Asset Recovery and Incorporation into Product Lines

Jens Knodel[1], Isabel John[1], Dharmalingam Ganesan[1], Martin Pinzger[2], Fernando Usero[3], Jose L. Arciniegas[4], Claudio Riva[5]

[1]*Fraunhofer Institute for Experimental Software Engineering (IESE), Kaiserslautern, Germany {knodel, john, ganesan}@iese.fraunhofer.de*

[2]*Institute for Informatics, University of Zurich, Zurich, Switzerland pinzger@ifi.unizh.ch*

[3]*Abengoa, Telvent, Seville, Spain fernando.usero@telvent.abengoa.com*

[4]*Dep. de Ingeniería de Sistemas Telemáticos, Universidad Politécnica de Madrid, Madrid, Spain jlarci@dit.upm.es*

[5]*Software Architecture Group, Nokia Research Center, Helsinki, Finland claudio.riva@nokia.at*

Abstract

Software product lines aim in having a common platform from which several similar products can be derived. The elements of the platform are called assets and they are managed in an asset base being part of the product line infrastructure. The products are then built on top of the assets. Assets can include own developments, open source or third-party software modules, as well as design and project documents. In the context of the European-wide project FAMILIES we concentrated on techniques used to build the platform with focus on the recovery of these assets from existing systems. We present an approach on how to incorporate existing assets into the product line infrastructure. Thereby we explicitly distinguish the asset origins and the different information sources available. The incorporation is a quality-driven process that is backed up by a set of reverse engineering techniques to evaluate the asset's internal quality. The quality assessment of an asset is the critical measurement for industrial development organizations in order to incorporate assets into their product line infrastructure.

Keywords: architecture, asset, asset incorporation, asset recovery, product line, product line architecture, product line infrastructure, reverse engineering.

1. Introduction

Software product lines are rarely created right away on the green field but they emerge when a domain becomes mature enough to sustain their long-term investments. The typical pattern is to start with a small set of products to quickly enter a new market. As soon as the business proves to be successful new investments are directed to consolidating the software assets. The various products are migrated towards a flexible platform where the assets are shared and new products can be derived from.

Thereby software product lines aim at sharing more than just the development effort (i.e., source code, components, design concepts, requirements, and test cases), they improve the quality, reduce time-to-market, and increase the number of derived products. Typically, product lines are built on top of existing, related software systems whereas the common artifacts among these systems are integrated into a common asset base managed in the product line infrastructure.

In order to keep the quality of the asset base high, the product line architects have to decide whether or not an existing asset becomes part of the asset base, in particular to identify the needs for adaptation of the asset to make it suitable for the whole product line. The derived decision either is reuse as-is, or reuse and adapt, or reconstruct, or refactoring or (re-) implement. Since high value assets can com from different origins (i.e., legacy, in-house, 3rd party, open source), the incorporation of these assets differs slightly because the asset origin dictates the available different information to assess the asset's internal quality.

In this work, we present a generic asset incorporation process to integrate existing assets into the asset base, and relate a set of reverse engineering techniques to the recovery of assets of different origins. The reverse engineering techniques are classified based on the information source available accompanying an asset including static and dynamic, document, and historic analysis techniques as well as architecture evaluation and conformance checking techniques. They aim at

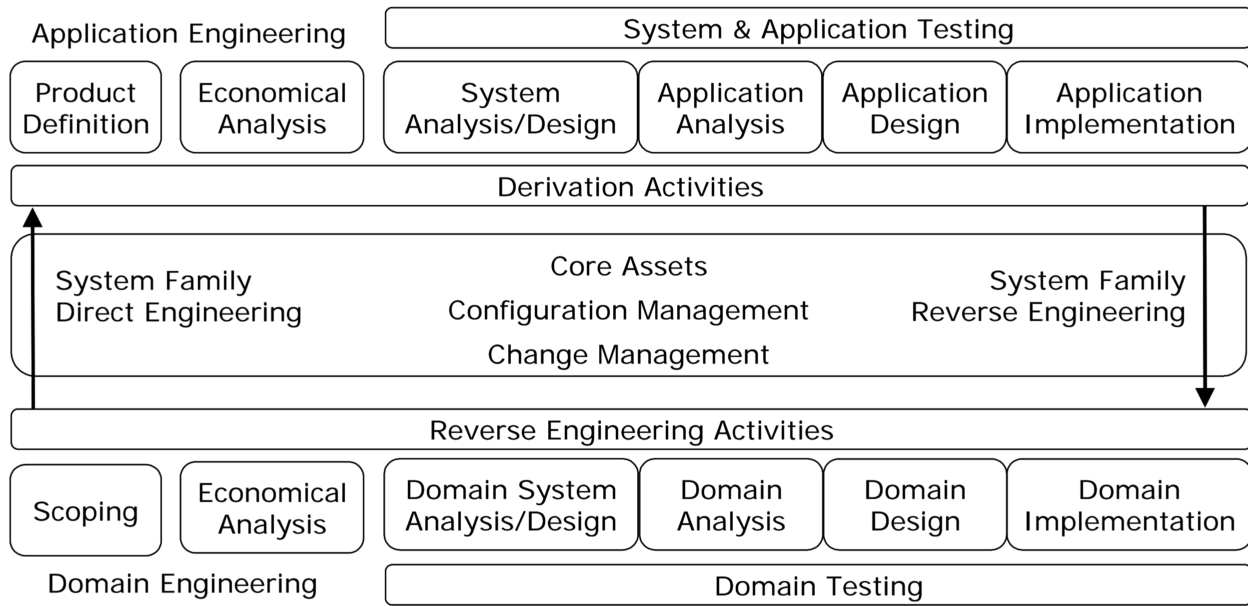

Figure 1: Development Process Model for Product Lines

populating an asset base and migrating existing single systems to the product line engineering paradigm. The same support is gained when enriching the asset base of a product line in order to address new requirements, or business goals.

The remainder of the paper is structured as follows: Section 2 introduces the product line engineering development process model developed in the FAMILIES project, then section 3 presents the generic asset incorporation approach. Section 4 continues with an asset classification, which is the basis for the selection a set of asset reverse engineering techniques presented in section 5. Section 6 presents related work, while section 7 finally draws conclusion on this work.

2. Product Line Engineering

Product line engineering introduces a systematic development approach that explicitly supports a family of similar systems. Such a family of products is designed to take advantage of their common aspects and predicted variabilities of a product line [1]. Figure 1 depicts the FAMILIES reference model, an overview on how an asset base (i.e., a product line infrastructure) supports the product line engineering activities. Domain engineering activities have the goal to develop, maintain and extend the infrastructure in form of an asset base. The domain engineering activities are balanced with application engineering activities, which actually build the concrete products. Application engineering activities use the core assets provided by the product line infrastructure. Thereby the generic artifacts contained in the asset base are used to build concrete products by resolving the variabilities. The resolution is made explicit, for instance by means of decision models.

Thus, central and crucial for successful product line engineering, are the core assets which contain the components that have a high product line impact, that provide key features, major variants or core functionality. When evolving a product line, or in a migration towards product line engineering, the software development organizations have the option to include existing assets (not developed in domain engineering, which means the asset are not prepared to suit the product line needs in most cases) instead of creating assets with similar functionality themselves. The main decision criteria are the quality of the existing assets and there suitability for the product line. When assessing an existing asset, the development organization has to decide for each asset on what to do to populate the asset base:

- **As-is Reuse**: Reuse the asset as it is, no or only small modifications to the asset are required. The asset quality and suitability is in such a manner so that it can be migrated to the product line infrastructure with limited effort.
- **Recovery and adaptation**: Recover an existing asset from an asset producer with reverse engineering analyses techniques and use the information to improve or rebuild the asset and adapt it so it fits into the product line infrastructure and fulfills the acceptance criteria (refactoring).
- **(Re-) Implementation**: Development and realization from scratch of a new asset and therefore rejecting the existing asset. In case of a re-implementation concepts coming from existing assets may contribute to the implementation.

In the next sections we show how existing assets can be incorporated into the product line infrastructure, how to classify assets, and introduce a common process for asset recovery.

3. Asset Incorporation

We use the Software Process Engineering Meta-model (SPEM, see [2], developed to describe concrete software development processes) as notation for the asset incorporation process. The process to incorporate assets into the product line infrastructure is mainly focused on domain engineering activities, in particular domain analysis, domain design, and domain implementation (see Figure 2).

In the activity of **domain analysis**, three essential sub processes were identified:

- **Need Analysis**: The results of need analysis are generic asset descriptions formulating the needs the asset has to fulfill in order to be appropriate for the product line infrastructure. For this purpose, the product line architects participate on requirement engineering activities such as requirements elicitation, negotiation and so on. The need analysis depends on the internal processes of the development organization; it may range from formal requirement engineering activities to agile modeling approaches. The need analysis has to make clear why an (existing) asset is needed.
- **Asset Recovery**: The output of this activity is a set of existing assets that are considered to fulfill the needs previously identified. This activity is performed together by product line architects, developers, and domain experts.
- **Asset Evaluation**: In this process, we propose to use existing standards for process evaluations, for instance ISO 14598 [3] or GQM [4]. However, if time constraints advice to not use such a formal approach, other approaches can be used, such as internal processes. Nevertheless, in this paper an asset evaluation is considered to be mandatory, because after knowing the generic asset to be incorporated, this asset has to pass an evaluation processes where its impact, cost, and quality is assessed. At this stage a decision may be made: If there are no assets that pass all the evaluation criteria, another need analysis should be carried out, otherwise the evaluation boundary is reduced and the best product fulfilling the evaluation is selected.

In the activity of **domain design**, there is an important process to be performed:

- **Architectural Asset Incorporation**: When including existing assets in the asset base of the product line, it is necessary to do this compliant to the product family architecture. It is one of the most critical steps because of the interaction with other core assets, potential side effects, and technical constraints. For instance, an unexpected interaction may lead to the instability of the architecture, which has to be avoided. Thus, as a result of this process the architectural compliance has to be ensured, in case it has to be updated. In realization of these architectural adaptations, impacts on the domain design are very likely. This activity is driven by the product line architects.

Finally, in the activity of **domain implementation**, there is also an important process to be performed:

- **Technical Asset incorporation**: This incorporation differs from the architectural asset integration in that *technical aspects* of the incorporation are taken into

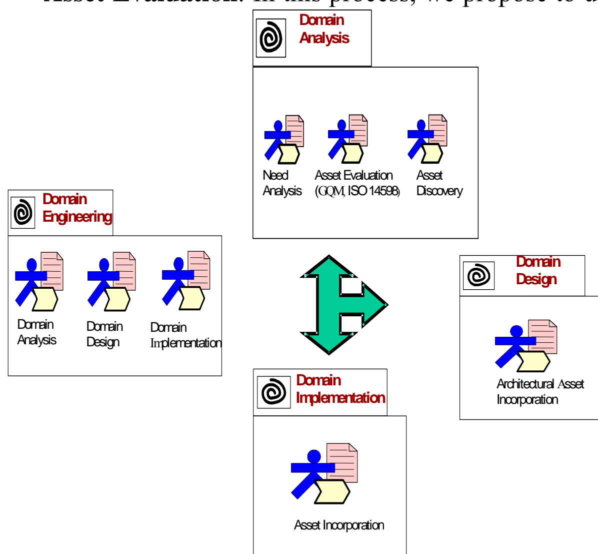

Figure 2: Domain Engineering

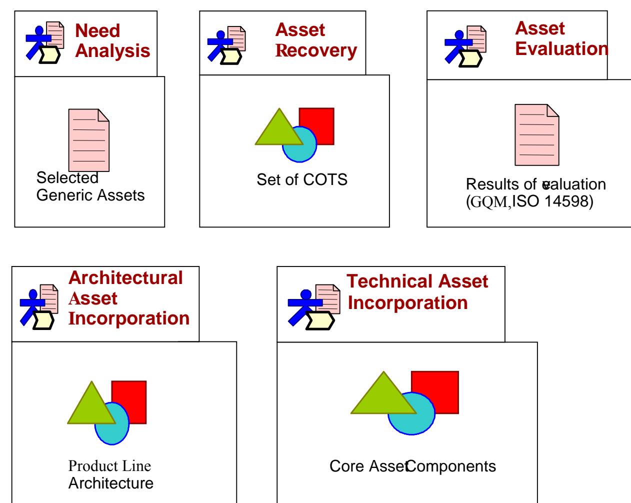

Figure 3: Results per Major Activity

account. Thus, interactions with other core assets are taken into account at low level. Configuration, change and traceability management activities become essential. This activity may involve a wide range of technical problems, but these types of problems can be solved. However, if there is an architectural mismatch, the problem will be still present even if the technical incorporation works well. Product line developers and component engineers are responsible of accomplishing this task.

Figure 4 depicts an overview of the process activities and roles, and how the different activities interact, while Figure 3 presents the results of the major activities. Thereby all the roles, products and sub-processes are include, so that it is possible to see the "whole picture".

4. Asset Classification

An existing asset has a distinct origin, describing the development unit that produced the asset.

- **Legacy**: The asset is developed, maintained, and managed by the same development organization unit that is responsible for developing the product line infrastructure (i.e., the domain engineering).

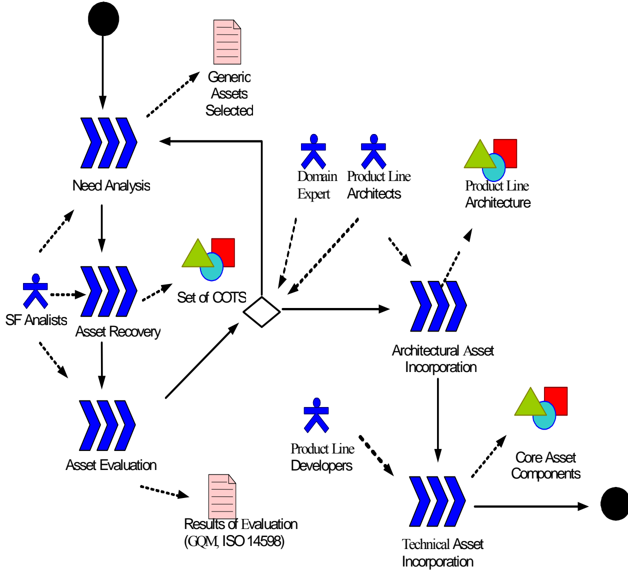

Figure 4: Asset Incorporation Phases and Roles

- **In-house**: The asset is developed, maintained, and managed by another in-house development organization unit that is not responsible for the domain engineering. The unit responsible for domain engineering has only limited influence on and access to the other unit since the other unit has other organizational objectives to achieve.
- **3rd party**: Assets have a 3rd party origin when they are developed by another organization not under control of the product line development organization. This means that the domain engineers are typically not available but there might be a support by the asset producers (e.g., hotlines). Typical examples for 3rd party assets are component of the shelf (COTS).
- **Open source**: Open source communities can produce assets that are of interest for a development organization. The development organizations can even decide to contribute to the open source development themselves, but they do not have to. Popular examples of open source assets that can be incorporated into the asset base of a product line infrastructure are, for instance the Eclipse platform and its various plug-ins.
- **Combinations**: Assets that evolved over a longer time period can have combined origins from the above list. This is especially true for large-scale development organizations that buy and sell organizational development units, and for open source assets that are adapted by a development organization to their specific needs.

Information Sources	Legacy	In-House	3rd Party	Open Source
Requirements specification	+	?	-	-
Architecture description	+	?	-	-
Design documentation	+	?	-	-
Interface documentation	?	?	X	-
Source code	X	X	-	X
Test cases	X	+	-	+
User manual	X	X	X	X
Version history	?	?	-	X
Bug reports	?	?	-	X
Asset expert	+	?	-	-

Table 1: Asset Information Sources

Depending on the asset origin there are different information sources from which assets are retrieved. Table 1 lists these information sources.

In this paper we assume typical cases ignoring the facts that individual cases may differ and respective development organizations have a certain degree of maturity producing artifacts as prescribed by most software development process models. The availability of information sources is classified by the asset origin, an

“X” denotes that the information source is available in all cases, a “+” in most cases, “?” means the availability is unclear, while a “-“ means for the unavailability of the information source.

5. Asset Recovery Techniques

Table 1 presented the different information sources that are available for different assets, depending on their origin. Based on this classification, it is possible to select appropriate reverse engineering technique to support the asset incorporation process as described in section 3. We first present a generic recovery process that spans over the individual techniques, and then we present a selection of techniques addressing the specific information sources (or a combination of information sources).

5.1. Recovery Process

The implemented system is essential in the recovery process, but it involves other factors and sub-processes. The recovery process is made up of five input data, five sub-processes, and four significant results.

The input data required for the recovery process are: Available documentation, Source code, System in run-time, Patterns and Expert information.

Several activities should be achieved in the recovery process (see Figure 5). The first one is Information extraction, their input data are the Available documentation and Source code. This process can be aided by experts [5], by obtaining information from user documentation [6], using techniques such as gathering [7], lexical analysis [8] or pattern matching [9]. The Information extraction objective is to obtain a Conceptual model the system.

Static-view extraction is the most common approach in the re-engineering process. By using tools the system static view is obtained from source code (classes, packages, interfaces, relationships between them and other relevant architectural elements). Sometimes this model is complemented by information from conceptual model [5]. As a result of this process an architectural static view is obtained [10].

Dynamic-view extraction obtains the system behavior. That is, by obtaining the traces from system-user or system-environment interactions [10]. As result of this process, an architectural dynamic view is obtained [8].

Abstraction, two essential objectives should be carried out in this process: 1) Reduce the complexity of the preliminary architecture, by increasing the abstraction level and 2) filter the preliminary architecture to the interest topic, for example; communication, security, management, etc. As result of the abstraction process, a refined architecture is obtained.

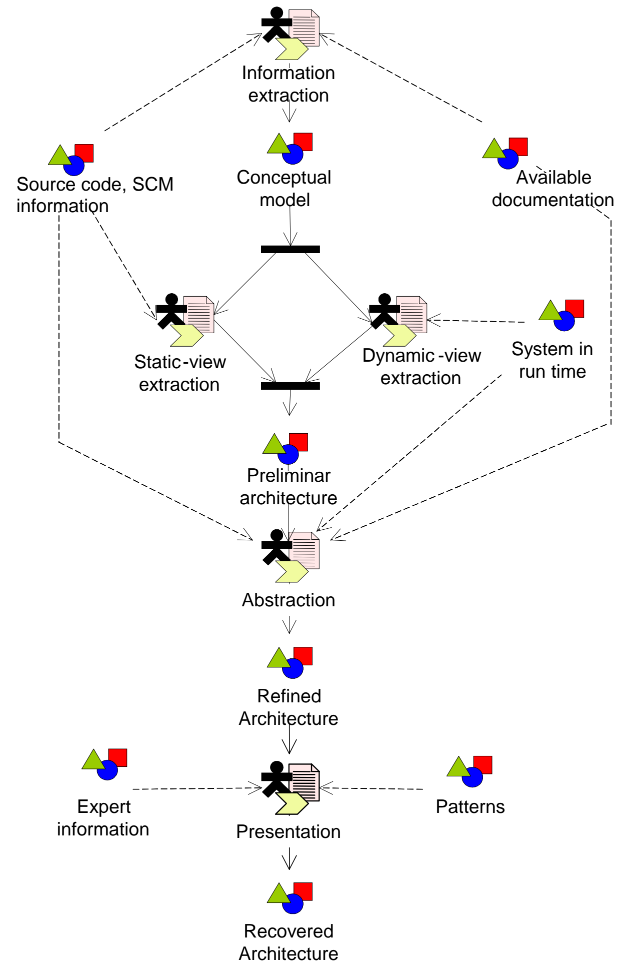

Figure 5: Recovery Process

Finally in Presentation, once the refined architecture is obtained, it is polished by experts and supported with reference patterns (see [5], [9], [11], [12]. As result of this process, the Recovered architecture is obtained that represents the ”as-built” architecture of a system (set of architectural views regarding different architectural aspects).

In the recovery process, there are partial results: Conceptual model or system meta-architecture, in MDA model known as Computer Independent Model (CIM), it is a set of concepts and the relationship between them. Preliminary architecture is made up of static and dynamic views of the system. Refined architecture are abstracted views of the preliminary architecture used to isolate certain architectural aspect. Finally, Recovered architecture, the refined architecture is rarely the definitive architecture; with the help of experts and patterns, an architecture close to “as-built” architecture of the system is obtained.

5.2. Reverse Engineering Techniques

The reverse engineering techniques introduced in this section are ordered by the type of analyses, namely static, dynamic, document, and historic analyses.

5.2.1. Static Analyses

Static analyses extract information mainly from the source code but without executing it. The output of static analyses range amongst others: static decomposition, hierarchies, static metric values, responsibilities, interfaces, naming conventions, dependencies, etc.

5.2.1.1. Architecture Evaluation

Static information can be used to refine a model (idealized architecture) with the actual architecture. This can be done to iteratively refine the expected mental model and the documentation before a specific activity and to track the difference and guide the architecture towards a to-be status.

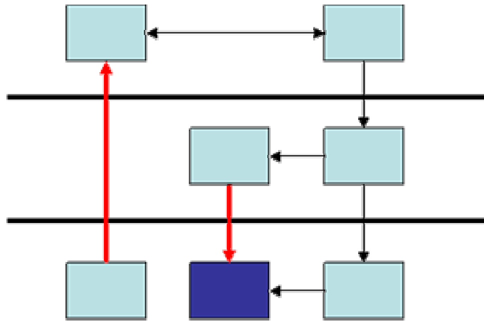

Figure 6: Example Architecture Evaluation

This activity is supported by an architecture evaluation tool (e.g., see the reflexion model technique [13] or the SAVE tool [14]). In both techniques experts describe the components and the relationships they expect among them, then they map these components to code constructs (e.g., files, classes, methods). The tool compares the difference between the expected relationships and the ones found in the system. The experts refine his model or the mapping, or the reverse engineers adapt the fact extraction process.

Figure 6 depicts an example for such an architecture evaluation. The red arrows indicate the divergences between the mental model and the dependencies extracted from the source code.

5.2.1.2. Interface Analysis

Documented interfaces are one of the prerequisites of effective reuse of components. Reuse works when the developers know which functionality is provided by a component and how to access the functionality implemented in such a component. Components in the context of interface analysis are collections of source code entities (e.g., files, groups of logically related routines, single or groups of classes or packages, or even whole subsystems). Applications of the interface analysis technique work with the following motivations:

- Reduction of the complexity of given components with respect to the number of offered routines by minimizing the provided interfaces to only the actually used interfaces when to facilitate reuse.
- Documentation of source code spots in usage lists where to change accesses to a component when migrating the software system towards component-based development.
- Extension of architectural descriptions (e.g., the module and/or the code view) by explicit notation of the provided functionality of a component.
- Migration of a group of entities towards an encapsulated component with explicit boundaries.

```
Required Routines of class1.method1:
   Class2.method2
      Located in:  c:/code/file2.pas
      Signature:   procedure class2.method2(val:integer)
      Returns:     -
```

Figure 7: Example Interface Analysis

Interface analysis reveals the connections of the subject component to the rest of the software system, or if it should become a real component in future, it documents the spots to be changed and how the future component is embodied in the system (see [15] for details).

5.2.1.3. Technique: Conformance and Recovery Processes

Conformance and recovery are two processes used in evolutionary software development. They are used as mechanisms for a quick feedback, to increase code reusability, and to increase quality. In traditional systems these processes allow implemented assets to be reused and compare them to a standard. Product line conformance and recovery processes have an additional value; both can be used to locate commonalities, variation and variation points. The orchestration of these processes is presented in Figure 8. The processes proposed focus on quality aspects [3] such as performance, security, usability and so on.

The conformance process needs previous phases where objectives and focus are defined. Both take input stakeholder requests and the quality of service as relevant. Then, two parallel activities should be achieved. The system architecture is obtained from the implementation domain, using the recovery process (recovered architecture). But conformance with respect to a particular quality is complex to solve between architectures, so a filtering process is required in order to obtain the Significant Implemented Assets (SIA); the

filtering process only selects assets related to a particular quality. On the other hand, similar processes should be achieved from the standard domain. The standard usually has a rich documentation, therefore an exemplification process deducts the generic architecture (standard architecture), if it is not defined in the standard (partial or totally). And finally, in the same way as in the implementation domain, the Significant Standard Assets (SSA) are extracted using a filtering process.

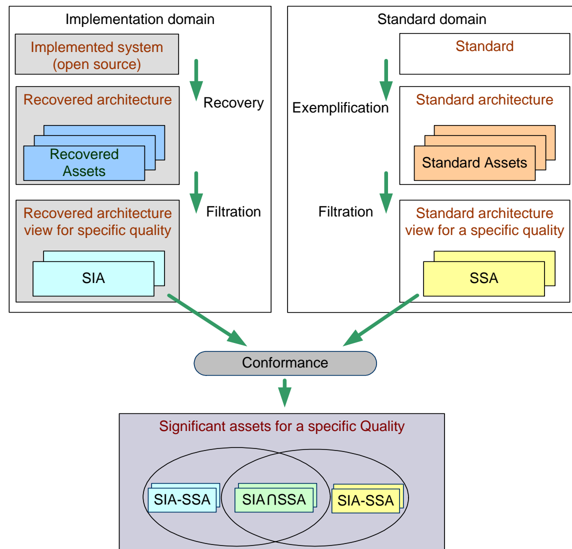

Figure 8: Conformance Process

Conformance process compares and identifies differences and coincidences, a number of methods and techniques could be used, such as: Ontology-based algorithms that search for common artifacts in a architecture [16], Numerical and graph-based algorithms to reduce complexity, Use cases to isolate parts of a system, Comparison of the abstract syntax tree of similar systems, Measurement of similarities using metrics (internal or external as defined to measure quality characteristics [17]) and so on.

Three relevant results are obtained:

- Proposal for enhancement of the SIA (SSA-SIA): as a product of the difference between the SSA and SIA, new requirements are identified and some deficiencies have been located in the current implemented architecture.
- Proposal for a standard (SIA-SSA): as a product of the difference between the SIA and SSA, some deficiencies have been located in the reference standard; it is frequent when technology goes beyond the standards.
- Common and variation point identification (SIA ∩ SSA): The common assets are identified and variation points located, it may be the main result of an implementation accreditation with respect to a standard.

5.2.2. Dynamic Analyses

Dynamic analyses extract information by instrumenting and executing source code. The output of dynamic analyses range amongst others: runtime traces, runtime behavior, execution and runtime metrics, source code element interaction, etc.

5.2.2.1. Technique: Dynamic Traces

The main aim of using dynamic or run-time traces is to recover sequence diagram. Sequence diagrams show the actually interaction among software components together with the messages they exchange over a period of time. A sequence diagram captures interaction among entities (for e.g., classes, components). In addition, it captures thread interaction. Sequence diagrams are a good source for understanding how a particular scenario or use case works when a program runs.

Such information for instance can be used to analyze the performance. Knowing how many objects are created when a program runs and the life-time of each object helps us to build a run-time model.

One of the challenges in recovering sequence diagrams is level of abstraction. A running program contains low-level information like function/method call, field/variable referenced by a function/method, thread starting another threads. This low-level information is difficult to analyze manually and deriving useful knowledge from it needs abstraction.

To build abstraction into the run-time traces, we make use of static information (e.g., classes, packages, files, folders, components, subsystems or layers) and use-cases (to find a high-level meaning for the interaction, e.g., provided by user manuals). Once an abstracted sequence diagram is constructed, this can be used to check consistency between specified sequence diagram and the running system

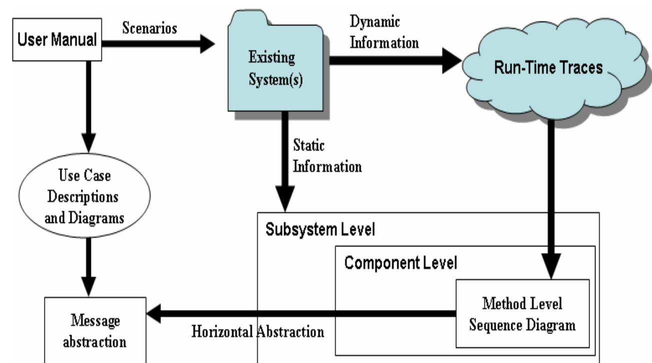

Figure 9: Example Dynamic Traces

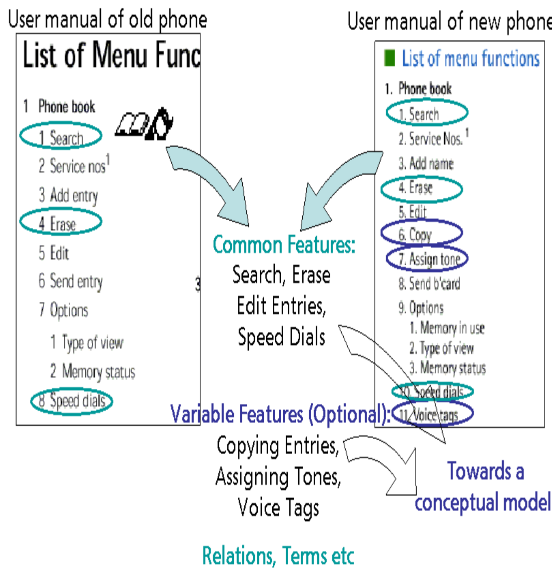

Figure 10: Example CaVE

5.2.3. Document Analyses

Document analyses extract information by analyzing the documentation of a system. The output of document analyses range amongst others: use cases, features, use case diagrams, description of functionality, commonalities and variabilities between products, conceptual models, domain wording, etc.

5.2.3.1. Technique: CaVE

CaVE (Commonality and Variability Extraction) is an approach enhanced with techniques for structured and controlled integration of user documentation of existing systems into the product line. Until now, the information needed to build a product line model is elicited interactively with high expert involvement. As domain experts have a high workload and are often unavailable this high expert involvement is a risk for the successful introduction of a product line engineering approach in an organization. The CaVE approach overcomes the following problems:

- Domain experts have a high workload and are hardly available so we need to relieve the experts by eliciting of product line related information from documents.
- There is a lack of guidance on how to integrate legacy information found in documents into product line models.
- There is no variability management approach that is general enough to integrate all kinds of artifacts into a product line model.
- Single system elicitation methods cannot be taken as they are because multiple documentations have to be compared, Commonalities and variabilities have to be elicited and additional concepts (e.g. abstractions, decisions) are needed.

With CaVE, common and variable features, Use Case elements, decisions and requirements can be elicited. The approach consists of the following phases:

- Preparation: The product line engineer prepares the user documentation and selects the appropriate extraction pattern.
- Analysis: The product line engineer analyses the documents with the selected extraction pattern and marks the elements found.
- Selection, and change: The selected elements are put together to partial product line artifacts and presented to the expert who can change elements and add additional information

The first two steps of the approach can be performed by persons who just have a slight domain understanding, they do not have to be domain experts. The third step requires involvement of domain experts (see [6] for details.

5.2.4. Historic Analyses

Historic analyses extract information by analyzing the data of a system. The output of historic analyses comprises change dependencies and bug dependencies. They indicate coupling dependencies used to assess the quality of assets and dependencies between them.

5.2.4.1. Technique: Change Coupling Analysis

For the extraction of change and bug dependencies we retrieve modification reports from versions systems (e.g., CVS) and problem (bug) reports from bug tracking systems (e.g., Bugzilla) and store them in the Release History Database (RHDB) [19]. Both reports refer to a product (i.e., asset) that is managed by these systems.

Change couplings are computed based on modification reports. A change coupling between two assets is established whenever changes to the two assets have been committed to the repository by an author in the same transaction. A bug dependency refers to change couplings with respect to a problem that has been fixed. The strength of change couplings is computed for a specified observation period, for instance from the last release to the recent release.

The number of change and bug dependencies and their strength is input to the asset incorporation process, in particular, to the architectural as well as technical asset incorporation processes. Change couplings denote interactions between assets on the architectural level as

well as technical level. For instance, on the technical level we determine change couplings between source files. By abstracting these coupling to the level of architecture, such as between features or software modules, we determine change couplings on the architectural level [20], [21]. On both levels we can use the number and strength of change couplings to assess the feasibility and effort to incorporate an asset into the asset base.

The technique can be applied to the different kinds of assets that are managed by configuration management systems including the source code as well as the different project documents.

6. Related Work

Related work concerns in general architecture recovery and feature location techniques used to extract and determine assets for building an asset base.

Regarding architecture recovery a number of tools have been developed that can be used to extract higher-level views on the implementation of software systems. Tools are, for example Bookshelf [24], Dali [25], or Rigi [23]. They follow the Extract-Abstract-View Metaphor described in [22]. Most of these tools differ in the underlying fact extraction technique, in the methods and details of fact representation, and in the analysis and visualization techniques.

In [22] Ebert et al. introduced GUPRO which is an integrated workbench that supports program understanding of heterogeneous systems on arbitrary levels of granularity.

The SAR method described by Krikhaar [5] concentrates on creating higher-level views on the architecture. The approach is based on Relational Partition Algebra [12] and defines a process for selecting the information sources from which higher-level views are abstracted.

Riva proposed a view-based architecture reconstruction approach named NIMETA [26]. Similar to Krikhaar the approach is based on relational algebra. NIMETA emphasizes the scrupulous selection of architectural concepts and architecturally significant views that are reflecting the stakeholders' interests.

Regarding feature location a number of approaches exist. Concerning the feature location in source code Wilde et al. presented pioneering work. They introduced the Software Reconnaissance approach that based on the execution of test cases determines features [27].

Eisenbarth et al. based their approach on the Software Reconnaissance technique and extended it by using the concept analysis technique for determining features [18].

A similar approach has been also presented by Wong et al. They analyze execution slices of test cases to determine the source code units that implement a feature [28]. These techniques can be integrated into our asset recovery and incorporation process.

7. Conclusion

Incorporation of assets into the asset base of a product line infrastructure has to ensure that the quality of the assets to be integrated suits the needs of the product line. Therefore, it is crucial to have a well-defined defined integration process and a set of reverse engineering techniques to analyze the assets and to assess its internal quality. This work presents such an asset integration process along with a set of used reverse engineering techniques.

In this work, we present a quality-drive asset incorporation process to integrate existing assets into the asset base, and relate a set of reverse engineering techniques to the recovery of assets of different origins. According to the information sources we present reverse engineering techniques to recover assets from existing information sources, such as static and dynamic, document, and historic analysis.

Out asset incorporation process evaluates recovered assets with respect to their need and quality. Quality is crucial because one asset may break the product line infrastructure, therefore we steer the incorporation process by evaluations (based on ISO or on GQM tree) to ensure the required qualities of the incorporated assets. This process assures the incorporation of those assets that are needed and fulfill that quality criteria whereas the other recovered assets are left out.

Future work includes refinement of the reverse engineering techniques and development of new techniques addressing information source not yet dealt with (e.g., test cases).

Another topic of ongoing work is to formulate guidelines that help the product line architects to monitor the asset incorporation process, and to develop customized GQM trees, which steer the analysis activities for distinct domains (assuming that different domains will differ in the required qualities as well).

8. Acknowledgements

We are grateful to the national ministries of Austria, Germany, Finland, and Spain for partially funding our work under EUREKA 2023/ITEA-ip00009 'FAct based Maturity through Institutionalization Lessons-learned an Involved Exploitation of System-family engineering' (FAMILIES).

Jose L. Arciniegas is a visiting scientist from Universidad del Cauca, Colombia. His work has been partially developed in the project TRECOM, granted by

the Spanish Ministry of Science and Technology under reference TIC2002-04123-C03-01.

9. References

[1] Weiss, David M.; Lai, Chi Tau Robert: Software Product-Line Engineering. A Family-Based Software Development Process, Addison-Wesley, 1999.

[2] Object Management Group (OMG): Software Process Engineering Metamodel Specification version 1.1, 2002.

[3] ISO 9126. Software product evaluation: Quality characteristics and guidelines for their use. ISO/IEC 9126. ISO, Geneva, Switzerland, 1991.

[4] Basili, V. R., and D. M. Weiss: A Methodology for Collecting Valid Software Engineering Data, IEEE Transactions on Software Engineering, Vol. SE-10, pp. 728–738, 1984.

[5] Krikhaar, R. Software Architecture Reconstruction. Ph.D. Thesis, University of Amsterdam, June 1999.

[6] John, I Dörr, J. Elicitation of Requirements from User Documentation, Ninth International Workshop on Requirements Engineering: Foundation for Software Quality. Refsq '03. Klagenfurt/Velden, Austria, June 2003.

[7] Boucetta, S. Hadjami Ben Ghezala, H and Kamoun, F. Architectural Recovery and Evolution of Large Legacy Systems. Proceedings of the International Workshop on the Principles of Software Evolution. Japan. July, 1999.

[8] Kazman, R. O'Brien, L.and Verhoef, C, Architecture Reconstruction Guidelines, 2nd Edition (CMU/SEI-2002-TR-034), 2002.

[9] Sartipi, K. and Kontogiannis, K. A Graph Pattern Matching Approach to Software Architecture Recovery, Proceedings of the IEEE International Conference on Software Maintenance, Florence, Italy, pp. 408-419, November, 2001.

[10] Ahmed E. Hassan and Richard C. Holt, Architecture Recovery of Web Applications. In Proceedings of the International Conference on Software Engineering, Orlando, Florida, pp. 19-25, May 2002.

[11] Guo, G. Atlee, J. and Kazman, R. A Software Architecture Reconstruction Method. Proceedings of the First Working IFIP Conference on Software Architecture, San Antonio, Texas, pp. 225-243, February, 1999.

[12] Feijs, L, Krikhaar, R., and Van Ommering, R., A Relational Approach to Support Software Architecture Analysis, Software Practice and Experience, Vol 28(4), pp. 371-400, April 1998.

[13] G. C. Murphy, D. Notkin, K. Sullivan: Software Reflexion Models: Bridging the Gap between Source and High-level Models, ACM Software Engineering Notes, 1995.

[14] P. Miodonski, T. Forster, J. Knodel, M. Lindvall, D. Muthig: Evaluation of Software Architectures with Eclipse, Kaiserslautern, (IESE-Report 107.04/E), 2004.

[15] J. Bayer et al: Definition of Reference Architectures based on Existing Systems, (IESE-Report 034.04/E), 2004.

[16] OMG. Ontology definition Metamodel. Request for proposal. August 18. 2003.

[17] ISO/IEC JTC1/SC7/WG6 N461. Information Technology – Software product quality –Part 1: Quality Model, Part 2: External Metrics, Part 3: Internal Metrics, Part 4: Quality In Use Metrics. ISO/IEC 9126, November 1999.

[18] Eisenbarth, T., Koschke, R., and Simon, D.: Locating Features in Source Code, IEEE Transactions on Software Engineering, March 2003.

[19] Michael Fischer, Martin Pinzger, and Harald Gall. Populating a Release History Database from Version Control and Bug Tracking Systems. In Proceedings of the International Conference on Software Maintenance, pp. 23–32, Amsterdam, Netherlands, September 2003.

[20] Michael Fischer, Martin Pinzger, and Harald Gall. Analyzing and Relating Bug Report Data for Feature Tracking. In Proceedings of the 10th Working Conference on Reverse Engineering, pp. 90–99, Victoria, B.C., Canada, November 2003.

[21] Martin Pinzger, Michael Fischer, and Harald Gall. Towards an Integrated View on Architecture and its Evolution. Electronic Notes in Theoretical Computer Science, 127(3):183–196, April 2005.

[22] Jürgen Ebert, Bernt Kullbach, Volker Riediger, and Andreas Winter. Gupro - Generic Understanding of Programs. Electronic Notes in Theoretical Computer Science, 72(2):59–68, 2002.

[23] Kenny Wong. The Rigi User's Manual — Version 5.4.4. University of Victoria, 1998.

[24] Patrick Finnigan, Richard C. Holt, Ivan Kallas, Scott Kerr, Kostas Kontogiannis, Hausi A. Müller, John Mylopoulos, Stephen G. Perelgut, Martin Stanley, and Kenny Wong. The software bookshelf. IBM Systems Journal, 36(4):564–593, November 1997.

[25] Rick Kazman and S. Jeromy Carriere. Playing detective: Reconstructing software architecture from available evidence. Automated Software Engineering, 6(2):107–138, 1999.

[26] Claudio Riva. View-Based Software Architecture Reconstruction. Ph.D. thesis, Vienna University of Technology, 2004.

[27] N. Wilde and M.C. Scully, Software Reconnaissance: Mapping Program Features to

Code, Journal of Software Maintenance: Research and Practice. vol. 7, pp. 49-62, Jan. 1995.

[28] W.E. Wong, S.S. Gokhale, J.R. Horgan, and K.S. Trivedi, Locating Program Features Using Execution Slices, In Proceedings of the IEEE Symp. on Application-Specific Systems and Software Engeering and Technology, pp. 194-203, Mar. 1999.