An Architectural Approach to Support Online Updates of Software Product Lines

Danny Weyns, Bartosz Michalik, Alexander Helleboogh, Nelis Boucké
DistriNet Labs, Katholieke Universiteit Leuven
Celestijnenlaan 200A, 3001 Leuven Belgium
{danny.weyns,bartosz.michalik,alexander.helleboogh,nelis.boucke}@cs.kuleuven.be

ABSTRACT

Despite the successes of software product lines (SPL), managing the evolution of a SPL remains difficult and error-prone. Our focus of evolution is on the concrete tasks integrators have to perform to update deployed SPL products, in particular products that require run-time updates with minimal interruption. The complexity of updating a deployed SPL product is caused by multiple interdependent concerns, including variability, traceability, versioning, availability, and correctness. Existing approaches typically focus on particular concerns while making abstraction of others, thus offering only partial solutions. An integrated approach that takes into account the different stakeholder concerns is lacking. In this paper, we present an architectural approach for updating SPL products that supports multiple concerns. The approach comprises of two complementary parts: (1) an update viewpoint that defines the conventions for constructing and using architecture views to deal with multiple update concerns; and (2) a supporting framework that provides an extensible infrastructure supporting integrators of a SPL. We evaluated the approach for an industrial SPL for logistic systems providing empirical evidence for its benefits and recommendations.

Categories and Subject Descriptors

D.2.7 [**Software**]: Software Engineering—*Maintenance*

General Terms

Design

Keywords

Software product lines, online updates, viewpoint

1. INTRODUCTION

Over the last decade, many companies have successfully adopted a software product line (SPL) approach. Clements and Northrop [7] define a SPL as a set of software-intensive systems (products) that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and are developed from a common set of core assets in a prescribed way. The key underlying success factor of SPL is the integrated perspective on reuse which includes all aspects of the company; from business strategy to architecture, processes, and the organization structure. Reported benefits of SPL adoption include [20]: improved productivity, enhancement of quality, and reduction of time to market.

Despite the success of SPL, managing the evolution of SPL for large-scale distributed systems remains difficult and error-prone [16]. Evolution of a SPL entails many aspects. A first aspect is the strategic planning of how to evolve a SPL over time in terms of scope. A second aspect is the concrete redesign and the development of the updated product line assets[1]. A third aspect are the actual update tasks integrators have to perform to update one or more deployed products of a SPL. The research presented in this paper is concerned with this third aspect. Our particular focus is on updating SPL products that require online updates with minimal interruption. Typical update scenarios are extending a deployed product with a new feature, replacing an asset (or parts of it) to solve a particular problem, etc. The complexity associated with performing update tasks of deployed SPL products is caused by the many interrelated concerns. From a literature study and our experience with SPL for complex distributed systems, we identified the following important concerns:

1. *Variability.* A key aspect of SPL is that products are derived from a shared set of core assets. However, the combination of assets is constrained, which captures the variabilities of the product line.

2. *Traceability.* Traceability refers to the ability to link assets of the SPL (documents, libraries, code, etc.). Tracing assets is crucial to understand the relations and dependencies among various assets of a SPL. This is particularly relevant for SPL which comprises legacy assets with incomplete or outdated documentation.

3. *Versioning.* SPL assets evolve over time to solve bugs, improve performance, enhance security, etc. With each release new versions of the assets can be added to the SPL. However, new versions of assets may not directly be applied to all products. As a result, multiple versions of SPL assets typically co-exist which are not always compatible.

4. *Availability.* Modern large-scale distributed software products often have demanding availability requirements. Therefore, the time (parts of) the deployed systems have to be shut down during updates should be minimized.

[1]With asset we refer to both core assets (the reusable artifacts of a SPL from which products are composed) and product-specific assets (the additional artifacts developed for one particular product).

5. *Correctness.* A correct update requires a valid sequence of update steps (adding/removing/replacing resources and stopping/starting processes) that bring the deployed system to the new configuration without compromising its consistency. Faulty updates increase integration costs and harm the reputation of the SPL owner.

A number of methods and mechanisms have been proposed to cope with SPL evolution, and updates in particular. However, these approaches typically focus on particular concerns while making abstraction of others, thus offering only partial solutions. An integrated approach that takes into account the various stakeholder concerns is lacking. This paper contributes with a comprehensive approach for online updates of SPL that ensures consistent integration of the changes in the affected SPL products. The approach takes an architecture-centric perspective and supports multiple concerns of SPL updates. It comprises two complementary parts: (1) an update viewpoint, and (2) a supporting framework. The update viewpoint defines the conventions for the construction and use of architecture views to deal with concrete update concerns of the stakeholders. The supporting framework provides a reusable and extensible infrastructure that assists integrators with product updates. The framework offers support for harvesting relevant architectural knowledge, analyzing the knowledge, constructing the view models, updating the SPL products, and checking consistency.

We have applied the approach to support integrators with the updates of an industrial SPL for logistic systems. This case study was performed in a joint R&D project between Egemin[2], an industrial manufacturer of logistic systems, and DistriNet Labs at K.U.Leuven. The case study is a representative example of a complex industrial SPL which consists of multiple subsystems (automated guided vehicles - AGVs, cranes, etc.). The different concerns with managing the updates of SPL mentioned above apply to the case.

The remainder of this paper is structured as follows. In section 2, we introduce the case study and describe the problems the company faces with updating their SPL products. Next, we present the constituent parts of our approach for updating product lines: the update viewpoint (section 3), and the supporting framework (section 4). In section 5, we report the results of an empirical evaluation of the approach. Section 6 discusses related work. Finally, we draw conclusions and look at future work in section 7.

2. PROBLEM

In this section, we pinpoint the problems with online updating of SPL products using a case study. We start by introducing the case. Then we illustrate the concrete problems with online updates.

2.1 Case

Egemin is a leading company that provides full life cycle support for logistic systems. Such systems are used for warehouse automation, e.g., for distributing manufactured products to storage locations or as an interprocess system between various production machines.

A logistic system has a three-layered architecture. The bottom layer is a *distributed host infrastructure* in .NET that provides basic support for sensor and actuator interfacing, network communication, etc. The *logistic platform* (middle layer) is a service framework developed by Egemin that provides common middleware services for logistic systems. The platform makes use of the distributed host infrastructure, providing general support for system configuration, communication, persistency, security, logging, visualization, and diagnosis. The logistic platform also offers basic support for dynamic updates, including the (de-)activation of components, buffering of messages, etc. Depending on the client-specific requirements, a logistic system typically integrates a number of *logistic subsystems* (top layer) that make use of the logistic platform. These logistic subsystems are built from a common set of components that can be customized and composed to the needs of the customers. Typical logistic subsystems include a warehouse management system (E'wms®) that is responsible for managing tasks in the system, and various transportation subsystems with control software such as automated guided vehicles (E'tricc® - Egemin transport intelligent control center), cranes (E'car® - Egemin crane automatic storage and retrieval system), and conveyers (E'con®).

Currently, Egemin's SPL comprises around 200 deployed logistic systems with 6 different subsystems to compose logistic systems. Per year, Egmin deploys 20 to 30 new logistic systems.

2.2 Problems with SPL Updates

Logistic systems are long-lived systems (typically 10+ years). During this lifespan, systems obviously have to evolve. On average, a logistic system requires three update tasks during the first year after deployment and an additional one per year afterwards.

Figure 1 shows a typical configuration of a logistic system. The software is deployed on four hosts. Each logistic subsystem comprises a service and a client that makes use of the distributed logistic platform. The service offers the functionality of the subsystem, while the client provides a graphical interface to access the service. In the scenario, there are services and clients for warehouse management, and for controlling cranes and automated guided vehicles.

Some typical update scenarios are:

- The E'car service needs to be upgraded from version v10 to version v12 which includes a new stacking algorithm that reduces the average retrieval time by 12%.
- An intermittent synchronization problem with database access has been discovered. Solving this problem requires the replacement of the database access component of the logistic platform.
- A new conveyer belt has been introduced in the factory and its control software needs to be integrated with the logistic system. The E'con service will be deployed on a new host, while the client software will be added to Host 4.

Egemin faces various problems with such update scenarios. Besides the intrinsic complexity of the logistic systems[3], the problems are caused by concrete instances of the main concerns of SPL updates as we discussed in the introduction:

1. *Variability.* Logistic systems are built from a set of core assets. Each system is unique in its composition and client-specific extensions. However, core assets cannot be composed arbitrarily. The constraints between the SPL assets and their assemblies introduce a complex management problem.
2. *Traceability.* Egemin's SPL comprises a lot of legacy software components. Documentation of the deployed systems is often incomplete or outdated. In addition, logistic subsystems are developed by different teams that work relatively independent. The lack of documentation and detailed knowledge makes it difficult to trace the complex dependencies between the installed subsystems.

[2]http://www.egemin.com/

[3]The code base of a typical logistic system consists of several 100K lines of code. The software is spread over hundreds of interdependent components (.NET assemblies) that are deployed on a network of heterogenous computer systems.

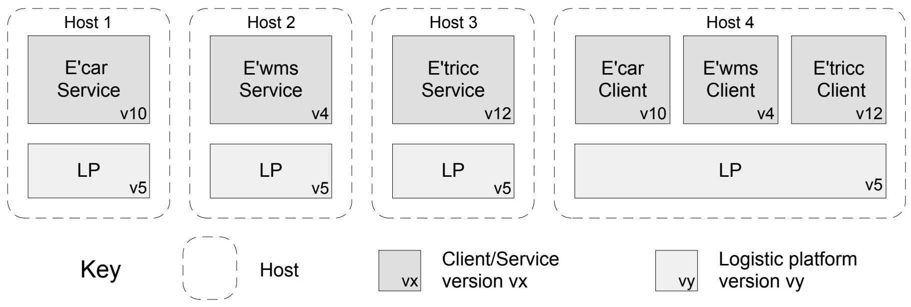


Figure 1: Typical configuration of a logistic system

3. *Versioning.* Multiple, often incompatible versions of logistic subsystems and their constituent components exist at the same time. Obviously, this adds to the complexity of updating logistic systems.
4. *Availability.* Logistic systems typically have to operate 24/7. Therefore, evolving the system with minimal interruption is crucial. Determining which processes (services and clients) have to be shut down and (re-)started and when is a complex problem. Restarting an incorrect configuration may compromise the consistency of the system.
5. *Correctness.* The lack of detailed information about deployed systems causes uncertainty about the correct sequence of update steps that have to be performed. As a result, only highly skilled engineers perform update tasks. But even so, it is well-known at Egemin that this approach does not guarantee that updates can be completed correctly. The only option is than to restore the system to the old version.

3. UPDATE VIEWPOINT

The central part of the approach we propose for updating deployed SPL products is the update viewpoint. The ISO/IEC 42010 standard [12] defines an architecture viewpoint as "a work product establishing the conventions for the construction, interpretation and use of architecture views to frame specific system concerns." As such, the update viewpoint establishes the conventions for defining and using architecture views to deal with the update concerns of SPL.

To the best of our knowledge, no architecture viewpoint to support software updates exists, either for regular software systems or for SPL. To define an update viewpoint for SPL, we combined a literature study on SPL evolution, including [16, 17, 20, 1, 3], with a series of interviews with stakeholders of Egemin's SPL. The focus of the study and the interviews was on identifying the key stakeholders involved in updating SPL and their main concerns, the relevant models, and the required analysis for evolving a SPL. From the literature study and the interviews, we constructed the update viewpoint for SPL. We documented the viewpoint using the template for architecture viewpoints proposed in the ISO/IEC 42010 standard [12] and illustrated in [11]. The update viewpoint enables identification of the concrete update concerns and discussion of the tradeoffs between the concerns.

Table 1 shows an overview of the resulting viewpoint. For brevity, we have omitted some details of the description. For the complete description of the viewpoint we refer the interested reader to [22].

Overview. The overview gives a brief summary of the viewpoint and its key features. The update viewpoint codifies reusable architecture knowledge to update SPL, independently of any particular context or technology. The focus of the update viewpoint is on the tasks integrators have to perform to update deployed products of a SPL with minimal interruption. The viewpoint can be used to guide stakeholders to construct a concrete update views for their particular context.

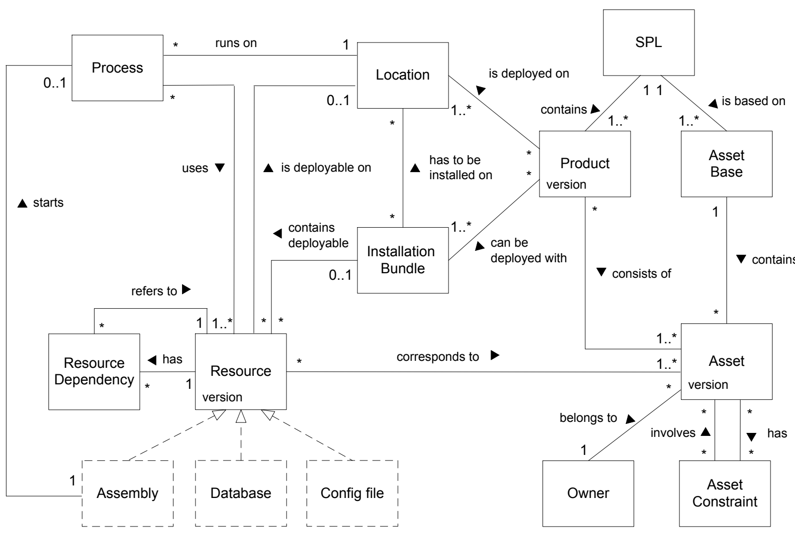
Concerns. The viewpoint captures the five main concerns of SPL updates as we discussed in the introduction and illustrated for Egemin in section 2.2. The questions for each concern help the architect(s) to interact with the other stakeholders and apply the viewpoint for their particular SPL. We illustrate this with two examples. Considering the question "On which locations are the assets deployed?" in the context of Egemin's SPL leads to the following clarifications w.r.t. the traceability concern: (1) a logistic system is a distributed system that is typically deployed on a heterogenous hardware and network infrastructure; (2) a location for Egemin's SPL is a directory on a particular host on which a part of the software of a logistic system is deployed; (3) the configuration/deployment of a logistic system is described in a project report. The deployment of a logistic system is also known in detail by the system administrator of the customer. Considering the question "How to perform the update with minimal interruption of the system" in the context of Egemin's SPL clarified the availability concern as follows: (1) when a logistic subsystem is shut down for an update, all ongoing transport tasks (e.g. AGVs that are carrying a load) first have to be completed to avoid inconsistencies, (2) as a result, shutting down a typical logistic subsystem (e.g. the AGVs or the cranes) takes on average 15 minutes, (3) subsystems have dependencies to one another which may imply that shutting down one subsystem for an update requires shutting down and updating other subsystems as well.

Stakeholders. The key stakeholders of the update viewpoint are the architect(s), integrators, and the costumer system administrator. At Egemin there are a number of architects and senior engineers that are responsible for the different types of logistic systems. Update tasks are typically performed by the senior engineers.

Model Kinds. The viewpoint defines four model kinds that deal with the various stakeholder concerns. Models M1 and M2 allow stakeholders to browse the locations and structure of the deployed product (as-is) and the future product (to-be) respectively. These models deal with the variability, traceability and versioning concerns. Model M3 shows the update steps that integrators have to perform to update a deployed product, dealing with the availability and correctness concerns. Finally, model M4 shows inconsistencies of the product, dealing with the correctness concern.

The four model kinds are based on an integrated meta-model that defines the conceptual entities, their attributes and the relationships that comprise the vocabulary of the model kinds. For a detailed discussion of the individual meta-models of the four model kinds, we refer the interested reader to [22]. Defining an integrated meta-

Table 1: Update viewpoint for SPL

Name	Viewpoint for online updates of SPL products
Overview	The viewpoint deals with the main stakeholder concerns related to online updates of SPL, (i.e., variability, versioning, traceability, availability, and correctness) and defines models for updating deployed SPL products and capturing decisions pertaining to the product updates. The models show the relevant architectural information to the stakeholders dealing with their update concerns. The decisions delineate the policies and mechanisms to correctly update deployed products with minimal service interruption, including policies and mechanisms for managing variants of assets, versions of the products, assets, and the resources that correspond to the assets.
Concerns	*C1 - Variability*: Which are the core assets and product-specific assets that are currently installed/should be present in the system after the update? Is the combination of assets a valid one? *C2 - Traceability*: On which locations are the assets deployed? Which assets map to the installation bundles? Which resources map to which assets? Which development teams have the ownership of the assets? *C3 - Versioning*: Which are the versions of the core assets and product-specific assets that are currently installed? Which are the versions of the deployed resources for each asset? Which are the versions of the assets and resources that should be present for each asset after the update? *C4 - Availability*: How to perform the update with minimal interruption of the system? Which processes are shut down and (re-)started during an update? *C5 - Correctness*: What is the procedure to perform a correct update? Which resources have to be replaced to perform a correct update? Which processes have to be shut down and (re-)started and when? Is the update correct? Which inconsistencies still reside in the as-is product?
Stakeholders	*Architect(s):* prepare the update (primary interested in concerns C1 to C4). *Integrators:* perform the update (C1 to C5) *Customer admin:* maintains and operates the product (C1, C3, C4).
Model Kinds	*M1 - As-Is Product Deployment (deals with concerns C1, C2, C3):* A browsable model of the current product that shows locations, assets, owners, asset constraints, deployed resources, and resource dependencies. *M2 - To-Be Product Deployment (deals with concerns C1, C2, C3):* A browsable model of the future version of the product that includes installation bundles, assets, owners, asset constraints, resources, and resource dependencies. *M3 - Update Procedure (deals with concerns C4, C5):* A model showing the activities to perform the update, such as the resources that must be replaced/removed for the update, and the processes that have to be shut down and restarted during the update. M3 shows the outcome of Analysis A1 (see below). *M4 - Update Inconsistencies (deals with concern C5):* A model showing the inconsistencies in the updated product. M4 shows the outcome of Analysis A2.
Meta-Model	 Process; runs on; Location; is deployed on; SPL; contains; is based on; Product version; Asset Base; uses; is deployable on; has to be installed on; starts; contains deployable; Installation Bundle; can be deployed with; consists of; contains; refers to; Resource Dependency; has; Resource version; corresponds to; Asset version; belongs to; involves; has; Assembly; Database; Config file; Owner; Asset Constraint *Key:* UML; dotted elements are examples *Modeling elements per model kind*: M1: Product, Asset, Owner, Asset Constraint, Location, Process, Resource, Resource Dependency M2: Product, Asset, Owner, Asset Constraint, Installation Bundle, Resource, Resource Dependency M3: Product, Asset, Location, Process, Installation Bundle, Resource M4: Product, Asset, Asset Constraint, Owner, Location, Resource, Resource Dependency
Analyses	*A1 - Identify differences between as-is and to-be product:* - Resources to be added (present in to-be, absent in as-is) - Resources to be removed (present in as-is, absent in to-be) - Affected processes (to shut down and (re-)start) *A2 - Identify inconsistencies of the as-is and to-be product:* - Violations of asset constraints - Dependencies between resources that cannot be resolved - The presence of multiple versions of the same asset/resource in the product - An incomplete/inconsistent set of resources for a particular asset

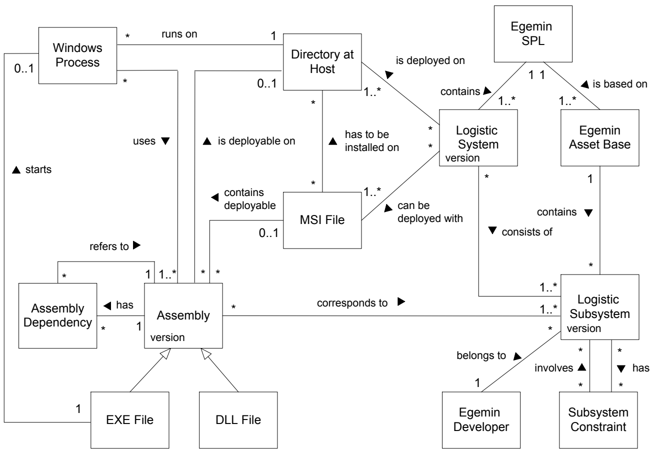


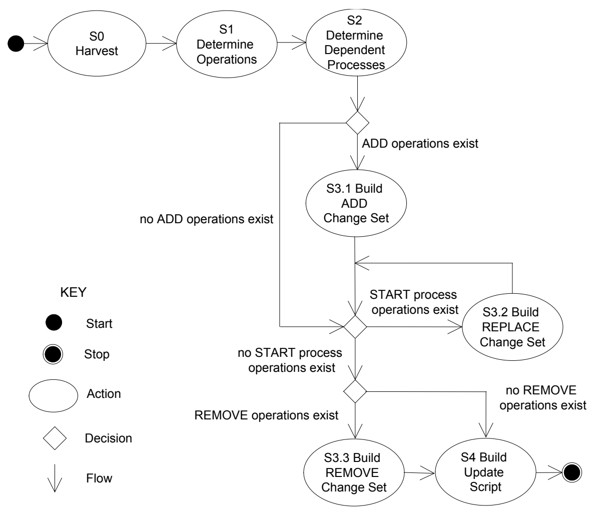
Figure 2: Integrated meta-model customized for Egemin

model has three motivations. First, the as-is product deployment model (M1) and to-be product deployment model (M2) are strongly related with several common concepts. Second, the analysis of the as-is and to-be product to determine the required update steps (M3) and the update inconsistencies (M4) are based on M1 and M2. Third, the integrated meta-model offers the basis for an architectural repository that we use to harvest the architectural relevant information from which the models are derived (we further discuss this in section 4). Note that the integrated meta-model includes the relations between elements of different models (i.e. model correspondences in terms of ISO/IEC 42010 [12]).

We briefly explain the main concepts of the meta-model in general. Then, we illustrate the customized concepts for Egemin's SPL. A *SPL* contains a set of *products*. A product has a version. Products consist of *assets* that are maintained in the SPL *asset base*. An asset has a version and an *owner*. The assets may have *asset constraints* that constrain the compositions of assets supported by the SPL. Examples of constraints are an asset *requires* another asset or set of assets, an asset *excludes* another asset, etc. A product can already be deployed on a set of *locations* (i.e., a deployed as-is product), or it can be ready for deployment with a set of *installation bundles* that have to be installed on locations (deployable to-be product). Each location on which a product is deployed contains a set of *resources* that correspond to particular assets. A resource has a version. Example resources are an assembly, a database, and a config file. A resource may have *resource dependencies* to other resources. A deployed assembly that is executable can be started as a *process* that runs on a location. A process uses one or more resources.

Figure 2 shows how the meta-model is customized for Egemin's SPL. Product is a logistic system, owner is an Egemin developer, asset is a logistic subsystem (E'wms, E'tricc, etc.), asset constraint is a constraint of a logistic subsystem to other logistic subsystems (e.g., the combination of E'tricc and E'can requires the installation of E'wms, or a particular version of E'tricc excludes particular versions of the logistic platform), installation bundle is a MSI file (Microsoft Installer), location is a directory on a host, process is a windows process, and assembly is a DLL file (Dynamic Link Library) or an EXE file (Executable).

Analysis. Finally, the update viewpoint defines two types of analysis. We give an overview of the algorithms of both analysis. For a detailed discussion, we refer the reader to [23]. The first analysis determines the differences between the as-is and to-be product determining the update procedure model (M3). Figure 3 shows an overview of the algorithm to generate the update procedure model.



S0 Harvest
S1 Determine Operations
S2 Determine Dependent Processes
ADD operations exist
no ADD operations exist
S3.1 Build ADD Change Set
START process operations exist
S3.2 Build REPLACE Change Set
no START process operations exist
no REMOVE operations exist
REMOVE operations exist
S3.3 Build REMOVE Change Set
S4 Build Update Script
KEY
Start
Stop
Action
Decision
Flow

Figure 3: Overview of the update script generation algorithm.

The algorithm consists of four main steps (S1 - S4). Step S0 is a preparatory step in which architecture knowledge is harvested from the deployed system and the installation bundles (we explain the practical aspects of harvesting in the next section).

In the first step, S1, the resource operations are determined (ADD, REPLACE, REMOVE). The set of operations is derived from a comparison of knowledge of the as-is and to-be system. For example, if there is a resource in the to-be system that is not in the as-is, a new ADD operation is defined for this resource. In the second step, S2, all the STOP and START operations for processes are determined. The set of affected processes consists of all the processes with a direct or indirect dependency on a resource for which a REMOVE or REPLACE operation exists. Determining the minimal set of resources that have to be updated not only ensures a minimal downtime of services of the updated product, it also minimizes the recertification costs for updated resources.

The third step, S3, consists of three sub-steps: S3.1 to S3.3. In each sub-step a particular *change set* is computed. A change set consists of a sequence of update operations that migrate the system from one consistent state to another. In step S3.1, the change set of ADD operations is determined. The ADD operations can be executed without shutting down any process of the deployed system. Next, in step S3.2, the change sets with REPLACE operations are determined. Each change set consists of the subset of REPLACE operations that are applicable to a set of resources that have dependencies with one another. The REPLACE operations will be preceded by STOP operations and followed by START operations for all processes with dependencies to any of the resources that have to be replaced. The services associated with the interrupted processes are not available during the execution of the REPLACE change sets. Finally, in step S3.3, the change set of REMOVE operations is determined. The REMOVE operations will be preceded by STOP operations for all processes that have to be terminated, i.e. the processes with only dependencies to resources that have to be removed. As such, REMOVE operations do not require a shutdown of active services of the deployed system.

In the final step, S4, the update script is built using the various change sets. The update script defines the sequence of update steps.

We illustrate the different steps of the algorithm with a simplified update scenario of the logistic system shown in Figure 1. The initial setting of the scenario is shown Figure 4.

The figure shows the resources and processes with there dependencies deployed on two hosts of the system. In reality, several hundreds of resources are deployed on each host of a logistic system. The installation bundles with the resources of a new version of E'car and the logistic platform are also shown. The arrows indicate on which locations the installation bundles have to be deployed.

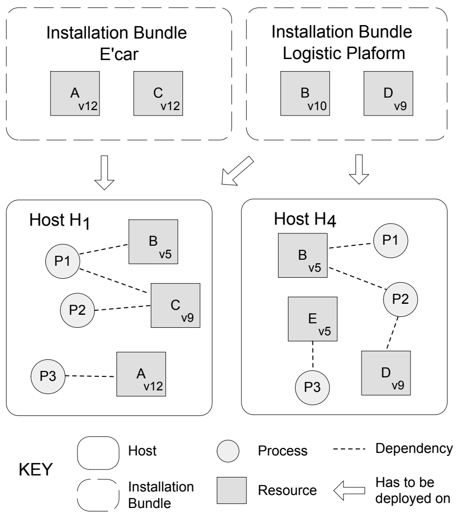


Figure 4: Example scenario to illustrate the update procedure

Listing 1 shows the update script for the example scenario. The update operations are grouped in three blocks that represent the different sets of resource operations. The first block contains a single ADD operation for resource D at host H_1. The second block contains REPLACE operations for resources B and C at host H_1, and resources B and D at host H_4. These operations are proceeded by STOP operations and followed by START operations for the dependent processes P1 and P2 at both hosts. The third block contains a single REMOVE operation for resource E at host H_4 with a STOP operation for process P3 that has to be terminated.

```
UPDATE SCRIPT:

  ADD change set
      ADD(Dv10,H1)

  REPLACE change set
      STOP(P1,H1)              STOP(P2,H1)
      STOP(P1,H4)              STOP(P2,H4)
      REPLACE(Cv9,Cv12,H1)     REPLACE(Bv5,Bv10,H1)
      REPLACE(Bv5,Bv10,H4)     REPLACE(Dv5,Dv10,H4)
      START(P1,H1)             START(P2,H1)
      START(P1,H4)             START(P2,H4)

  REMOVE change set
      STOP(P3,H1)
      REMOVE(Ev5,H4)
```

Listing 1: Update script for the example scenario

The second analysis identifies inconsistencies of the updated product (M4). Inconsistencies may refer to conflicting versions of assets and resources of assets and errors with respect to the resources of each asset, including errors related to versioning and dependencies. Figure 5 shows an overview of the algorithm to determine the update inconsistencies model.

After the preparatory harvesting step S0, the algorithm determines asset and resource inconsistencies in parallel. In the first step, S1,

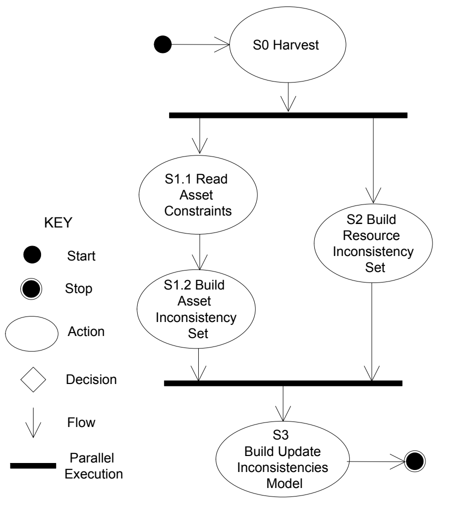

S0 Harvest
S1.1 Read Asset Constraints
S2 Build Resource Inconsistency Set
S1.2 Build Asset Inconsistency Set
S3 Build Update Inconsistencies Model
KEY
Start
Stop
Action
Decision
Flow
Parallel Execution

Figure 5: Overview of the algorithm to determine update inconsistencies.

the asset inconsistencies are computed. First, the asset constraints are read (S1.1). Next, violations of asset constraints are identified by analyzing the assets of the deployed product and checking whether they violate any of the asset constraints (S1.2). For example, if one asset excludes another asset, but both are present in the product, this is a violation. Violations are collected in the asset inconsistency set.

In the second step, S2, resource inconsistencies are computed. A distinction is made between two types of inconsistencies. Resource dependency inconsistencies are identified by resolving each resource dependency. Each identified failure leads to a resource dependency inconsistency that is added to the resource inconsistency set. Resource version inconsistencies are identified by comparing the available versions of each asset and resource of a product. In case multiple versions co-exist there is a resource version inconsistency. As an example, assume that an integrator performs the update steps as shown in Listing 1, but forgets to deploy the updated logistic platform at host H_1. This will result in two inconsistencies as shown in Listing 2.

INCONSISTENCIES:

(R1) $if\ (E\text{'}car_{v12}, H_i)\ then\ (LP_{v10}, H_i)$
$\neg$ R1 : $i = 1$

(R2) $if\ (R_{vx}, H_i) \wedge (R_{vy}, H_j)\ then\ x = y$
$\neg$ R2 : $(R_{v5}, H_1) \wedge (R_{v10}, H_4)$

Listing 2: Inconsistencies after incorrect update for the scenario

The first rule expresses an asset inconsistency between E'car and the logistic platform. In particular, it states that the updated E'car software with version 12 requires the deployment of the logistic platform of version 10 at the same host. This constraint is violated for host H_1. The second rule expresses a resource version inconsistency that states that only the same version of each resource can be deployed in the system. This constraint is violated for resources B at host H_1 (version 5) and host H_4 (version 10). Deployment of the installation bundle of the logistic platform at host H_1 will resolve the inconsistencies.

4. FRAMEWORK

The update viewpoint is supported by a framework built in Java.

This framework provides a reusable and extensible infrastructure for updating deployed products of a SPL. The key functionalities supported by the framework are: (1) harvesting relevant architecture knowledge; (2) storing the harvested knowledge, (3) querying and analyzing architectural knowledge, (4) visualizing the architectural models in a comprehensive way for the stakeholders.

Figure 6 shows the primary components of the framework that we developed to support online updates of SPL products. We discuss each of the components in more detail and indicate how we have tailored the framework for Egemin's SPL.

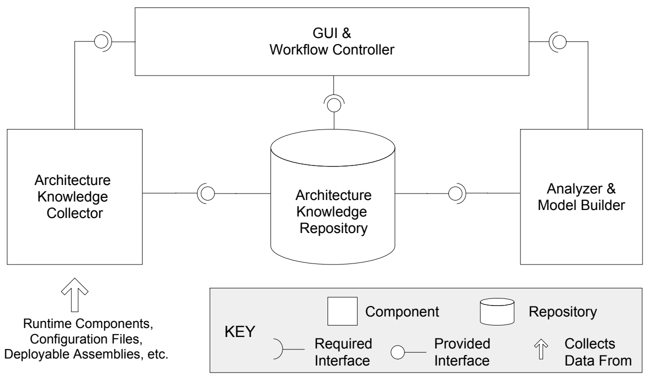

GUI &
Workflow Controller
Architecture Knowledge Collector
Architecture Knowledge Repository
Analyzer & Model Builder
Runtime Components, Configuration Files, Deployable Assemblies, etc.
KEY
Component
Repository
Required Interface
Provided Interface
Collects Data From

Figure 6: General overview of the supporting framework

4.1 GUI and Workflow Controller

System integrators interact with the framework through a standard GUI (Graphical User Interface). The GUI allows integrators to configure the update setting, harvest architecture knowledge, build models (as-is, to-be, update procedure, and update inconsistencies), and browse the models. To configure the update setting, the integrator has to import a specification of the different locations where the system is deployed, as well as the location of the installation bundles. To execute the actions of the integrator, the GUI triggers the workflow controller which activates the architecture knowledge collector, architecture knowledge repository, and the analyzer & model builder in the right sequence. The typical workflow is to (1) prepare the architecture repository for new input; (2) trigger the harvesters to populate the repository; (3) trigger the analyzer & model builder to query the repository and perform the necessary analysis, and (4) update the visual models.

4.2 Harvesters

The architecture knowledge collector comprises a number of plug-in harvester components that perform the actual knowledge gathering. Architecture knowledge can be harvested from run-time system components, resource files, system configurations, etc. The knowledge collector provides a control interface that allows the workflow controller to invoke the harvesting process and to check its progress. Harvested knowledge in stored in the architecture knowledge repository. Internally, a harvester component can use any technology to gather architecture knowledge. Three example harvesters that we used to harvest knowledge for Egemin's SPL are:

- Assembly Harvester: gathers knowledge about the assemblies of the deployed system per location, including assemblies' version and compile time dependencies. Technically, this harvester includes a C# program based on the Mono.Cecil library[4] that supports inspection of programs and libraries.

[4]http://www.mono-project.com/Cecil

- Config File Harvester: gathers configuration knowledge about dynamically loaded assemblies and the run-time dependencies between assemblies. Two examples of configuration files harvested this way are the .Net App.config file and the ProfileCatalog.xml for SmartClients in .Net.
- MSI files Harvester: gathers knowledge about the to-be deployed product. This harvester uses the two previous harvesters to collect knowledge of the assemblies from a MSI file, including versions and dependencies.

4.3 Architecture Knowledge Repository

The architecture knowledge collected by the harvesters is used to populate the architecture knowledge repository. The repository stores architecture knowledge that complies to the integrated meta-model described in the update viewpoint. We used the Eclipse Modeling Framework (EMF) as a basis for implementing the repository. EMF supports specifying a meta-model, and generating a Java implementation along with set of adapter classes that enable basic viewing and command-based editing. The meta-model specification can be customized to suit the need of a particular SPL. For Egemin, we customized the meta-model as shown in Fig. 2.

4.4 Analyzer & Model Builder

Finally, the analyzer & model builder queries the architecture repository and generates on demand the requested architecture models. The analyzer & model builder consists of a query engine and a set of model components, one for each model kind.

Query Engine. The query engine offers a customizable application programming interface (API) that enables harvesting knowledge from the architectural repository for the model components. Internally the query engine relies on the EMF Query library to perform basic queries on the architecture repository. For the as-is and to-be deployment models, the API offers methods that uses simple queries to retrieve knowledge from the architecture knowledge repository. For example, the following example method returns all the assemblies that depend on a given assembly:

```
Collection<Assembly>
    findDependentAssemblies(Assembly assembly)
```

For the update procedure model and the update inconsistencies model, the API offers methods that use series of queries retrieve knowledge and perform the necessary analysis to generate the knowledge required for representing the models. For example, the following method computes all the assemblies that exist in the to-be system but that are not present in the as-is system for a given host:

```
Collection<Assembly>
    computeToADDAssemblies(Host host)
```

The query engine supports the definition of new built-in queries.

Model Components. The model components take care of the visualization of the architecture models in a way convenient for the stakeholders. As the framework supports interactive architectural models, the model component also handles model-specific user interactions with the system, such as browsing the resources deployed at a particular location and the dependencies of the resources.

Model components are designed according to the Model-View-Controller (MVC) architectural pattern. The *content provider* component (i.e. the model of the MVC pattern) manages the interactions with the query engine and notifies the *viewer* component when that information changes. The viewer renders the content into a form suitable for interaction with the stakeholders, typically a user interface element. The *controller* is responsible for handling user actions by instructing the content provider and the viewer to execute the actions. Technically, the user interface is built on top of Eclipse's Standard Widget Toolkit (SWT). The MVC pattern enables adding new model components or altering the content and the presentation of an existing model.

Figure 7 shows a snapshot of a update procedure model for one of the products of Egemin's SPL. The box top left shows the different locations on which the product is deployed. The box on the right hand side shows the installation bundles (product installers) that have to be deployed on the selected location. The box at the bottom shows the update script that resulted from the analysis. The update script shows the subsequent update steps the integrator has to perform to realize the update.

5. EVALUATION

We performed an empirical evaluation of the approach for updating deployed SPL products in the context of Egemin's SPL. The focus of the empirical evaluation is on two key concerns of online updates of SPL: correctness and availability. In addition we evaluated a number of properties related to the use of the framework. The study is based on a comparison between Egemin's current practice to update products of their SPL (denoted as the baseline approach) and the use of the new proposed approach (denoted as architectural approach). In this section, we summarize the evaluation setup and the most important results of the evaluation. For a detailed description of the empirical evaluation, we refer the interested reader to [13].

5.1 Hypotheses

The experiment is used to empirically verify the following hypothesizes:

H1. The architectural approach improves the correctness of updating a product. More specifically, integrators will make less errors when updating a product with the architectural approach.

H2. The architectural approach improves the availability of the system during a product update. More specifically, using the architectural approach requires less shutdowns of processes during a product update.

To verify the hypothesis, professionals were asked to perform a number of realistic update scenarios to existing logistic systems in a controlled setting. Additionally, we used questionnaires to probe the integrators' confidence in the correctness of the updates.

5.2 Empirical procedure

Technically, the evaluation is a supervised experiment with a paired comparison design [24]. In total 17 professionals served as subjects. The experience of the test persons with updating logistic systems is mixed. To reduce the learning curve, each test person started with two introductory scenarios, using both the baseline approach and the architectural approach. Subsequently, the test persons performed four update scenarios in a random order, with a randomly selected approach. The test persons were asked to perform the updates correctly and with minimal process shutdowns.

To ensure objectivity, test persons cannot perform the same update scenario with the two approaches. Therefore, we defined two sets of pairwise scenarios with comparable complexity. One set contains two simple scenarios requiring an update to a single logistic system (both the client/server). The other set contains two complex scenarios, where an update propagates to multiple servers and clients. In total 68 updates were performed (#test persons $\times$ #scenarios), evenly distributed over both approaches.

Each update scenario was monitored by a supervisor who took notes of particular events. All the user actions were logged using a set of tools. After each scenario, the test person filled out a questionnaire assessing his/her level of confidence in the correctness of the update. After performing all scenarios, the test person filled out a second questionnaire with questions concerning his experience, the overall assessment of both approaches, etc.

5.3 Update Scenarios

We give a brief description of one of the update scenarios as illustration. The interested reader is referred to [13] for a detailed description of all the scenarios.

Setting. The update is performed on the Kimberly Clark (KC) project. The system is installed in the standard location (Program Files/Egemin) and the system is operational. The following subsystems are running:

- E'tricc server (control software AGV system)
- E'wcs server (warehouse management system)
- E'pia server (logistic platform)
- Shell (clients)

Update Scenario. A new version of the EPIA platform is available and should be installed on the KC system. The new installation files of the KC system can be found in a directory called InstallationFiles on the Desktop. This directory contains the installation files of the complete updated system. It contains both the compressed MSI files and a directory with the unpacked installation files.

Assignment. Perform an update of the KC system for the above described scenario. The customer expects the update to be performed with minimal interruption of services.

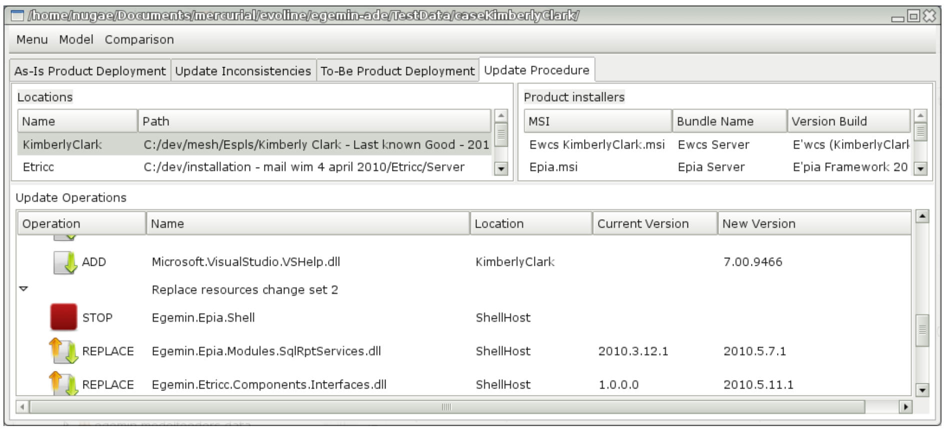
5.4 Summary of Results

The results of the study are based on a statistical analysis with a 95% confidence interval.

Correctness. We compared the number of updates that were performed correctly according to the requirements of the update scenarios. With the baseline approach 71% of the simple scenarios where performed correctly and only 18% of the complex scenarios. With the architectural approach all scenarios were performed correctly (100% correctness). For the complex scenarios a statistical analysis based on the Wilcoxon-test revealed a significant difference in correctness between both approaches. For the simple scenarios there was insufficient evidence.

Availability. We compared the number of unnecessary process shut downs which cause interrupts of logistic subsystems that were not needed for the update. With the baseline approach, in 17% of the simple scenarios and 33% of the complex scenarios at least one process was shut down unnecessarily. With the architectural approach in 6% of the complex scenarios at least one process was shut down. Note that these processes were shut down due to mistakes of integrators. In the simple scenarios, no unnecessary processes were shut down. Statistical analysis showed significant difference between both approaches for the availability results.

Confidence. All test persons confirmed that the architectural approach gives them more confidence to evolve logistic systems. However, for the baseline approach we found no correlation between the reported confidence of a test person in the correctness of the update and the actual correctness. This means that test persons are often unaware that the updated logistic system contains errors. With the architectural approach, the test persons confirmed that they were confident that the update scenarios were performed correctly.



Locations

Name	Path
KimberlyClark	C:/dev/mesh/Espls/Kimberly Clark - Last known Good - 201
Etricc	C:/dev/installation - mail wim 4 april 2010/Etricc/Server

Product installers

MSI	Bundle Name	Version Build
Ewcs KimberlyClark.msi	Ewcs Server	E'wcs (KimberlyClark
Epia.msi	Epia Server	E'pia Framework 20

Update Operations

Operation	Name	Location	Current Version	New Version
ADD	Microsoft.VisualStudio.VSHelp.dll	KimberlyClark		7.00.9466
	Replace resources change set 2			
STOP	Egemin.Epia.Shell	ShellHost		
REPLACE	Egemin.Epia.Modules.SqlRptServices.dll	ShellHost	2010.3.12.1	2010.5.7.1
REPLACE	Egemin.Etricc.Components.Interfaces.dll	ShellHost	1.0.0.0	2010.5.11.1

Figure 7: Example of an update procedure model.

5.5 Threats to validity

We briefly discuss two threats to the validity of the experiment. First, the experiments were performed in a controlled lab setting. People may react differently when no real customers are involved or faulty updates to logistic machines cannot cause real damage. We anticipated this threat by explaining the test persons that minimal shutdowns and correctness were key requirements of the updates.

Second, the experiment did not allow the test person to interact with other people. In practice, integrators may contact colleagues during updates. Note that both approaches received an equal treatment. Moreover, it is known that fixing errors by calling colleagues only helps in a small number of cases.

6. RELATED WORK

The research presented in this paper is related to three extensively studied fields in software engineering: software evolution, architecture reconstruction, and architecture viewpoints. Here, we mainly focus on work related to these three fields in the context of SPL.

Software Evolution. Several authors have pointed to the complexity of SPL evolution. Svahnberg and Bosch [19] argue that the higher level of interdependency between the various software assets makes evolution in SPL a more complex process. Pohl at al. [16] state that stakeholders of a SPL are faced not only with evolution over time, but also with the existence of different variants at the same time. The authors put forward the consistent integration of changes in SPL as an open research challenge in SPL engineering. Hendrickson and van der Hoek [10] point to the complexity associated with the typical mismatch between architectural variability and the actual variability. Ajila and Kaba [1] discuss an evolution process for SPL that supports local and complex (i.e. inter-project) evolutions. The authors also present a number of supporting mechanisms for evolving SPL, including mechanisms to check consistency during maintenance at different levels of granularity (architecture, product line, product). Tools like SELECTA [9] which uses composition of meta-models, and pure::variants [5] (industrial tool) help with the variability space evolution. Whereas the existing approaches focus on particular concerns with respect to different aspects of SPL evolution, our approach is targeted at runtime updates of SPL products. Our work contributes with a unified view on SPL updates that covers multiple concerns which are captured in an update viewpoint.

Architecture Reconstruction. In a recent article, Ducasse and Pollet [8] provide a general overview of the state of the art in software architecture reconstruction. When accurate architecture knowledge is lacking, SPL methods for SPL updates (and evolution in general) can benefit from the use of architecture reconstruction techniques.

Anquetil et al. [3] identify four orthogonal traceability dimensions in SPL (refinement, similarity, variability, and versioning). The authors propose a model-driven traceability framework based on the specification of a metamodel for a repository of traceability links. The approach enables retrieving and manipulating the various types of traceability links in SPL. [2] employ source code mining techniques to understand SPL evolution. Chen proposes a set of tools to assist maintainers with the co-evolution of product and product line architectures [6]. The tools support determining the difference between two selected (versions of) a product architecture, and merging the difference back into the original product line architecture.

Several general tools have been developed that support architecture reconstruction. These tools can be useful to support updates of SPL. ArchView [15] visualizes a software architecture that is built from the source code and historical information. The SEXTANT [18] tool enables retrieving architecture relevant information from multiple sources, and uses this information to discover dependencies between entities using the XQuery framework.

Comparing to the discussed approaches, our work derives as-is views from a heterogenous set of information sources of deployed products. We also extract to-be architectures, i.e. the valid installation bundles of updated products. Finally, our approach goes beyond identifying differences between as-is and to-be architectures by deriving the concrete tasks that are needed to evolve SPL products.

Architectural Viewpoints and Views. The research devoted to architectural viewpoints and views specific for SPL is quite limited. We discuss one representative approach from the domain of SPL and some more general relevant related work.

O'Brien [14] describes a process and a supporting workbench that was used to reconstruct architectural views of three small automotive motor systems. The author discusses how different architecture views and styles were extracted from the source code of the systems, supported with staff interviews.

Rozanski and Woods [17] describe an evolution *perspective*. An architecture perspective is defined as "a collection of activities, tactics, and guidelines that are used to ensure that a system exhibits a particular set of related quality properties that require consideration across a number of the system's architectural views." The scope of the evolution perspective is very broad and covers the following concerns: the magnitude of change, dimensions of change, likelihood of change, timescale for change, when to pay for change, development complexity, preservation of knowledge, reliability of change.

van Deursen et al. [21] propose an interesting approach for view reconstruction, called Symphony. Symphony comprises a two-stage process for reconstructing architecture views. The first stage produces a reconstruction strategy defining the views to reconstruct. The second stage of Symphony concerns the execution of the reconstruction strategy, during which the views are populated and the results are interpreted. The two stages have similarities with the view definition and update execution of our work.

Arias et al. [4] define an execution viewpoint to facilitate evolution of large and complex software-intensive systems. The authors obtain a set of as-is execution models by performing measurements on a running system. Based on those models, budgets for future designs can be expressed as to-be models. This work follows a similar idea as our work, however, the type of viewpoint is different.

The contribution of our work to architectural viewpoints is the definition of an update viewpoint for SPL that covers the important concerns and enables stakeholders to deal with the tradeoffs between the update concerns.

7. CONCLUSIONS

In this paper, we presented an architecture approach to support the updates of SPL products. Central in this approach is an update viewpoint that frames the multiple stakeholder concerns for updating SPL products, including variability, traceability, versioning, availability, and correctness. The fact that we used an architectural viewpoint is not by coincidence. Architecture viewpoints are an established approach to manage multiple concerns. The update viewpoint captures reusable architectural knowledge that can be customized for a concrete SPL. A customized update viewpoint offers the means for stakeholders to reason about their concrete concerns for updating the SPL. The proposed approach employs views as an instrument to support online updates of SPL products. The models of the update views generated by the supporting framework guide integrators by listing the concrete tasks they need to perform when upgrading deployed products and by showing inconsistencies when they fail to do so.

Considering multiple concerns at the same time inevitably implies tradeoffs. We applied our approach to support updates of SPL products at the granularity of assets and their corresponding resources. Consequently, consistency and correctness can only be guaranteed at the same level of granularity. However, applying the approach at a finer level of granularity increases complexity and costs.

Although we successfully applied the approach to Egemin's SPL, several research challenges remain. In particular, we plan to formally proof the correctness and availability properties of the proposed approach. We also plan to challenge the generality of the approach by applying it to a different domain with different technology.

8. REFERENCES

[1] S. Ajila and A. Kaba. Evolution support mechanisms for software product line process. *Journal of Systems and Software*, 81(10):1784–1801, 2008.

[2] S. A. Ajila and R. T. Dumitrescu. Experimental use of code delta, code churn, and rate of change to understand software product line evolution. *Journal of Systems and Software*, 80(1):74–91, 2007.

[3] N. Anquetil, U. Kulesza, R. Mitschke, A. Moreira, J.-C. Royer, A. Rummler, and A. Sousa. A model-driven traceability framework for software product lines. *Software and Systems Modeling*, 2010.

[4] T. Arias, P. America, and P. Avgeriou. Defining execution viewpoints for a large and complex software-intensive system. In *Joint Working Conf. WICSA/ECSA*, 2009.

[5] D. Beuche. Variant management with pure:: variants. *Pure-systems GmbH, Tech. Rep*, 2003.

[6] P. Chen, M. Critchlow, A. Garg, C. Van der Westhuizen, and A. van der Hoek. Differencing and merging within an evolving product line architecture. In *Software Product-Family Engineering, LNCS vol. 3014*. Springer, 2004.

[7] P. Clements and L. Northrop. *Software product lines*. Addison-Wesley Reading MA, 2001.

[8] S. Ducasse and D. Pollet. Software architecture reconstruction: A process-oriented taxonomy. *IEEE Trans. on Software Engineering*, 35(4):573 –591, 2009.

[9] J. Estublier, I. A. Dieng, and T. Leveque. Software product line evolution: the selecta system. In *Product Line Approaches in Software Engineering*. ACM, 2010.

[10] S. A. Hendrickson and A. van der Hoek. Modeling product line architectures through change sets and relationships. In *ICSE 2007*. IEEE Computer Society.

[11] R. Hilliard. A trust viewpoint. *Technical Report*, 2009. http://mysite.verizon.net/rfh2/writings/hilliard-TrustVP-r1.pdf.

[12] ISO/IEC. Systems and software engineering - architecture description. *ISO/IEC standard, draft D8*, August 2010.

[13] B. Michalik, N. Boucke, D. Weyns, and A. Helleboogh. *Empirical Evaluation of EvoLine*. Katholieke Universiteit Leuven, 2011. TR. Available via http://people.cs.kuleuven.be/danny.weyns/EvoLineEvaluation.pdf.

[14] W. O'Brien. Architecture reconstruction to support a product line effort: Case study. In *TR CMU/SEI-2001-TN-015*. Software Engineering Institute, 2001.

[15] M. Pinzger. Archview-analyzing evolutionary aspects of complex software systems. *PhD, Vienna University*, 2005.

[16] K. Pohl, G. Böckle, and F. Van Der Linden. *Software product line engineering: foundations, principles, and techniques*. Springer-Verlag New York Inc, 2005.

[17] N. Rozanski and E. Woods. *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives*. Addison-Wesley Professional, 2005.

[18] T. Schafer, M. Eichberg, M. Haupt, and M. Mezini. The sextant software exploration tool. *Software Engineering, IEEE Transactions on*, 32(9):753 –768, sept. 2006.

[19] M. Svahnberg and J. Bosch. Evolution in software product lines: Two cases. *Journal of Software Maintenance*, 11(6):391–422, 1999.

[20] F. Van Der Linden, F. van der Linden, K. Schmid, and E. Rommes. *Software product lines in action: the best industrial practice in product line engineering*. Springer-Verlag New York Inc, 2007.

[21] A. Van Deursen, C. Hofmeister, R. Koschke, L. Moonen, and C. Riva. Symphony: View-driven software architecture reconstruction. In *Fourth Working IEEE/IFIP Conf. on Software Architecture*, 2004.

[22] D. Weyns, B. Michalik, N. Boucke, and A. Helleboogh. *Viewpoint for Online Evolution of Software Product Lines*. Katholieke Universiteit Leuven, 2011. TR. K.U.Leuven http://people.cs.kuleuven.be/danny.weyns/EvoLineViewpoint.pdf.

[23] D. Weyns, B. Michalik, A. Helleboogh, and N. Boucke. Codifying architecture knowledge to support online evolution of software product lines. In *Sixth Workshop on SHAring and Reusing architectural Knowledge, SHARK*, 2011.

[24] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in software engineering: an introduction*. Kluwer, 2000.