Traceability for Model Driven, Software Product Line Engineering

Nicolas Anquetil[1], Birgit Grammel[2], Isménia Galvão[3], Joost Noppen[1,3], Safoora Shakil Khan[4], Hugo Arboleda[1], Awais Rashid[4], Alessandro Garcia[4]

[1] Ecole des Mines de Nantes, France
{nicolas.anquetil,johannes.noppen,hugo.arboleda}@emn.fr

[2] SAP Research CEC Dresden
birgit.grammel@sap.com

[3] University of Twente
i.galvao@ewi.utwente.nl

[4] Lancaster University, UK
{shakilkh,garciaa,marash}@comp.lancs.ac.uk

Abstract. Traceability is an important challenge for software organizations. This is true for traditional software development and even more so in new approaches that introduce more variety of artefacts such as Model Driven development or Software Product Lines. In this paper we look at some aspect of the interaction of Traceability, Model Driven development and Software Product Line.

Keywords: Product line, traceability, object-oriented, aspect-oriented.

1 Introduction

Traceability of artefacts elicits the means of understanding the complexity of logical relations and dependencies existing among artefacts that are generated during the software development lifecycle. Numerous kinds of artefacts are generated at the individual development stages, ranging from requirement artefacts to design elements down to source code fragments. With the inception of model-driven software development the scope of artefacts has been diversified by introducing models concerning, business processes, system requirements, architecture, design, tests, etc.

Since software development is ever facing the challenge to minimise development costs, advancing fields of Software Product Line (SPL) engineering and generative programming have been fostered. This in turn raises the need for more intricate traceability solutions, which in addition to classical end-to-end traceability, have to support for the traceability of variabilities and commonalities in the SPL. One of the main objectives of the European project AMPLE[1] is to bind the variation points in

1 http://ample.holos.pt/

various development stages and dimensions into a coherent variability framework across the SPL engineering life cycle thus providing forward and backward traceability of variations and their impact.

In this paper we present various perspectives of the AMPLE project on traceability for Model Driven, SPL engineering. The remainder of this paper is organized as follows. Section 2, introduces basic concepts of SPL engineering and contextualize traceability in it. Section 3, proposes a categorization of traceability links for SPL. Section 4, discusses how to deal with uncertainty and tracing the rational of decisions during the SPL development process. Section 5, looks at fine grained traceability links when mixing Model Driven development and SPL. Finally, Section 6 presents our conclusions and future work.

2 Software Product Line

The software industry is in crisis. It is unable to produce software at the pace required by the market: Projects are delayed, they fail to meet quality requirements, their budget is exceeded, expected functionalities are not delivered. SPL comes as an answer to this situation. It promises to deliver software faster, with higher quality and at a lower cost [1]. In this section we will introduce the basic concepts of SPL and how SPL and traceability interact.

2.1 Basic Concepts

The key to SPL promises (faster, better, cheaper) is to target, not a single system, but a family of similar systems all tailored to fit the wishes of a particular market from a constrained set of possible requirements. SPL is about producing software for a well defined market, from a base software architecture, with a predefined set of options, called variation points. To achieve higher quality more rapidly, it is based on reuse: of the software architecture and of the software components that may be plugged into it. Although the initial architecture and software components may be costly to develop, successive applications inside the family are cheaper and cheaper as they reuse most of what was build for the previous applications [1]. The SPL paradigm uses two processes and two main focuses (see Figure 1): (i) *Problem Space* focuses on defining what problem the family of applications, or an individual application in the family, will address; (ii) *Solution Space* focuses on producing the software components to solve that problem; (iii) *Domain Engineering* is responsible for establishing the software family platform, first by identifying typical requirements of the problem space and where they will be authorized to vary, second by developing the software architecture that address these requirements and the software components that fit into the architecture, and; (iv) *Application Engineering* is responsible for deriving individual applications from the requirements of a customer (inside the set of authorized variations) and composing this application from the family architecture and the available components.

Although the two were conceived separately, Model Driven Development is a natural candidate to fit in the general framework of SPL: One may develop a meta-model that can be transformed in different applications according to the wishes of the customer [2,3]. The general solution is described in Figure 1. First, in domain engineering, one defines a meta-model of the problem (top left), specifying the concepts that may be used in the creation of a solution --an application--, and a feature model (model of all the features available). Second, still in domain engineering, but in the modelling of the solution (top right), one defines transformation rules to generate code from the application model (when it will be available). Obviously the future application model must conform to the meta-model of the product line. In addition, manually written software components (source code) can also be created that are intended to be combined with the automatically generated code later. Third, in application engineering (bottom left), one defines a model of a particular application (conforming to the meta-model defined in domain engineering). One also selects the features that should be implemented in this particular application. Finally, in the solution space (bottom right), the application is automatically derived from the model by applying the transformation rules.

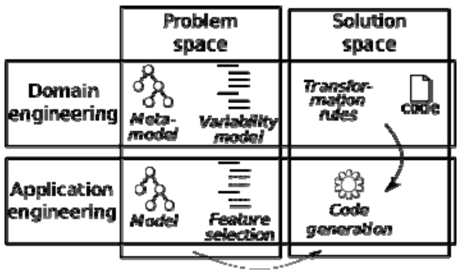

Fig. 1. How MDD fits in the two processes and two spaces of SPL.

2.2 Traceability for SPL

Traceability is recognized by all to be highly important for SPL engineering. On top of traditional concerns for traceability, SPL has to deal with variability and with two development processes. Variability is the description of all possible variation points in the family products, and all the variants, the available options for each variation point. Traceability appears as a key asset to manage this complexity. Variability is seen as the one fundamental aspect of SPL, and specific to it, that needs to be traced.

The difficulties linked to traceability in SPL are [4]: (i) there is a large number and heterogeneity of documents, even more than in traditional software development; (ii) one needs to have a basic understanding of the variability consequences during the different development phases; (iii) one needs to establish relationships between product members (of the family) and the product line architecture, or relationships between the product members themselves; and (iv) there is still poor general support for managing requirements and handling complex relations.

We could not find much tool support, neither available for industrial use nor in form of research prototypes in academia. Traceability means to link several artefacts at different levels and the rationale of this link. One has to link documents, stakeholders and the rationale behind the links. Since software development and more specifically SPL development is a complex task one has to trace many objects of various kinds with different structures. In [5] a general presentation of the traceability needs and the integration in a SPL are proposed. The traceability requirements are: (i) it should be based on the semantics of models used in the SPL infrastructure; (ii) it should be customized to capture relevant trace types; (iii) it should be capable to handle variability; (iv) a small set of traces is better; and, (v) it should be automated when possible.

Berg *et al*. [6] view software engineering for single (traditional) systems in two dimensions, one for the development process and the other for levels of abstraction. All development artefacts can be placed somewhere in these dimensions. Variability adds a third dimension that explicitly captures variability information between product line members. This approach establishes a conceptual variability model which provides the appropriate mapping between all variation points in the two dimensional space (development process and levels of abstraction).

Ajila and Ali Kaba [7] use traceability to manage the SPL evolution. They identify three sources of changes in product line: (i) changes in an individual product; (ii) changes in the entire product line; and (iii) repositioning of an architectural component from individual product to the product line. The authors also analyse more precisely the reasons and the nature of changes in SPL development. The dimensions of analysis can be: motivations that led to the change (external or internal) and changes in the management process.

In [8] Moon and Chae propose a meta-modelling approach to trace variability between requirements and architecture in SPL. They define two meta-models for requirements and architecture integrating variability. These matrices contain information computed from the software structure and the variability points. Three kinds of relationships are provided: (i) between artefact constituents and trace matrix; (ii) between artefact constituents and models, and; (iii) between artefact constituents and specifications.

We will now present several propositions to treat traceability in Model-Driven SPL. Not all of them are specific to SPL, but all of them will be applied to this context in the AMPLE project.

3 Categorization of Traceability

As pointed out by the Center of Excellence for Traceability [9], the precise semantic of traceability links is poorly understood, and there is possibly a wide range of semantics. This is aggravated by the fact that it may not be desirable (or even possible) to create a closed set of semantic kinds of links. If one casts into stone the kinds of semantic links, then one loses flexibility for user-defined links that might be necessary to meet different project or company needs. But not predefining the link semantics, would greatly complicate automatic and elaborate treatment. We chose to develop a two layered solution with a high level abstract categorization, that we hope is general enough to fit all purposes; and a lower level, more detailed categorization, that may be too specific in some specific situations. This is still a research issue and we have no definitive answer yet.

3.1 Dimension of Traceability

Traditional (non SPL) software engineering (e.g. [10, p.59], [11, p.526], [12]) defines two forms of traceability: vertical and horizontal. Unfortunately, different authors swap the respective definitions of vertical and horizontal! In this paper, we will call them intra and inter traceability. *Inter* traceability refers to relationships between different levels of abstraction: from requirements to models to implementation. *Intra* traceability refers to relationships between artefacts at the same level of abstraction: between related requirements, between models, between software components, etc. To this initial framework, SPL engineering introduces a third dimension, orthogonal to the two other ones to deal with *variability* and its implications (See also [6]). Traceability links are required to relate variation points (options) to their variants (choices), variants between themselves (when one choice constrains another one), variation points between themselves, low level artefacts to variation points or variants, and finally, choices made at the application engineering level to options offered at the domain engineering level. Finally, since dealing with configuration management is also a goal of the AMPLE project, we include a fourth dimension, *evolution*, for relationship between the various versions and revisions of a given artefact.
Note that there may be interactions between the different dimensions. For example, intra and inter traceability links may evolve between two versions of the SPL. This indicates that *intra* and *inter* traceability links may themselves be related by *evolution* traceability links. Variability traceability links are also subject to evolution over time. Finally, intra and inter traceability links may also be subject to variability traceability. For example, if two artefacts have an intra or inter traceability link in the domain model, and if both appear in the corresponding application model, then they should exhibit the same intra or inter traceability link in the corresponding application model. In summary, we may propose a hierarchy of dimensions: *Evolution* traceability may also apply to *intra*, *inter* or *variability* traceability relationships (and not only on artefacts). *Variability* traceability may also apply to *intra* or *inter* traceability relationships. All other interactions between two dimensions are considered meaningless.

3.2 Taxonomy of Traceability

There are quite a few approaches for inter and intra tracing intraditional systems [22],[23]. But these approaches do not fulfil the needs of SPL due to dependencies existing: (i) from core assets (domain engineering) to products (application engineering); (ii) between commonality and variability at different abstraction level; (iii) for core assets used by multiple products in a family of products. During the development of a SPL, numerous entities, artefacts, and models are created during both domain engineering and application engineering [16][17]. This makes it complex to maintain and evolve the large number of intricate trace dependencies.

To facilitate trace maintenance and evolution in SPL, we propose to move away from simple associative trace links to links that capture the semantics of the relationship between the traced artefacts. We define a semantics-based dependency taxonomy wherein the dependency information: captures intricate information about the traces; promotes better understanding of the trace relationships; justifies the rationale for existence of a particular trace link, and; determines the significance of a trace link and help determine its consequence or impact on tracing information during SPL evolution. The taxonomy describing various facets of a dependency is influenced from conventional requirements engineering approaches, SPL concepts, and work on dependencies by [21]. We also investigated two case studies: HealthWatcher [18][19] and MobileMedia [20]. From these studies we structured the dependency links around two characteristics: *nature* and *granularity* .The *nature* of a dependency describes the fundamental categorization of the trace formed and helps define the significance of the dependency (which may vary from domain to domain) holding at the same level of abstraction (intra), higher to lower abstraction (inter), or between core assets and product(s). The nature of dependency can be categorized as: *Goal*, *Conditional*, *Service*, *Task*, *Temporal*, and *Infrastructure*. A more detailed discussion on *nature* of dependency taxonomy is presented in [19]. The *granularity* of a dependency elaborates on the trace by providing a better insight into the fundamental categorization of the trace (nature of dependency) formed at the same level of abstraction (intra), higher to lower abstraction (inter), or from core assets to products. The *granularity* of dependency helps identify the number of entities impacted directly or/and indirectly when a requirement, design, or implementation is evolved. The granularity of dependency can be categorized as: *Refinement*, *Composition*, *Constraint*, *Multiplicity*, *Behavioural*, and *Structural*.

We now discuss a brief trace scenario from the SmartHome industrial case study to showcase the dependency taxonomy. The SmartHome application bridges different technologies in a house like central heating, security system, household appliances through mobile phones and/or personal computers to retrieve the status, set or modify the control/setting of the devices. Our example scenario describes traces amongst the artefacts in application domain. The climate control system for managing the central heating ensures that the temperature a user (owner) has specified for the house is maintained. The desired temperature is maintained by automatically turning the central heating on/off when the specified temperature is reached. The nature of dependency for the requirement forms a (service, conditional) dependency with the HeatingComponents and Thermometer components at architectural level. Service and conditional dependency is formed as the Thermometer component gets the temperature of the house and the HeatingComponent turns the central heating on/off if the temperature is above or below the specified temperature range. The granularity of dependency is (behavioral) as the HeatingComponent reacts to the data output from the Thermometer component. The example shows the dependency model help extract end-to-end trace information between the loosely and/or tightly coupled requirements and architecture providing the system analyst an understanding of how requirements are being realized at architecture level.

4 Traceability in the Presence of Uncertainty

Independent of the categories of traceability and their nature and granularity, we propose to attach additional information to traceability links: The rational for its creation and the confidence we have in this rationale.

During software development, a large number of design decisions must be resolved. Typically, for each design issue several candidate solutions are considered. The rationale behind these design decisions is frequently based on assumptions made about diverse relevant criteria related to these candidates, calculating the alternatives' overall quality, and choosing the most appropriate solution. Ideally, the information used for taking such decisions would be of perfect quality, i.e. clear and accurate. However, in practice it is very difficult to attain accurate information at the moment it is required. As important decisions are taken in early phases of development, software architects will only have a partial and abstract view of the final, complete system. As a result, the design activities generally are performed with assumptions on relevant system characteristics that only partially provide the information with the desired quality. The rationale for design decisions is naturally subject to uncertainty.

Uncertainty plays a role in any system that needs to evolve continuously to meet the specified or implicit goals of the real world [13]. But while SPL engineering is based on the principles of reuse and variability management, the development of SPLs can suffer from uncertain information. As product line architectures are used over a prolonged period of time, they become subject to unforeseen evolution and maintenance. Moreover, the requirements definition and architectural design phases typically will be prone to uncertain inputs, as the product line is intended to support a versatile product family in volatile markets with changing demands. As a result of the variety of product families, the complexity of product line architectures and the longevity over which these must be maintained and evolved, it can be argued that the impact of uncertainty on product line development can be even more sever than traditional software systems.

As seen in section 2.1, the evolution of SPL artefacts in the problem space and the solution space, both in domain engineering and application engineering levels, can profit from model-driven techniques. The flexibility of model transformations offers ample means to address evolution of product lines. For example, model-driven approaches can automate the generation of trace links between source and target artefacts involved in a transformation [14]. Nonetheless, the application of MDE approaches does not resolve all problems caused by evolution and uncertainty in SPL development. MDE artefacts are subject to evolution. Change requests may cause the evolution of metamodels, models and model transformations. Moreover, the definition and realization of a model-driven approach can suffer significantly when uncertainty in the available information is not recognized and addressed accordingly.

Under this perspective, traceability of design decisions in SPL development is an important and relevant issue, as these are key points where uncertainty influences the design process. For performing traceability in the presence of uncertainty, the focus of handling uncertain information in particular should be on the rationale used to resolve design decisions. By identifying the uncertainty that exists in design decision rationale and modelling it accordingly in the decision process, its negative influence can be minimized. Further, tracing information on design decisions facilitates the understanding of the impact of the uncertainty on the development of the SPL. Tracing the rationale of decisions improves the understanding of the important contextual factors that impact the quality of the SPL and variability management.

To this end, we have defined a meta-model that conceptualizes the kinds of design decision rationale in which we are interested, such as problem, alternatives, quality attributes, context and arguments. This meta-model comprises elements from argument-based rationale methods, problem-solving approaches and quality evaluation methods. Moreover, the meta-model accommodates the representation of uncertainty in the assumptions made by the developers while taking design decisions. Uncertainty is represented by utilizing techniques from fuzzy set theory.

The rationale behind each relevant design decision can be a model instantiated from the design decision rationale meta-model. Such models are themselves also considered as traceable artefacts. Therefore, the traces related to or from design decision rationale instances are stored along with inter or intra traceability relationships. For example, the design decision rationale can be traced to other decisions, or from and to other artefacts, such as requirements and architectural models. In this way, we are able to analyse the influences of uncertainty in the design rationale, while performing traceability for the sake of, for example, change impact analysis and root-cause analysis.

5 Traceability and Fine Grained Variability

We saw in Section 2.1 how, in the Model Driven, SPL approaches [2,3] a particular application is defined as a model conforming to the meta-model of the product line. The application must also choose available features from a feature model. By default, this approach does not allow fine grained selection of features, an application either has or not a feature. We call *large variation* a characteristic that affects the whole application [15]. For instance, properties such as localization (English or German), or, in a Smart Home system, a large variation could express that the house can have automatic lights (this would imply that *all* the lights in the houseare managed automatically). In contrast to this, we also define the concept of *fine variation* [15]. A fine variation is a characteristic that may be applied to specific elements of the application model. For example, in a Smart Home system, a fine grained variation could express that specific rooms of the house have the feature automatic light, but not all of them. Note that, large variations can be treated as special case of fine variation where all the elements of the model individually have the feature of interest.

The gain in flexibility of fine variation comes at the cost of more complex models and meta-model, with many new artefacts, model-to-model transformations, etc. Maintaining and evolving all the relations between individual elements and their features would require detailed management of traceability at a fine grained level. To manage this additional complexity, we defined a constraint models as part of the problem space modelling, during domain engineering [15]. This constraint model expresses what features may be linked (with fine variation) to what element of the meta-model. The constraint model also restrict the possible bindings by bounding possible cardinality and specifying properties that the element should have when bound to the feature (for example, room that have automatic light requires some sensor). The actual binding of an element of the application to a feature occurs at the application engineering level, during the modelling of the problem. This is the moment where possible bindings described in the constraint model are created (or not) between actual elements of the application (concept in the model of the application) and the features offered in the feature model. Each binding defined in the application model is automatically checked against the restrictions expressed in the constraint model. The transformation of the application model to implementing code is realized by transformation rules.

Fine grained variability works in three dimensions of traceability: The specification of binding constraints between meta-concepts and features is an intra traceability, as both are at the same level of abstraction (during domain engineering, as part of the problem space modelling); binding of a concept to a feature is a variability traceability as the concepts appear during application engineering whereas the features are specified at the domain engineering level. Finally, the implementation of a given binding between a concept and a feature is an inter traceability.

6 Conclusion and Future Work

There is no doubt that traceability is a fundamental discipline of modern software development. As new development approaches emerge, such as Software Product Lines (SPL) engineering, the challenges of traceability, still not complete tackled, are increased. For example, SPL engineering increases the range of artefacts (variability model, variation points, variants). Model Driven Development (MDD) is another approach that also introduces new artefacts (meta-models, transformation rules).

In this paper, we looked at the AMPLE project, which is interested in the interaction of MDD and SPL with respect to traceability. We proposed a categorization mechanism for traceability links that offers to level of semantic: at the higher level we have four general traceability dimensions; at the lower level we propose finer grained semantic categories that may be specific to Model Driven, SPL engineering. We also discuss the problem of tracing development decision in the presence of uncertainty. Finally, we proposed a fine grained traceability mechanism between a domain meta-model and a product line variability model.

AMPLE is a project in progress and we started to implement these ideas in a traceability framework (described in another paper presented at this workshop). Other actions include creating industry case study to test our tools.

Acknowledgements. This work is supported by the European Commission grant IST-33710 - Aspect-Oriented, Model-Driven Product Line Engineering (AMPLE),

References

1. K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering - Foundations, Principles, and Techniques*. Springer Verlag, Heidelberg, 2005.

2. K. Czarnecki, M. Antkiewicz. “Mapping Features to Models: A Template Approach Based on Superimposed Variants”. *GPCE'05.*, Lecture Notes in Computer Science, Vol. 3676, pages 422-437. Springer Verlag. 2005.
3. M. Voelter and I. Groher. “Product Line Implementation using Aspect-Oriented and Model-Driven Software Development”. *SPLC'07*, pages 233-242
4. W. Jirapanthong and A. Zisman. “Supporting Product Line Development through Traceability”. *APSEC’05*, pages 506--514. .
5. J. Bayer and T. Widen. “Introducing traceability to product lines”, *Fourth International Workshop on Product Family Engineering (PFE-4)*, pages 399-406, 2001.
6. K. Berg, J. Bishop, and D. Muthig. “Tracing software product line variability - from problem to solution space”, *SAICSIT '05*pages 182-191
7. S. Ajila and B. Ali Kaba. “Using Traceability Mechanisms to Support Software Product Line Evolution”. *IRI’04*, pages 157-162.
8. M. Moon and H. S. Chae, “A Metamodel Approach to Architecture Variability in a Product Line”, *, 9th International Conference on Software Reuse*, Lecture Notes in Computer Science vol. 4039, pages 115-126, Springer Verlag, 2006.
9. G. Antoniol et. al.: “Grand Challenges in Traceability”. *Technical Report COET-GCT-06-01-0.9*, Center of Excellence for Traceability, September 2006.
10. J. O. Grady. *System Requirements Analysis. Academic*. ISBN: 978-0120885145, Academic Press, Inc., Orlando, FL, USA, 2006.
11. S. L. Pfleeger. *Software Engineering: Theory and Practice*. ISBN: 0131469134 Prentice-Hall, Inc., 3rd edition, 2005.
12. S. Ambler, “What's up with agile requirements traceability?”, Dr.Dobb's portal, http://www.ddj.com/architect/184415807, Oct. 18, 2005 (consulted Apr. 18, 2008).
13. M. M. Lehman and J. F. Ramil. “Software Uncertainty”. *Soft-Ware 2002: Computing in an Imperfect World*, Lecture Notes in Computer Science vol. 2311, pages 477–514. Springer Verlag, 2002.
14. I. Galvão and A. Goknil, “Survey of Traceability Approaches in Model-Driven Engineering”. *EDOC’07*, pages 313-326,.
15. H. Arboleda, R. Casallas, and J.-C. Royer. “Dealing with Constraints during a Feature Configuration Process in a Model-Driven Software Product Line”. *DSM’07*, pages 178–183.
16. S. D. Kim, S. H. Chang and H. J. La. “Traceability Map: Foundations to Automate for Product Line Engineering”. SERA 2005, pages 340-347
17. W. Jirapanthong and A. Zisman. *XTraQue: traceability for product line systems.* Journal: Software and Systems Modeling. Springer Berlin/Heidelberg, 2007
18. P. Greenwood et. al. “On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study”. ECOOP’07, pages 176-200
19. S. S. Khan, et. al.: “On the Interplay of Requirements Dependencies and Architecture”. CAiSE’08
20. E. Figueiredo, et. al.: “Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability”, ICSE’08
21. B. Ramesh and M. Jarke. “Towards Reference Models for Requirements Traceability”. IEEE Transactions on Software Engineering. 27(1), Jan 2001.
22. S. Ajila. *Software Maintenance: An Approach to Impact Analysis of Object Change*, Software Practice and Experience, Vol. 25, No., pp. 1155-1181, 1995.
23. S. Ibrahim et al.: *A Requirements Traceability to Support Change Impact Analysis*. Asian Journal of Information Tech. 2005, Vol. 4, No. 4, pages 345-355