to derive a new system or application from a common set of assets (e.g., components, known requirements or design elements, models) in the same domain.

The main (core) models produced during product line engineering are feature models and structural models, where features specify domain-related system properties while structural models specify architecture-level decisions. A feature is “a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems” (Kang et al., 1990). It may also be viewed as a requirement or characteristic provided by members of the SPL (Gomaa, 2004). A structural model is generally used to identify components, interfaces and their interconnections and represents a structural (i.e., topological) description of the system software architecture. Components in an SPL may be mandatory (i.e., always required), optional (may or may not be present in a product), and variant (when different alternatives of the same solution are available (Jaring and Bosch, 2002)). For example, the engine is (still) a mandatory component of a car, while the air-conditioning is optional. Both can exist in multiple variants: engines can have different performance levels; air-conditioning can be automatic or manual.

Both feature and structural models can be utilized either during domain engineering or during application engineering. During domain engineering, commonality and variability of the entire family is defined. Commonality defines what different products have in common and guides the production of domain-specific core assets. Variability, instead, is defined as the ability to change or customize a software system (Jaring and Bosch, 2002). During application engineering, the single product is derived from the domain model, by reusing as much as possible the domain assets and by exploiting the commonality and variability of the product line (Böckle et al., 2005). Product derivation involves a quite complex decision-making process to exploit the commonality and to bind the variability according to the product requirements.

2.2. Traceability issues

In this section we describe three important traceability issues raising during product derivation. Those issues relate the representation of commonality and variability in terms of features (feature models) and the representation of solutions in the form of interconnected components (structural models). Both are represented at the family level and at the product level.

The three issues are visualized in Figs. 2(a) and (b), and 3, respectively, where the boxes show the representations we consider, and the labeled arrows indicate the traceability links we need for solving the three issues.

Fig. 2. (a) Traceability links between feature and structural models and (b) traceability links between SPL level and product level models.

Fig. 3. Traceability links between generic and concrete structural models.


Fig. 2 also depicts a dashed arrow from feature models to structural models. This represents a typical usage in SPL engineering, that is, the derivation of a specific product from the SPL. We use this typical usage to reason about the core set of traceability paths between feature models and structural models. As product derivation is typically top-down, the restriction to this typical usage justifies the existence of unidirectional links in Figs. 2 and 3. Though, in the same way typical usages other than product derivation are applicable, as illustrated in the industrial study presented in Section 3.

2.2.1. Issue I1 – Between features and structural elements

The need to trace the correspondence between different views on the same software system is well known (Clements et al., 2002). For example, in the field of requirements traceability the mapping between the problem and the solution space is achieved by traces between requirements and structure. If these traces are missing, the verification of compliance of the system with requirements or the assignment of requirements to components becomes very hard if not impossible.

In SPL engineering, we have a similar scenario for features and structural models: the reasoning about requirements desirable for a product line or an individual product is often based on feature representations (Lago and van Vliet, 2005). We need to select a set of desirable features, and then derive the system that is able to deliver these features. To be able to carry out this derivation, or at least to select the corresponding components, we need to make the mapping between features and their realization in the structural model explicit. During subsequent evolution we want to be able to reason about the impact of a change. For example, if we remove a feature, we need to know which product elements must be updated.

In summary, issue I1 calls for traceability links from feature representations to structural representations at both the product line level and the individual product level (arrows $a_1$ and $b_1$ in Fig. 2a).

2.2.2. Issue I2 – Between software product line level and product level models

An advantage of adopting an SPL engineering approach is the ability to benefit from the similarity between products in the same domain. To this end we need to document how the variability modeled at the product line level is resolved in the different products. In other words, we need to keep track of the traces between SPL models and individual products’ models. If these traces are left implicit, it is difficult to reason whether the properties of the SPL are reflected in a given product.

Summarizing, issue I2 calls for explicit traceability links between SPL level models and the decisions taken within each product (arrows $a_2$ and $b_1$ in Fig. 2b)).

2.2.3. Issue I3 – Between concrete and generic solutions

A common good practice is to transfer the experience and knowledge acquired in the past to a new development project. Reuse of successfully applied solutions are documented and codified in software patterns, architectural styles, reusable libraries, etc.; and pattern languages have been defined to indicate the relationships between these patterns. In a similar way, we need to represent the relationships between the patterns and the concrete solutions using them. These relationships are required to reconstruct which patterns are embedded in which existing systems. In SPL engineering this is especially important: during reuse we need to recognize the products that are compliant with certain reference architecture. For example, we want to assess which products behave according to an architectural style, or that a product reflects the structural properties of a certain architectural standard.

Issue I3 focuses on the structural models. We need to make explicit the links between generic (reusable) models (like architectural patterns) and the concrete architectural solutions (concrete models) that use them (arrows $c_1$ and $c_2$ in Fig. 3). This allows us to recognize reused patterns, styles and the like.

2.3. Example: The cooperative writing system product line

Lay and Karis (1991) define collaborative writing (CW) as “the process in which authors with different expertise and responsibilities interact during the invention and revision of a common document”. A CW system involves two or more people (geographically distributed) working together to produce a common document. Authors can be required to write a specific portion of a paper, interacting with other authors. A manager introduces/removes authors, divides the work of writing between groups, checks the quality of the resulting product.

The main features we identify for a CW SPL are shown in Fig. 4a (top, feature model):

- the AuthorHandler permits to read and write documents;
- the AccountHandler allows for login/logout handling, account creation and maintenance;
- the ManagerHandler allows for users insertion and deletion, and optionally, for groups insertion and deletion;
- the DocManager allows for document creation, assignment, and segmentation;
- the MessageHandler permits to chat or to be involved in a forum with other users (this is an optional feature);
- the CommunicationHandler allows for either synchronous or asynchronous communication (this is a variant feature);
- the InfoManager permits to get information about groups, documents and authors.

The main components realizing the described features are illustrated in Fig. 4b (top, structural model):

- The CW INTERFACE represents the interface between the cooperative writing engine and the system users. It is composed of four components:
  - ManagerInterface: manages the CW system by registering authors, setting the documents structure and assigning documents to authors;
  - Login: authenticates users;
  - AuthorInterface: GUI for authors;
  - Chat: optional, used for communication among users.
- The CW AUTHOR ENGINE implements all the services required from authors. It is composed of three components:
  - DocManager: implements the services for authors (open, read, write, upload and replicate);
  - Wrapper: optional, to save and read messages in specific formats;
  - DocHistory: keeps track of document changes.
- The CWManagerEngine, variant, provides services to the ManagerInterface component.
- The DBO represents the system database.
- The CWSyncronizer, variant, allows synchronous or asynchronous communication.

Fig. 4a and b – bottom – illustrates features and components (respectively) derived for a specific product.

2.4. Core traceability paths in SPL engineering

In this section we first elaborate each of the three types of traceability paths solving the issues described in Section 2.2. We then
describe how these traceability paths are used to carry out the derivation represented in Fig. 2 by the dashed arrow. Where needed, we illustrate our arguments with examples from the CW product line.

2.4.1. Traceability path: from feature to structural models and back

Issue I1 addresses the explicit documentation of the dependencies between feature model and structural model of the SPL and the products. This issue is located along the arrows labeled $a_1$ and $b_2$ in Figs. 2 and 4.

The links between model elements determine if, e.g., a feature is supplied by one or by multiple structural elements (such as objects, interfaces or components). For example, in the product P1 in Fig. 4, feature AuthorHandler is partly implemented by components AuthorInterface, Login and DBO. The other way round, for a given component the dependencies denote the list of feature(s) it implements. For example, component AuthorInterface mentioned above also implements part of feature AccountHandler. Notice that Fig. 4 and this paper itself do not define a specific technique to show how features and structural models’ elements are linked together. The focus of this paper, in fact, is to address the need of explicit traceability links between SPL core models. Only when what to trace is made clear, specific traceability techniques can be worked out.

These traceability links can be modeled as bi-directional traceability links. If made explicit, we can use these traces between the feature model and the structural model at both the family and product level: the SPL level traces from features to structure allow us to translate the features characterizing the domain into the architectural framework that models the general architectural decisions of the whole product line; the product level traces between features and structure show how concrete features (i.e., features implemented by some variants) are organized into a product architecture. Related work discussed in Section 4 shows that no unified solution for this has been proposed so far. In particular, while vertical arrows ($a_2$ and $b_2$) are often discussed in the literature, the realization of the horizontal ones ($a_1$ and $b_1$) is not so evident.

If traceability links for I1 are absent, it becomes particularly difficult to analyze the solution given in the structural model with respect to feature-related requirements. For example, if feature AuthorHandler is removed from the CW SPL, the identification of those components affected by this change may become expensive and error prone. The other way round, if the Chat component is removed (as done to realize product P1 in Fig. 4), we are unable to verify which features the resulting CW system is going to miss.

2.4.2. Traceability path: from software product line level to product level and back

Issue I2 concerns the documentation of where SPL commonality is realized in the different products. This issue is located along the arrows labeled $a_2$ and $b_1$ in Figs. 2 and 4. The models at the SPL level define the architectural framework shared by all products in the family. It includes common architectural elements that are then specialized and extended by each product to accomplish its specific requirements. SPL commonality can be specified in many different ways: they can be features, interface specifications, groups of objects, one or multiple components, etc.

It is important to be able to locate commonality in the different products, to, e.g., recognize how a certain common feature or common component has been specialized for different products. The other way round, for a selected product we need to discriminate...
between features/structural elements common to the whole family, and features/structural elements that belong to the specific product only. When the product architecture becomes complex in size and interconnections, it becomes very expensive to isolate the decisions that go back to the SPL architecture. There is a need for explicit traces to document them.

In product P1 in Fig. 4, the Chat optional component has been discarded, while the Wrapper optional component has been selected, and the CWManagerEngine – type1, AsynchCW, and AuthorView components are introduced. Explicit traceability links are needed to discern if a component is product-specific or part of the product line. For example, we need to keep track of the fact that the AsynchCW and the CWManagerEngine – type1 components are variants of the CWsynchronizer and CWManagerEngine components, respectively, and that the AuthorView is a product-specific component. A purely syntactical or graphical mapping is certainly not enough to clearly expose the information above. Explicit traceability links are necessary to explain family to product relationships.

If traceability links for I2 are absent, product specific and SPL generic requirements and choices are not explicitly separated, and hence we cannot discriminate them.

2.4.3. Traceability path: from concrete to generic structural models and back

Issue I3 concerns the documentation of the generic models embodied by the concrete structural solutions devised for the product line. The use of reference architectures and architectural and design patterns is well known, and some SPL engineering approaches (e.g., Hallsteinsen et al., 2003; Atkinson et al., 2002) explicitly address it. What in our opinion is still missing is the explicit and integrated representation of both the generic models (reference architecture and architecture/design patterns) and where and how they have been used within a certain concrete model. To this end, at the SPL level we make explicit the generic models used to shape the family architecture (arrow c1 in Fig. 3). At the product level we document the generic models used to shape the structure and properties of the individual products (arrow c2 in Fig. 3). In this way, we retain the documentation of the concrete system and the generic properties we want the system to yield.

This integrated representation requires to document the traceability links between generic models and concrete models. In this way, we can identify if certain architectural decisions taken in different products are actually of the same type, and in that case trace them back to the generic architectural framework.

For example, back to our CW SPL, if its architectural style is the “Blackboard” (see Fig. 5), we need to specify the style characteristics, create the links between the generic model and the product line and validate the family conformance to the generic model. The Blackboard is characterized by three main types of component (Corkill, 1991): the knowledge sources (KS), which are independent modules that contain the knowledge needed to solve the problem, the blackboard, which is a global repository containing input data and partial solutions, and the controller, which is responsible for managing the course of problem solving (e.g., coordinating the different KS). Next, we create traceability links between the Blackboard elements and the SPL structural model. For example, three main scenarios (drawn in the bottom area of Fig. 5) may be considered: (i) the Chat component is not selected (as in product P1), (ii) the Chat component is selected as a blackboard element, or (iii) it is selected as a KS element. In all cases, ManagerInterface and AuthorInterface are KSSs. One of the Blackboard style properties states that different KSSs may not communicate directly; they may communicate by sharing data available on the blackboard. In scenario (i), since the Chat is not selected, ManagerInterface and AuthorInterface communicate through the DBO that acts as blackboard element. In this way the property holds. In scenario (ii), since the Chat acts as a blackboard with data sharing, the property also holds. Finally, in scenario (iii) the Chat acts as KS and since it communicates directly with the other KSSs (the ManagerInterface and AuthorInterface) the property does not hold. Therefore, the resulting product does not conform to the blackboard generic model.

These dependencies can be modeled as bi-directional traceability links. Traceability links between generic models and concrete architectural solutions provide support for the verification of the general architectural properties (and hence qualities) brought in by applying general solutions. Also by reusing the generic models we can develop a library of recurring domain-specific common solutions.

2.5. Example usage: derivation of a product from a product line

To carry out the derivation of a product from SPL features we need the explicit links that allow us to resolve the SPL features (modeled in the feature model at the SPL level) into a desired product architecture (represented by the structural model at the product level). These links can be of four types, corresponding to the arrows labeled a, b, c, and d. In principle we can follow either of the traceability paths ab1, and a2b2.

If the derivation of the new product follows traceability path ab1, this means that we reason about the required features (and select the desired feature set) at the SPL level. The selected features correspond to a subset of the family architecture (arrow a1) that must be instantiated into a specific product architecture (arrow b1). In other words, we first use family features to select a struc-

---

**Fig. 5.** The blackboard style and its instantiation with scenarios (i)–(iii).
tural model for the whole SPL, and next customize this for a specific product. If instead the derivation of the new product follows traceability path $a_1b_1$, we first instantiate the required family features and obtain a product-specific feature model (arrow $a_2$). We then obtain the corresponding product architecture (arrow $b_2$). In other words, we first select the features for a specific product, and next select the appropriate structural model.

For instance, in the CW product line,

- Through the traceability path $a_1b_1$ we need to first identify how features are realized by components, and then we can select the SPL structural model elements we want for product P1. For example, as illustrated in Fig. 6 the WriteDoc feature (part of feature AuthorHandler) is implemented by components AuthorInterface, Login and DBO. The chat feature instead is implemented by components ManagerInterface, AuthorInterface, and Chat. When moving along arrow $b_1$ from the SPL-level structural model to product P1 in Fig. 6, both components for feature WriteDoc are selected, while the Chat component is disregarded.

- If instead we follow the traceability path $a_2b_2$, we need to first select the SPL features (and the variants) we want for product P1, and then select (and compose) the structural elements that realize those variants. In Fig. 6, the Asynchronous alternative feature is selected, which implies the selection of the AsynchCW variant component. Since the chat feature has not been selected, the Chat component is not selected either.

3. A study in software process management

In software process management, traceability support is required both among different artifacts/roles/activities (e.g., from requirements to design artifacts), and among artifacts, roles, and activities (e.g., from the requirements elicitation activity to its resulting requirements document artifact). As pointed out by Maeder et al. (2007), keeping traceability links during the whole software development process and integrating such links into development methods is mandatory. However, a trace-all approach may produce a too complex and difficult traceability matrix (as exemplified in Maeder et al., 2007).

In the study illustrated hereunder we present a shift in focus from product- to process-traceability support that can be supported by our approach without problems. In process traceability we use process models as core models, and identify the core traceability paths needed to support the work of the involved stakeholders. For each stakeholder (i.e., manager and developer) we have identified the core traceability paths supporting each standard activity they carry out. These core traceability paths have been further used to build four traceability views supporting such activities. In the study description given here we illustrate one of the four views.

![Fig. 6. Highlighting traceability links in the CW SPL.](image-url)
3.1. Background on software process management

A software development process defines who does what, when, and how to reach a certain goal (Kruchten, 2003). A “role” defines the behavior and responsibilities of an individual or group of individuals working in a team (i.e., the who). An “activity” assigned to a specific role consists in a set of “tasks” assigned to a person owning a certain role (i.e., the what). An “artifact” represents the output of the process, which can be created, modified and used (i.e., the goal). A “workflow” describes the sequence of activities, which must be executed by certain roles in order to produce certain artifacts (i.e., the when).

Process elements are clearly not independent. Decisions reflected in an artifact (or made by a stakeholder) strongly impact other artifacts (or other stakeholders). Thus, explicit traceability support is required to relate different process elements.

3.2. Traceability issues

In this section we describe two important traceability issues that arise during software process management. These issues concern the problems emerging whenever traceability between software process elements is missing. The identification of the traceability paths solving such issues depends on the activities that the various stakeholders typically carry out to monitor and/or to manage the software process.

3.2.1. Issue I1 – Between artifacts and tasks

Process models often treat tasks as first class elements and marginally focus on the other process elements, like the software artifacts taken as inputs and produced as outputs. In this way, it is especially difficult to trace back which requirements, design or implementation artifacts have been approved, rejected or produced by which tasks.

For example, system features (or requirements) decided with the customers and approved by the managers are often directly described in terms of tasks. Consequently, there is no distinction with the other types of tasks, like for instance sub-tasks created to break-down the work, or modification requests. Further, features requested by customers and then rejected by managers are not recorded at all, hence loosing the traces to the important knowledge about what has been rejected and for what reasons.

In a similar way, design artifacts are often not explicitly linked to the related tasks. Although software designs are being made, they are not explicitly maintained in the Software Configuration Management (SCM) system as part of the project documentation: they reside on disk somewhere, meaning that developers lack insight in the blueprint governing implementation, and implement their tasks the way they feel is right. This hinders traceability from requirements to code and leads to expensive integration sessions to re-align the work.

To solve issue I1 we need to represent all process elements as first class elements and make explicit the traceability links between tasks and the related artifacts. This allows us to recognize the process chain from requirements down to implementation. Further, it enables recording both positive and negative decisions (i.e., approved as well as rejected requirements) hence enabling the transfer of knowledge across projects.

3.2.2. Issue I2 – Between standard activities and project-specific process elements

To carry out a certain activity in a given project, a stakeholder needs to effectively navigate or gather the information about process elements. For example, to check the project progress, a project manager needs to identify the active tasks and the current status and their expected outputs.

Often, this type of traceability is possible only at the task-level. Therefore, the only way to trace, e.g., what changes are applied in the implementation of a selected task is to label all implementation entities in the SCM system with the same numeric task identifier kept by the task registration system. This usually is a manual and time-consuming process. Further, if no distinction is made among requirements, bug-reports and sub-tasks, simple searches like looking for the features implemented in a certain product becomes a difficult and tedious work.

Issue I2 requires the identification of core activities, and the access from a centralized point to the information about process elements necessary to accomplish such an activity. This allows stakeholders to find the right information in a focused way, so that time and effort be optimized.

3.3. The industrial context

NUBIKO is a small and dynamic Dutch IT company delivering software development, concept development and consultancy in the insurance, media and entertainment domains. It develops software for mobile (PDA’s and smart-phones), Web (online shops, Intranet and general web sites) and desktop environments. When we approached the company, the management explained that they were expanding their business. As a consequence, they were experiencing problems due to insufficient control over the progress, the results and related quality of the running process. As a solution they were seeking mechanisms to represent in an explicit way the process tasks and their expected/current inputs and outputs.

The software development processes in NUBIKO were not explicitly documented: the employees simply knew what to do and how to do it. We made the current practice explicit in the process model visualized in Fig. 7 where each project is defined in terms of tasks to be completed: tasks are identified by a set of stakeholders, and result in a software implementation.

Interviews carried out at NUBIKO revealed that three types of stakeholders are directly involved in its development processes: Managers communicate with new and current customers to understand and discuss the needed features to be developed in the products. Customers work together with managers to identify features. These are directly translated into tasks aimed at their realization. Developers are assigned with a number of tasks they implement in software. They can create new tasks for new features or split up the work into (smaller) sub-tasks. Of course multiple developers can work on shared features.

In NUBIKO, a task is defined as a set of related changes to separate objects (Belkhatir and Estublier, 1996). Even though they were not explicitly differentiated, there are four types of tasks: (new) feature requests, modification requests (for existing features), bug reports and sub-tasks. Tasks are created and maintained in a task registration system, which assigns each task with a numeric identifier and a number of fields that can be used (but seldom are) to describe the task.

An activity, instead, is a set of tasks assigned to a certain stakeholder. The activities common to all projects in NUBIKO are report creation, release planning and software development. These standard activities represent the “core” activities needing traceability and hence driving the definition of the core traceability paths that we need to record for NUBIKO. While details about these standard activities are given in Meijs (2008), we report here only one of them, the Release planning activity. Release planning is carried out

---

2 This term will be used hereafter to identify an activity, a role, or an artifact.

3 www.nubiko-development.nl.
together by managers and developers. Managers plan new projects or new releases for existing systems. This involves determining milestones and tasks and their association. Developers estimate the time needed to complete tasks, supply information about their availability and help completing task assignments.

Software implementations are stored in a SCM tool, supporting concurrent development and reconfiguration.

The old process in Fig. 7 includes the two traceability issues described in Section 3.2. These issues recognized in NUBIKO caused unnecessary project delays. These delays occurred in standard activities carried out in all their development projects: by making explicit the different types of tasks and by defining a focused and explicit traceability support they were expecting to shorten the time it takes to accomplish such standard activities, hence improving the overall project performance.

3.4. Core traceability paths in software process management

In this section we first elaborate on the traceability paths solving the issues described in Section 3.2 and depicted in Fig. 9. We then describe how these traceability paths have been used to define a traceability view (in Fig. 8) helping developers in activity Release planning. The study execution shows the improvement in performance achieved during a real industrial development process.

3.4.1. Traceability path: between process elements artifacts and tasks

Issue I1 concerns the documentation of the dependencies between the tasks carried out during a development process and the related software artifacts. It is important to keep these dependencies explicitly, so that the stakeholders involved (either during the process execution or after its completion) are able to efficiently trace back the requirements and the design decisions driving implementation.

In NUBIKO issue I1 has been addressed by representing artifacts as first class entities in the (new) software process model shown in Fig. 7. Here, the different types of tasks are made explicit and software design is integrated in the information flow. In this way, the potential traceability links are naturally visualized by the information flows labeled (a)-(f).

This new process model provides the solution to issue I1 described in Section 3.2: making explicit the requirements enables finding back the knowledge about what has been requested by customers, what has been rejected and accepted; making explicit the software design creates incentives to the developers using it as blueprint and enables alignment with design decisions previously taken.

3.4.2. Traceability path: between standard activities and process elements

Issue I2 (gaining overall traceability from a centralized point aiding standard activities) has been solved by defining the core traceability paths for the standard activities identified in Section 3.3. Namely, for each involved stakeholder (i.e., manager and developer) we identified the core traceability paths supporting each standard activity they carry out. These are further used to build four traceability views supporting the activities. In the following, we illustrate one of the four views with the corresponding core traceability paths, i.e., Release Planning by developers. The others (Report Creation by managers, Release Planning by managers and Development Overview for developers) are further described by Meijers (2008).

Planning for releases is done by managers working together with developers. When planning a release it is important to know what requirements and bug-reports have the highest market potential and the lowest implementation time. The market potential of requirements and bug-reports is determined by managers, and the estimates of the implementation time are determined by developers. In order to determine the implementation time for a requirement or bug-report, the developer needs to know what the implications of the changes are. These implications can be determined by analyzing the design and, in the case of a bug-report, the related source code and its implementers. This means that (see Fig. 8)

- for a requirement the developer needs traceability from requirements to design. It might also prove necessary to look into the discussion information regarding a requirement, as there may be vital information which is not entirely reflected in the design;
in a similar way, to estimate the implementation time of a bug-report the developer needs traceability to the design, too. In addition, (s)he also needs traceability to the code implementing the feature subject of the bug-report. Additional information is needed about the developer(s) who wrote the code in the first place, as that person might give a more detailed time indication.

This means that release planning demands traceability from requirements and bug-reports to the source code and their developers. With this information available any implemented feature or corrected bug can be quickly identified in the code together with all authoring information.

Specifically, in order to estimate the implementation time for a requirement or bug-report, a developer needs to know in which Developer Tasks the Requirement or Bug Report is (being) implemented. This can be traced via the Implemented in traceability links. The Code that belongs to the Developer Tasks can then be traced via the Code contained in traceability link, while the Developer who implemented the Developer Task can be traced via the Implemented by traceability link. For both Requirements and Bug Reports it can prove vital to have knowledge of the related Discussion information, for instance to gain a thorough understanding of what needs to be implemented. This information can be traced via the discusses traceability links. Finally, it is necessary to know in which Components the Requirements are implemented, what is the Rationale behind the decisions, and what Alternatives have been considered. This can respectively be traced via the traceability links Contained in, Rationale for and Alternative for.

The traceability path used to build the traceability view for release planning by developers is depicted in Fig. 8. The path is needed between developers (D) and code (C). The decomposition of the traceability path into individual links is shown inside the rectangle. These traceability links are not necessarily part of the path from customers to developer tasks, but are used during release planning to gather the needed additional information. An example of such a link used in the traceability view is the link between requirements (R) and alternatives (A).

3.5. Study setup

The goal of the study has been to investigate whether core traceability paths improve the execution of the standard project activities in practice. This is expressed by the following research question:

**Research question 3.1.** New projects supported by the core traceability paths are completed in shorter time than old projects, i.e., the performance of new processes is better than the performance of old processes.

To ensure that the study be representative for the company, we required its context to be industrially realistic. This means that the selected projects must be sufficiently complex in terms of size,
target domain and technology; they must be representative of the company business, and comparable. For these three reasons we have selected as pilots the Barcode Library project and the SPOT project. The details about these two projects, as well as how they met our complexity, representativeness and comparability requirements, are explained hereafter.

3.6. Pilot projects

The research study is aimed at researching the performance of the new processes versus the old processes with respect to traceability. For this study, we have measured the time it takes to accomplish a predefined task within a certain process (e.g., creating a report for a certain version of the software project). The measurement is taken for a project developed with the old processes and for a project developed with the new processes. This allows us to evaluate the measurements, and identify which process performs best.

For the study we used the task of creating a report for a software project. This is done for two projects, one that uses the old processes and one that uses the new processes. This means that both reports need to be equal in terms of the type of information they hold. If the amount of information differs, it would take more effort to create one report than the other. In order to circumvent this, report templates have been created, holding the information that needs to be in the report. These reports are then re-created for the study, and the time it takes to make the report is considered to be equal to the time spent on traceability.

The time it takes to create the report is an indication of the traceability. The report will only contain information about requirements and bug-reports, so no additional information with no relevance to the process traceability is in the report. Having such information in the report could influence the measurement, because for one project this information might be readily available, while for another project it might have to be collected. This would take time, and hence would influence the measurement of the traceability.

The time it takes to create reports is a good indication of the traceability because creating reports is a standard process done very often and in all company projects. When this process can be handled faster than in the old process, the new process performs better. Since the research only covers a single study it is assumed that if the introduction of traceability for the report creation process shows improvement, this will also hold for the other processes.

The metric used to validate our research question must give an indication of the time it takes to create a report and discriminate noise due to, e.g., the number of requirements and bug-reports that are traced. In order to filter out these differences, we measured the average of all traces:

\[
R = \frac{\sum_{i=1}^{n} t_i}{n}
\]

Formula (1) measures the traceability rating \( R \) for a certain development process. The rating takes the average of a number of measurements by summing these measurements up and dividing each by \( n \), where \( n \) is the number of traces being measured, and \( t_i \) is the time (in seconds) it takes to trace the requirement or bug-report and include it into the produced report. In \( R \) we assume trace durations are normally distributed.

The rating \( R \) for the traceability of a process indicates the performance of that process regarding traceability. The higher the rating is, the less efficient the process performs with respect to traceability. Even though there is no scale indicating what value means that traceability is good or bad, this formula allows us to compare different processes with respect to their traceability.

The selected “old” project developed the Barcode Library system. This is a DLL containing functionality to create, print, scan and recognize bar codes. This library has been used by multiple other projects and has been developed using the old practice. During its execution, this project made use of the task registration system, hence providing us with all the data we needed.

This project was implemented by two developers, who also maintain the source code, which is reused as part of various later systems. This means that these developers determine most new requirements for the Barcode system, which are always discussed internally. The original project plan contains 27 functional and non-functional requirements, used later on to develop new versions of the Barcode Library system.

The selected “new” project (executed as part of this study by using the traceability paths) developed the SPOT System, which is a Web-based system for e-learning and for assigning employees to projects. The system allows the administrative staff to identify and allocate the employees that are qualified for a certain man-power request. Employees can use the system to sign up for on-line education events. This project has been setup using the new process model of Fig. 7. Accordingly, requirements, bug-reports and developer tasks are made explicit. To inform the customer of the project progress, NUBIKO issued progress reports weekly. To communicate with the developers, the task registration system has been used. Accordingly, the developers access only their tasks, and can place comments and questions that the managers at NUBIKO then answer. This way, all discussions and rationale are kept in a central place. Discussions often contain documents such as workflow diagrams and user interface sketches. This project contains 37 requirements.

These two pilot projects present a good average project size in terms of project duration and number of requirements. The project duration for most projects varies between one and six months, while the average number of requirements for a project is about 30–40. These projects respectively took two and three months and have 27 and 37 requirements.

Of course this is not sufficient to state that the pilot projects are also comparable. To determine if the projects are indeed comparable, the complexity of the projects has been researched. There are different metrics for determining the complexity of software projects. In this work we used Use Case Points, a technique for determining the size of the project, in order to estimate projects cost.

Use case modeling is an accepted and widespread technique to capture the business processes and requirements of a software application project. Since use cases provide the functional scope of the project, analyzing their contents provides valuable insight into the effort and size needed to design and implement a project. In general, projects with large, complicated use cases take more effort to design and implement than small projects with less complicated use cases (Clemmons, 2006).

Table 1 shows the results. When the UUCP factors are calculated in the standard way, the final UCP value is an estimate of the number of hours needed to implement the project. The values are close, the SPOT project being only 7.3 points higher than the Barcode Library project. This relates to roughly 13% difference, so it is safe to state that both projects have a similar size.

<table>
<thead>
<tr>
<th>Factor</th>
<th>Barcode library values</th>
<th>SPOT values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unadjusted use case points (UUCP)</td>
<td>8.15</td>
<td>7.79</td>
</tr>
<tr>
<td>Technical complexity factor (TCF)</td>
<td>0.92</td>
<td>0.96</td>
</tr>
<tr>
<td>Environmental complexity factor (ECF)</td>
<td>0.38</td>
<td>0.43</td>
</tr>
<tr>
<td>Productivity factor (PF)</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>Use case points (UCP)</td>
<td>57.0</td>
<td>64.3</td>
</tr>
</tbody>
</table>
3.7. Study execution

The study has been executed in NUBIKO by two persons, a manager (person A) and a developer (person B). The traceability times for both are indicated by $t_a$ and $t_b$, respectively. All measurements have been performed without influence by the researchers in any form. Both persons were experienced in using the task registration tool and the tools creating the reports. In this way, the study is not influenced by noise like tool learning factors. The times are recorded per requirement/bug-report. Where the time could not be determined for a single requirement or bug-report but only a set, each of the requirements and bug-reports in the set were given an equal part of the total time for the set.

Both persons were asked to create two reports, one per project. Each report had to include all the requirements (listed in Meijers, 2008). This reduces the chance of either person looking up more information that the other, giving a better average over the two ratings per project. Meijers (2008) also discusses all details about the collected raw data, including the individual measurements $t_i$ for the traces to the Barcode Library system reports and the SPOT system reports.

3.8. Results

With the values for $t_i$ measured, the other variables of the rating formula $R$ can be determined to get the rating for both processes, i.e., the performance of both processes regarding traceability. The results are shown in Table 2, where $n$ is the number of measured traces, i.e., $n_{BARCODE} = 27$ and $n_{SPOT} = 37$.

Further, $T_{BARCODE}$ and $T_{SPOT}$ are the total times (in seconds) for the measured traces for the two pilot projects and $R_{BARCODE}$ and $R_{SPOT}$ their ratings. As shown, the new process has much better results yielding a much lower rating than that of the old process. In other words, to perform the same task the “new” process takes in average 36.25% less time (16.75 s) than the “old” process (46.2 s).

It can be argued that this performance improvement will be roughly the same for the other processes, because the increase in speed comes from the fact that requirements, bug-reports and developer tasks can now be easily distinguished and traced. Since this information is important in all other tasks as well, the resulting improvement will also be effective for these tasks.

3.9. Evaluation of results

The introduction of traceability for the core business processes at NUBIKO was relatively straightforward. The tools that were already in place could be used for the new processes, too. All of the tools provided functionality which was sufficient to support the introduced traceability. Because no new tools had to be purchased, the cost of introducing traceability was close to zero. The allocation of manpower was also very limited, as all existing tools could be re-used and configured to support the new processes.

The results from the previous Section show that the new processes provide better traceability support than the old processes. The main reason for this improvement is the increase in the number of entities that can be traced. By having more entities that are explicitly and easily traceable, the process makes it easier to get the information that is needed. The results also show that the total time it took to create a report was cut in half with the new processes. When the rating is considered, which is in fact the average time it takes per traceability link, the time has been 36.25% of the original time. The study only discusses the report generation process, but we corroborate that by providing traceability for the other processes too, all processes will show a significant performance increase with respect to tracing information.

The project setup with the new processes, SPOT, has been a big success for NUBIKO. Not only did the company manage to deliver the software on time, but the software contained 18 extra features and remained within budget as well. NUBIKO also reported to their customer on a weekly basis, which proved to be easier to accomplish. During the last two weeks of the project the project manager had his vacation planned and had to delegate his tasks to another manager. Even though all of the design stages had already been traversed, which was most of the work, the replacement manager had to report the implementation progress to the customer weekly. This proved to be so easy that it only took about half an hour to an hour per week to generate the reports. These achievements cannot be measured with metrics, but are worth a lot to NUBIKO as a company.

### Table 2

<table>
<thead>
<tr>
<th></th>
<th>$T_{BARCODE}$</th>
<th>$T_{SPOT}$</th>
<th>$R_{BARCODE} = \frac{\sum_{i=1}^{27} t_i}{27}$</th>
<th>$R_{SPOT} = \frac{\sum_{i=1}^{37} t_i}{37}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Person A</td>
<td>1313.0</td>
<td>623.0</td>
<td>48.3</td>
<td>16.80</td>
</tr>
<tr>
<td>Person B</td>
<td>1190.0</td>
<td>618.0</td>
<td>44.1</td>
<td>16.70</td>
</tr>
<tr>
<td>Average</td>
<td>1251.5</td>
<td>620.5</td>
<td>46.2</td>
<td>16.75</td>
</tr>
</tbody>
</table>

4. Discussion

This section presents a discussion in the context of related work. Section 4.1 addresses traceability in general, while Sections 4.2 and 4.3 discuss traceability support in SPL engineering and software process management, respectively.

4.1. Traceability

The notion of traceability is not new. Fisher (1966) used tables to cross-reference design artifacts and descriptive system documentation like what we nowadays call requirements. It has been further thoroughly discussed in the requirements engineering field, as the information about the links between requirements, the sources of requirements and the system design (Palmer, 1997; Kotonya and Sommerville, 1998). Traceability is then centered around requirements. Palmer (1997) and Gotel and Finkelstein (1994) classify traceability to support engineering to and from requirements. Its importance has been recognized in the software specification and verification community too, where traceability has been viewed as “relating the abstract values of the specification to the concrete values of the implementation” (Dick and Fairve, 1993). Some work has been done on traceability at the software architecture level. Berry et al. (2003) and Nuseibe (2001) propose ways to bridge the gap between requirements and software architectures. Grünbacher et al. (2004) propose ways to trace requirements to software architecture models, and in Medvidovic et al. (2003) they cover traceability from software architecture to implementation, too. Traceability in emerging forms of software engineering is analyzed in Spanoudakis et al. (2002).

Like Palmer (1997) and Kotonya and Sommerville (1998) we address the need for traceability between different software development life cycle steps. We further embrace the idea discussed by Medvidovic et al. (2003) of tracing different models on the same abstraction level (e.g., tracing product feature models to product structural models).

4.2. Traceability in SPL engineering

Jirapanthong and Zisman (2005) advocate the use of traceability relations to support SPL engineering. Relations between product members and the product line architecture, and among the prod-
uct members themselves must be automated. For this purpose, a rule-based approach for generating traceability relations among different types of artifacts is presented. Ten different types of traceability relations (between the various documents) are identified, among feature, sub-system, process, module, use case, class diagram, statechart diagram and sequence diagram models. An extended version of XQuery is used to represent the rules and to support extra functions to cover some of the traceability relations being proposed. Differently from Jirapanthong and Zisman (2005), the traceability framework we propose covers both activities, roles, and artifacts, and not just artifacts. Moreover, we focus on core traceability paths, hence limiting the amount of traces to be created and maintained.

Schmid et al. (2006) focus on tool support for SPL engineering and discuss how a seamless integration of product line concepts in existing tools is possible and desirable. When analyzing the requirements a tool extension for product lines must support, traceability support is considered as mandatory. Thus, the authors introduce in REMAP (the extension to DOORS for requirements modeling in SPL engineering) an explicit and automatically generated trace between a requirement in the product line and a requirement in the product: if a requirement is added by instantiation to a product, REMAP creates a link from the requirement in the product to the respective requirement of the product line infrastructure.

By restricting the focus on traceability among feature and structural models (representing the core artifacts in the problem and solution spaces in SPL), many of the existing approaches for specifying SPL (e.g., Matinlasi, 2004) address in various ways (typically, through implicit traceability links) the relation between problem and solution space.

For example, Lee et al. (2000) present an extension of FORM (feature-oriented reuse method) that helps to systematically derive structural elements from features. The extension makes use of feature classification as the conceptual framework on which feature types and object categories are mapped, so that product derivation is enabled. This classification is based on the characteristics of the domain and the SPL elements, while in our case traceability links are based on the type of use. Gomaa (2004) captures the dependencies between features and classes, by covering how features addressing functional requirements relate to the software classes realizing the product line’s functionality. A feature-based impact analysis is employed to determine feature/class dependencies. Bayer et al. (1999) represent the same dependencies by means of a decision model describing the SPL level as open decisions and possible resolutions. This open decision model is then used during application engineering to instantiate the SPL into a product that implements its requirements.

While previously analyzed SPL engineering approaches mainly focus on the problem space (feature model), to our knowledge only Ménage, Koala and KobrA offer explicit means to manage the solution space – the structural model – as well. Since we focus on the explicit representation of the links between feature and structural models, we report on Ménage, Koala and KobrA below.

Ménage (Hoek, 2005) is an approach for managing evolving product line architectures. It provides a definition of how to represent product line architectures (addressing variability and also evolution) and a graphical environment which facilitates the specification, in the defined representation, of SPL architectures. Ménage supports the specification of three kinds of variation points: optional elements, variant types, and optional variant elements. The SPL architecture produced with this environment thus includes not only the SPL architectural core elements but also optional and variant elements. In order to derive a product architecture out of the SPL architecture, Ménage includes the “Selector” tool which resolves the variation points.

Koala (Ommering, 2004), instead, is an ADL specially designed for modeling embedded software for consumer electronics. It inherits from Darwin (Magee et al., 1995) the main concepts and ideals, even though it is more oriented to notations and concepts commonly used in consumer electronics products. Koala allows to specify hierarchical architectures, it makes a distinction between component types and instances, it allows to construct configurations by instantiating components and connectors and explicitly models optional interfaces.

KobrA (Atkinson et al., 2002) is a software process- and product-centric engineering approach. KobrA (Component-based Application Development) is product-line centric since it implements a product-line philosophy: it identifies commonalities and variability in a specific application domain and creates a reusable software infrastructure (or framework) which may be instantiated to identify the specific products. A KobrA framework is a collection of “components” (i.e., software modules which provide/require services to/from other components) organized in a hierarchical tree structure. Each model representing a KobrA variant component contains a “decision” which allows to resolve the component variability. A “Decision model” collects the different decisions related to each component. During the “Application Engineering” phase, products are identified by instantiating the product line framework, making use of the Decision model.

In their respective ways, Ménage, Koala and KobrA each support the arrow $b_1$ in Fig. 2. While Ménage and KobrA explicitly keep the traceability links realizing $b_1$, Koala supports $b_1$ without treating product line and product level in a different way. Ménage is the only approach which provides notational support for generic models. In KobrA generic models can be modeled but not enforced through an explicit and separate representation.

Traceability paths define a conceptual framework able to support the application of an existing approach like, e.g., the ones described above. In this respect, open research questions are: “Which stakeholders are going to apply the approach? What are their concerns? How can these stakeholders use the approach to address their concerns?” To answer these questions we can adopt a scenario- or usecase-driven approach similar to Kruchten et al. (2005). By interviewing relevant stakeholders we can elicit the industrial usage requirements toward SPL engineering, and translate them in terms of scenarios and the associated traceability requirements (i.e., usages and their supporting traceability paths). Furthermore, once the relevant usages are clear, they can be used to drive the development of tool support: usages correspond to potential services of an SPL development environment; traceability paths identify the conceptual model to be managed. For example, earlier we mentioned that Ménage includes the “Selector” tool which resolves variation points during product derivation: selection of the variants realizing variation points is indeed one of the possible relevant usages. In a similar way, Ménage could be extended with additional tools supplying additional usages.

Traceability paths accommodate the explicit representation of the traceability links between concrete models and generic models. Generic models capture properties that should hold for, e.g., the whole SPL; traceability paths linking the generic model with the SPL ensure that consistency is properly checked. According to Finkelstein et al. (1994), inconsistency can be defined as any situation in which a set of descriptions does not obey some relationship that should hold between them. In SPL engineering, inconsistencies may be potentially located on each of the traceability paths presented in this paper.

In general, inconsistencies are more likely to occur between any of the models depicted in Fig. 2 if the links between them are left implicit. For example, in the CW SPL there is no explicit impediment to declare the chat component as mandatory, while the feature char is optional in the SPL feature model. This of course
introduces an inconsistency that is difficult to identify. This type of inconsistency is due to the inconsistent handling of variability in the two models. A further example is if the WriteDoc feature (mandatory at the SPL level) does not appear at all at the product level feature model. The documentation of traceability paths is needed in order to provide a formalized basis for consistency checking.

Lastly, traceability paths can have a positive impact on the maintainability and evolvability of SPL artifacts. Whenever SPL artifacts change, the “architectural drift” problem described by Perry and Wolf (1992) may arise, i.e., to meet tight deadlines only the low-level design and implementation are changed, while the architecture is not changed accordingly. Traceability paths may limit such a problem, by capturing the links important for evolution.

4.3. Traceability in software process management

Asuncion et al. (2007) propose a software traceability tool for a comprehensive process-oriented traceability support during the entire software development life cycle, by focusing on both requirements traceability and process traceability. The authors of this paper highlight the need of process traceability and point out that “the identification of a few types of global artifacts mitigates the complexity associated with traceability”, which is perfectly in line with the approach proposed in this paper.

Jarke et al. (2004) propose a product-oriented and process-oriented traceability process, claiming that traceability is essential for improving the reuse of both product and process experiences. The paper then focuses on linking goals, scenarios and models. This paper shares with us the claim that both product and process traceability shall be managed.

Maeder et al. (2007) stress the importance of traceability links during the whole software development process and the need of integrating such links into development methods. Then, they analyze and classify the Unified Process (UP) artifacts to establish a traceability link model for this process. Similarly to us, they observe that explicit traceability management requires a clear and precise definition of activities, roles and artifacts. As a result of their study, a complex traceability link model for the UP workflow requirements, analysis and design is presented. The main difference with our approach is in the types of traceability paths to be kept and on the amount of links to be traced.

5. Conclusions

In this paper we have presented an approach to customize traceability to the situation at hand. Instead of automating tracing, or representing all possible traces, we scope the traces to be maintained to the activities stakeholders must carry out. We define core traceability paths, consisting of essential traceability links required to support the activities.

We concretely illustrate the applicability of our approach in two very different areas, SPL engineering and software process management. In both cases, we described the traceability issues we encountered and the traceability paths solving them.

In SPL engineering we introduced three traceability issues: (i) traceability between feature models and structural models, (ii) traceability between product line models and product level models, (iii) traceability between generic and concrete models. For all of them we identified the needed traceability paths: their specialization into sets of specific traceability links depends on the choices of the using companies.

In software process modeling we introduced two traceability issues: (i) traceability between software artifacts and tasks, and (ii) traceability between standard activities and project-specific processes elements. The identified traceability paths, again, depend on the specific processes and activities in the company.

We argue that these traceability paths define a necessary set of traceability links that we call “core” set. We need this “core set” to deal with some major issues that arise during software engineering. We still need to investigate if this “core set” is also sufficient in general. To this end we need more examples, and studies with industrial cases that use traceability information, to see if these uses can be mapped on our core set.

Our work differs from previous efforts on traceability in that we recognize and explore the need for a minimal set of links to make traceability manageable. In this way, traceability becomes manageable and focused to the most relevant concerns. By knowing in advance the use we aim at, we can consciously decide on which traceability links to document, and can exclude those that are not necessary. We argue that traceability paths also help in better focusing on which traces are really important for the business purposes of the company, and which instead can be neglected.

Acknowledgements

The authors would like to thank Alexander Egyed and M. Ali Babar for their constructive comments and suggestions on various versions of this article; Jesse Meijers who carried out the industrial study as part of his Master thesis, and NUBIKO who hosted the industrial study and provided the necessary resources.

References


Patricia Lago is Associate Professor in Software Engineering at the VU University Amsterdam, The Netherlands. Her research interests focus on software and service architectures, software architecture knowledge management and modeling. She co-authored various refereed publications in international journals, books and conference proceedings. She is co-organizer of the International Workshop on Sharing and Reusing architectural Knowledge (SHARK). She also acts as PC member and reviewer in several international conferences and journals in the field. A full publication list is available at www.cs.vu.nl/~patricia.

Henry Muccini is Assistant Professor at the University of L’Aquila, Italy. Henry’s research interests are in the Software Engineering field, and specifically on software architecture modelling and analysis, component-based systems, model-based analysis and testing, fault tolerance, and global software engineering. He has published various conference and journal articles on these topics, co-edited two books on “Software Engineering of Fault Tolerant Systems” and “Architecting Fault Tolerant Systems V”, and co-organized various workshops on related topics. He serves as PC member and reviewer in many international conferences and journals. More detailed information may be found at http://www.HenryMuccini.com.

Hans van Vliet is Professor in Software Engineering at the VU University Amsterdam, The Netherlands, since 1986. He got his PhD from the University of Amsterdam. His research interests include software architecture and empirical software engineering. Before joining the VU University Amsterdam, he worked as a researcher at the Centrum voor Wiskunde en Informatica (Amsterdam). He spent a year as a visiting researcher at the IBM Almaden Research Center in San Jose, California. He co-authored over 100 refereed articles. He is the author of “Software Engineering: Principles and Practice”, published by Wiley (3rd Edition, 2008). He is a member of IFIP Working Group 2.10 on software architecture, and the Editor in Chief of the Journal of Systems and Software. http://wwwp.dmsalas.org/wiki/User:Hans.