Codifying Architecture Knowledge to Support Online Evolution of Software Product Lines

Danny Weyns and Bartosz Michalik
DistriNet Labs, Katholieke Universiteit Leuven
Celestijnenlaan 200A, 3001 Leuven Belgium
{danny.weyns,bartosz.michalik}@cs.kuleuven.be

ABSTRACT

A company's architecture knowledge is often personalized across specific people that share experience and knowledge in the field. However, this knowledge may be important for other stakeholders. Omitting the codification of the architecture knowledge may result in ad-hoc practices, which is particularly relevant for software evolution. In a collaboration with Egemin, an industrial manufacturer of logistic systems, we faced the problem with a lack of codified architecture knowledge in the context of the evolution of a software product line (SPL). In particular, maintainers lack the architecture knowledge that is needed to perform the evolution tasks of deployed products correctly and efficiently. Ad-hoc updates increase costs and harm the company's reputation. To address this problem, we developed an automated approach for evolving deployed systems of a SPL. Central in this approach are (1) a meta-model that codifies the architecture knowledge required to support evolution of a SPL, and (2) and algorithm that uses the architecture knowledge harvested from a deployed system based on the meta-model to generate the list of tasks maintainers have to perform to evolve the system. Evaluation of the approach demonstrates a significant improvement of the quality of system updates with respect to the correct execution of updates and the availability of services during the updates.

Categories and Subject Descriptors

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

General Terms

on-line, evolution, SPL, software produt line

Keywords

Architecture knowledge, software produt line, evolution

1. INTRODUCTION

Maintaining architecture knowledge in the face of system evolution is a difficult task [5]. A company's knowledge management strategy captures which architecture knowledge is *codified* in explicit accessible models, and which knowledge is *personalized* across specific people that share their experience and knowledge in the field [4]. However, personalized architecture knowledge may be important for other stakeholders. Omitting the codification of this architecture knowledge may result in ad-hoc practices. Lacking of important architecture knowledge is particularly relevant for maintainers that have to deal with software evolution. The problem may be even harder in the context of a Software Product Line (SPL). A SPL comprises a set of software systems (products) that share a common, managed set of features [7]. Stakeholders of SPL are faced not only with the evolution over time, but also with the existence of different products at the same time [15].

In a recent R&D project in collaboration with Egemin[1], an industrial manufacturer of logistic systems, we faced the problem with a lack of codified architecture knowledge in the context of the evolution of a SPL of logistic systems. Logistic systems include a warehouse management system and control software for one or more transportation subsystems, such as automated guided vehicles, cranes, and conveyors. During their lifespan, logistic systems obviously have to evolve. Our particular focus of evolution is on the execution of the concrete update tasks maintainers have to perform to evolve deployed systems. E.g., the software of a subsystem needs to be upgraded to improve performance, or a customer introduces a new conveyor belt and its control software needs to be integrated with the existing logistic system. Architects and maintainers at Egemin face various problems with such evolution scenarios.

Most of the architecture knowledge of Egemin's SPL is personalized across a small group of people. In addition, the logistic systems comprise a lot of legacy software components, which documentation is often incomplete or outdated. As a result, maintainers lack the architecture knowledge that is needed to perform the evolution tasks efficiently and correctly. Faulty updates increase maintenance costs, harm the company's reputation, or even worse, they may cause serious damage to industrial installations.

To address this problem, we developed an automated approach for evolving deployed systems of a SPL. Central in this approach are (1) an architecture meta-model that codifies the architecture knowledge required to to support evolution of a SPL, and (2) and algorithm that uses the architecture knowledge harvested from a deployed system based on the meta-model to generate the list of tasks maintainers have to perform to evolve the system. Note that we consider a specific form of architecture knowledge in this paper which includes knowledge about the course grained structures of SPL products and their deployment, inconsistencies of installed products, and the update tasks required to evolve a deployed product. We developed a supporting tool to assist maintainers with the evolution tasks of deployed systems. Evaluation of the approach demonstrates a significant improvement of quality of system updates with respect to correctness and availability of services during the updates.

The reminder of this paper is structured as follows. Section 2 explains the problems with architecture knowledge of Egemin SPL evolution in more detail. In Section 3, we present the meta-model

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

SHARK '11, May 24, 2011, Waikiki, Honolulu, HI, USA

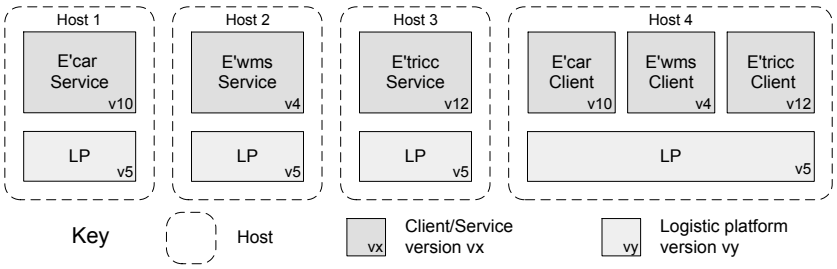

Figure 1: Typical configuration of a logistic system

that codifies the architecture knowledge required to support evolution of a SPL. Section 4, the central part of the paper, explains in detail the algorithm that uses the harvested knowledge based on the meta-model to generate the list of tasks a maintainer has to perform to evolve a deployed system. In Section 5, we briefly report on the evaluation of the approach. Finally, we present related work in Section 6, and draw conclusions in Section 7.

2. PROBLEMS WITH SPL EVOLUTION

Figure 1 shows a typical configuration of a logistic system. The system includes a warehouse management system (E'wms® - Egemin warehouse management system) that is responsible for managing tasks in the system, and control software for automated guided vehicles (E'tricc® - Egemin transport intelligent control center) and cranes (E'car® - Egemin crane automatic storage and retrieval system). These logistic subsystems are built from a common set of components that can be customized and composed to the needs of the customers. 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, whereas the client offers a graphical interface to access the service. The distributed logistic platform is a component framework developed by Egemin that provides common middleware services for logistic systems such as support for system configuration, communication, persistence, online updates, etc.

Egemin's SPL comprises over 200 deployed logistic systems. During their lifespan (typically 10+ years), logistic systems obviously have to evolve. An example of an evolution scenario for the configuration shown in Figure 1 could be an upgrade the E'car service from version v_{10} to version v_{12} which includes a new stacking algorithm that reduces the average retrieval time by 12%.

The goal of a system evolution is to migrate the deployed logistic system (*as-is*) to a new version (*to-be*). The new version is available as a set of installation bundles that comprise all the resources (e.g. executables or libraries) of the updated system together with a specification of the target location of each resource. Two important requirements that have to be satisfied during system evolution are:

R1. *Correctness*: The maintainer should perform a correct sequence of update steps to bring the system from the as-is to the to-be version. Update steps include adding/removing/replacing resources and stopping/starting processes. Restarting an incorrect configuration may compromise the consistency of the logistic system.

R2. *Availability*: The maintainer should minimize the total shutdown time of the various logistic subsystems. Logistic systems typically have to operate 24/7. Interruption of its services is costly and should be kept minimal.

To perform an update, maintainers need detailed knowledge of the deployed system and the installation bundles. Required knowledge of the deployed system includes the set of running subsystems each with its resources, the location and version of each resource, the dependencies between resources, the set of running processes, etc. Required knowledge of the installation bundles include the set of subsystems that have to be deployed after the update each with its resources, the target location and version of each resource, the dependencies between resources, the set of processes that have to run after the update, etc. Based on this knowledge, a correct sequence of update steps have to be derived that ensure a correct update of the deployed system with minimal interruption of its services.

Unfortunately, this detailed knowledge is not codified in explicit models. Since different subsystems are developed by different teams in the company, the personalized knowledge is spread across a number of people. Moreover, the deployed logistic systems comprise a lot of legacy software components for which documentation is often incomplete or outdated. The lack of documentation and detailed architecture knowledge leads to the ad-hoc update practices which are inefficient and error prone.

Together with the architects and maintainers at Egemin, we identified three models that codify the architecture knowledge required to evolve deployed logistic systems (products) of the SPL:

M1 *As-Is Product Deployment Model:* A model of the current product that shows locations, deployed subsystems with their resources, resource dependencies, and the running processes.

M2 *To-Be Product Deployment Model:* A model of the future version of the product which is available as a set of installation bundles. This model shows the target set of subsystems with their resources, target locations of the resources, resource dependencies, and processes that have to run after the update.

M3 *Update Procedure Model:* A model that describes the sequence of update steps (resource changes and process manipulations) that need to be performed to evolve the deployed logistic system from the as-is to the to-be version.

In summary, the problem we have to solve is: (1) to codify the architecture knowledge of models M1 and M2, (2) to codify the architecture knowledge of model M3 by analyzing models M1 and M2, and (3) to develop support for automatic reconstruction of this knowledge for a particular evolution setting to enable maintainers to evolve deployed systems. The focus of this paper is on the codification of the architecture knowledge, i.e. subproblems (1) and (2).

3. META-MODEL

A meta-model presents the conceptual entities, their attributes and the relationships that comprise the vocabulary of a type of model [14]. We use meta-models to codify the architecture knowledge required to evolve deployed SPL products. We follow a bottom up approach which reflects the way we defined the models in practice. We start by introducing as-is and to-be product deployment meta-models for Egemin's logistic systems. Then we integrate these models in a single meta-model. This first set of models was defined in close interaction with the key stakeholders at Egemin. The

model concepts are specific to the company's technological context (.NET). Next, we generalize the integrated meta-model based on a thorough literature study, including [2, 3, 15, 18]. The generalized meta-model offers reusable architecture knowledge to support the evolution of deployed SPL products, independently of any particular technology. The section concludes with a brief note on the a supporting framework we developed to support harvesting of architecture knowledge based on the integrated meta-model.

3.1 As-Is Product Deployment Meta-Model

Figure 2 shows the as-is product deployment meta-model. This meta-model codifies the architecture knowledge of *deployed logistic systems* of Egemin's SPL (as-is products) required to perform updates.

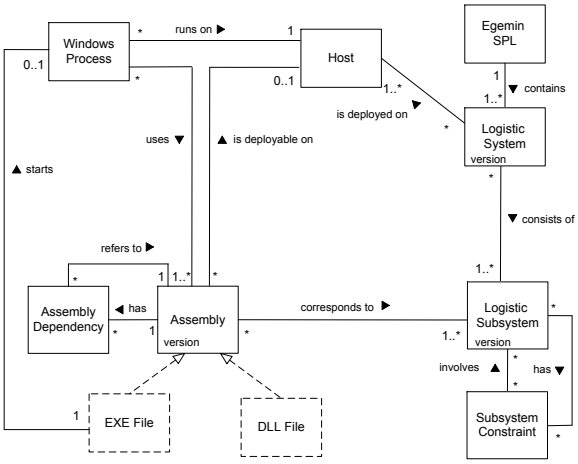

Figure 2: As-is product deployment meta-model

Egemin's SPL contains a set of *logistic systems*. A logistic system consists of *logistic subsystems*, such as management systems and control software for automated guided vehicles. Both, logistic systems and subsystems have a *version*. Logistic systems may have *constraints* to each other that put restrictions on their composition. For example, the combination of E'tricc and E'can requires the installation of E'wms, or a particular version of E'tricc excludes a particular version E'can. A logistic subsystem is deployed on a set of *hosts*. Each host contains a set of *assemblies* that correspond to particular logistic subsystems. An assembly has a version. Assemblies include *EXE files* (Executable) and *DLL files* (Dynamic Link Library). An assembly may have *dependencies* to other assemblies. A deployed executable assembly can be started as a *process* that runs on a host. A process uses one or more assemblies.

3.2 To-Be Product Deployment Meta-Model

Figure 3 shows the to-be product deployment meta-model. This meta-model codifies the architecture knowledge of Egemin's *future logistic systems* (to-be products) that is needed to perform updates.

A future logistic system can be deployed with a set of *MSI Files* (Microsoft Installer). An MSI file contains deployable assemblies that correspond to logistic systems and can be installed on hosts.

3.3 Integrated Meta-Model

Figure 4 shows the integrated product deployment meta-model that fuses the as-is and to-be product deployment meta-models.

The integrated meta-model offers the basis for an architectural repository that can be populated with the necessary architecture knowledge for a particular setting by harvesting the knowledge from the deployed system and the installation bundles. The harvested

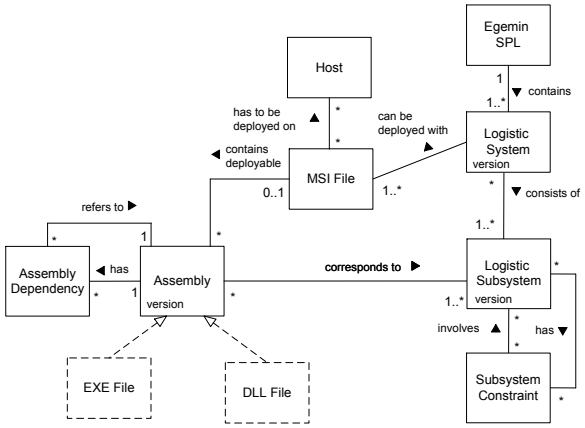

Figure 3: To-be product deployment meta-model

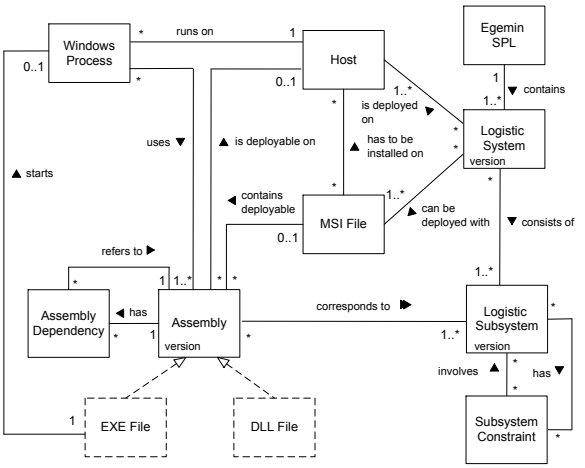

Figure 4: Integrated product deployment meta-model

knowledge is then used to derive the update procedure model that defines the sequence of update steps the maintainer has to perform to evolve the system from as-is to to-be. We explain the algorithm to determine the update steps in section 4.

3.4 Generalized Meta-Model

Figure 5 shows the generalized product deployment meta-model. The generalized meta-model specifies the conceptual entities and their relationships that codify the architecture knowledge required to evolve deployed products of a SPL, independently of any particular technology. The mapping between the concepts of the generalized meta-model and the concepts of Egemin's SPL are summarized in the following table.

General Meta-Model Concepts	Egemin Meta-Model Concepts
Product	Logistic System
Asset Base	Egemin's asset Base
Asset	Logistic Subsystem
Asset Constraint	Subsystem Constraint
Location	Host
Installation Bundle	MSI File
Resource	Assembly (EXE, DLL File)
Resource Dependency	Assembly Dependency
Process	Windows Process

In section 4, we will use the concepts of the generalized meta-

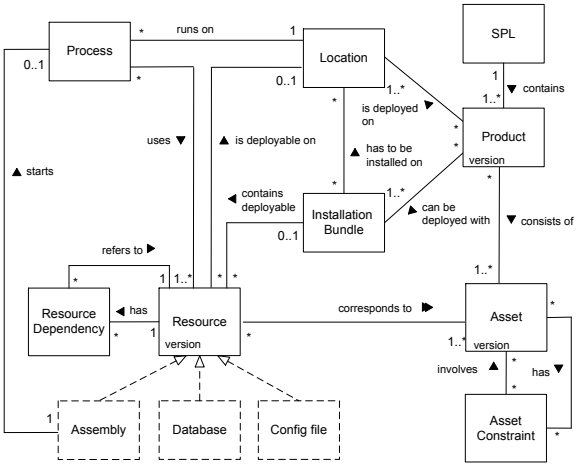

Figure 5: Generalized product deployment meta-model

model for the specification of the algorithm to generate the update steps to evolve a product of a SPL.

3.5 Supporting Framework

Figure 6 shows the primary components of the framework that we developed to support online evolution of SPL.

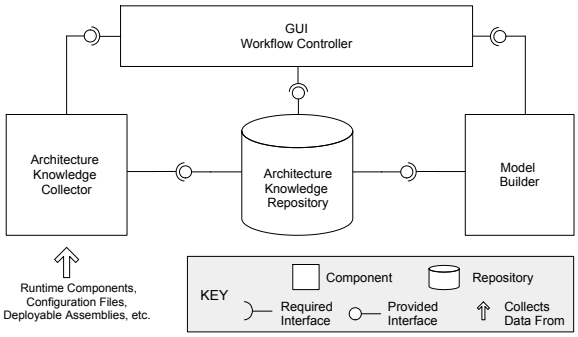

Figure 6: General overview of the supporting framework

System maintainers interact through a standard *GUI* (Graphical User Interface) to harvest architecture knowledge, build models (as-is, to-be, and update procedure model), and browse the models. The *workflow controller* triggers the *architecture knowledge collector*, *architecture knowledge repository*, and the *model builder* to execute these actions.

The architecture knowledge collector comprises a number of pluggable harvester components that perform the actual knowledge gathering. Knowledge can be extracted from run-time system components, resource files, system configurations, etc. Three example harvesters that we used to harvest knowledge for Egemin's SPL are:

- Assembly Harvester: gathers knowledge about the the assemblies of the deployed system per location, including assembles' version and compile time dependencies. This harvester includes a C# program based on the Mono.Cecil library (http://www.mono-project.com/Cecil) that supports inspection of programs and libraries.

- 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.

The architecture knowledge collected by the harvesters is used to populate the architecture repository. The repository stores architecture knowledge that complies to the integrated meta-model discussed above. We used the Eclipse Modeling Framework (EMF) as a basis for 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.

Finally, the model builder queries the architecture repository to generate on demand the architecture models requested by the maintainer. The as-is deployment model (M1) and to-be deployment models (M2) can directly be derived from the knowledge stored in the repository. The update procedure model (M3) requires further analysis of the stored knowledge. We discuss the algorithm to generate the update procedure model in the next section.

4. UPDATE PROCEDURE MODEL

We now discuss in the algorithm to generate the updated procedure model in detail. This model is derived from the analysis of the knowledge harvested from a deployed system and the installation bundles. The update procedure model lists a sequence of steps that maintainers have to perform to evolve a system correctly and with minimal interruption of its services. The algorithm is based on the assumption that the system under evolution allows online updates, This requires support for (de-)activation of components, buffering of messages, etc. Egemin's logistic platform offers such support (see section 2).

This section starts with a high-level overview of the algorithm and a description of a running example. Next, a number of definitions are introduced that will be used in the algorithm. Then, the algorithm is discussed in detail and illustrated with excerpts of the running example.

4.1 Overview Algorithm

Figure 7 shows an overview of the algorithm to generate the update procedure model.

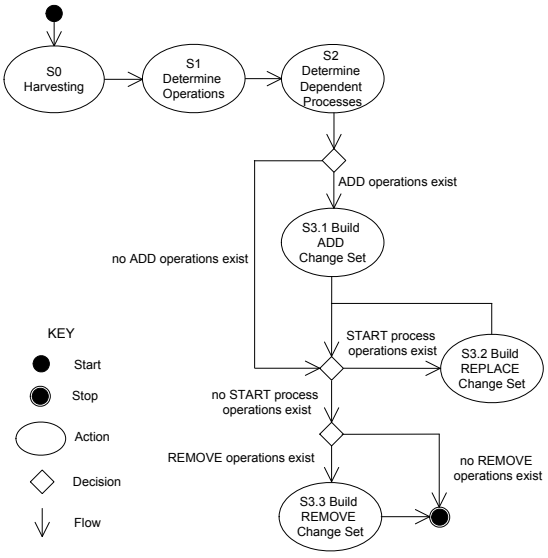

Figure 7: Overview of the update script generation algorithm.

The algorithm consists of three main steps (S1 - S3). Step S0 is a preparatory step in which the architecture knowledge is harvested from the deployed system and the installation bundles and stored in the architecture repository.

In the first step, S1, the resource operations are determined (ADD, REPLACE, REMOVE). The set of operations is derived from a comparison of the architecture 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.

In the third step, S3, all the operations are ordered to ensure that the shutdown time of the system services is minimized. 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 part 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 end with START operations for all processes with dependencies to any of the resources involved in the change set. 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 dependencies to any of the resources involved in the change set. As such, REPLACE operations do not require a shutdown of active services of the deployed system.

4.2 Running Example

We will illustrate the different steps of the algorithm with a simplified evolution scenario of the logistic system shown in Figure 1 of Section 2. The initial setting of the scenario is shown Figure 8.

The figure shows the resources and processes with their dependencies deployed on two locations 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.

4.3 Definitions

Before we explain the algorithm in detail, we first give a number of definitions.

Type Definitions

The basic types for resources, processes, dependencies etc. are derived from the generalized product deployment meta-model shown in Figure 5 of Section 3. Due to space constraints, we omit the formal specification of these types. In addition, we introduce the `Operation` type that defines atomic steps of the update procedure. We define a specialized operation for each kind of update step:

`ADD(`r `: Resource,` l`: Location)`
Adds resource r to location l.

`REMOVE(`r`: Resource,` l `: Location)`
Removes resource r from location l.

`REPLACE(`r_o`,` r_n `: Resource,` l `: Location)`
Replaces resource r_o with resource r_n at location l.

`STOP(`p `: Process,` l `: Location)`
Stops process p at location l.

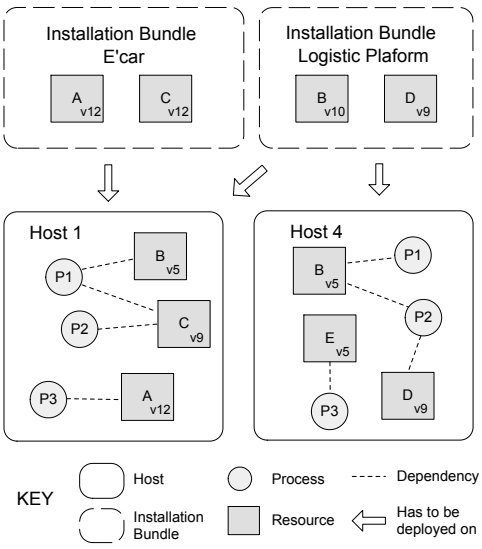

Figure 8: Example scenario to illustrate the update procedure

`START(`p `: Process,` l `: Location)`
Starts process p at location l.

The result of the algorithm is the update procedure model that we represent as an update script:

`updateScript : Operation[]`

The square brackets define a sequence, in this case a sequence of operations. We use $\{\}$ to define a regular set.

Input Functions

The harvested knowledge stored in the architecture repository is used as input for the algorithm. We introduce the following functions to access this knowledge:

`locations() : Location{}`
Returns the set of locations on which the logistic system is deployed.

`resources`$_{as-is}$`(`l `: Location) : Resource{}`
Returns the set of deployed resources for the given location.

`resources`$_{to-be}$`(`l `: Location) : Resource{}`
Returns the set of resources of an installation bundle that have to be deployed on the given location.

`processes(`l `: Location) : Process{}`
Returns a set of processes running at the given location.

`lockedBy(`l`: Location,` r`: Resource,`p`: Process): bool`
Evaluates $true$ when process p is using resource r at location l, and false otherwise.

Helper functions

The following helper functions are defined to simplify the description of the algorithm:

`getOperations(`rs`: Resource{},` os`: Operation{}): Operation{}`
Selects all the operations from the set os that are defined on resources from the set rs.

`getResources(`os`: Operation{},` l `: Location): Resources{}`
Selects all the resources deployed to location l for which an operation in the set os is defined.

`append(`*operations*`: Operation[], o : Operation)`
Appends operation *o* to the given sequence of *operations*.

4.4 Algorithm

We now give a detailed description of the subsequent steps of the algorithm to generate the update procedure model. Unless stated differently, all sets and sequences are initialized as empty. Due to space constraints, some parts of the algorithm are omitted. The complete algorithm specification is available for the interested reader[2].

Step 1. Determine Resource Operations

Initialization:
1: $toAdd : Resource\{\}$
2: $toRemove : Resource\{\}$
3: $r_{add} : Resource$
Procedure:
4: **for all** $l \in locations$ **do**
5: $toRemove \leftarrow$
6: $\{r : r \in resources_{as-is}(l) \wedge r \notin resources_{to-be}(l)\}$
7: $toAdd \leftarrow \{r : r \in resources_{to-be}[l] \wedge r \notin resources_{as-is}(l)\}$
8: **for all** $r \in toRemove$ **do**
9: $r_{add} \leftarrow x : x \in toAdd \wedge r.name = x.name$
10: **if** $r_{add} \neq nil$ **then**
11: $operations_{rep}[l] \leftarrow$
12: $operations_{rep}[l] \bigcup \{REPLACE(r, r_{add}, l)\}$
13: $toAdd \leftarrow toAdd \setminus \{r_{add}\}$
14: **else**
15: $operations[l]_{rem} \leftarrow$
16: $operations[l]_{rem} \bigcup \{REMOVE(r, l)\}$
17: **end if**
18: **end for**
19: **for all** $r \in toAdd$ **do**
20: $operations[l]_{add} \leftarrow operations[l]_{add} \bigcup \{ADD(r, l)\}$
21: **end for**
22: **end for**

Listing 1: Determining the resource operations.

The resource operations are derived from the comparison of the sets of resources of the deployed system (as-is resource set) and the installation bundles (to-be resource set) for each location. Lines `5, 6` define the $toRemove$ set. This set contains all the resources that are currently deployed, but are not present in the installation bundles. Line `7` defines the $toAdd$ set. This set contains all the resources which are present in the installation bundles and are not deployed yet.

The remainder of the procedure defines the proper resource operations based on the $toRemove$ and $toAdd$ sets. Line 9 checks for each element of the $toRemove$ set whether it also belongs to the $toAdd$ set. If the element belongs to both sets, a `REPLACE` operation is added to the set of replace operations for the corresponding resource, and the element is removed from the $toAdd$ set (lines `11-13`). Otherwise, a `REMOVE` operation is added to the set of remove operations (lines `15-16`). Finally, for all remaining elements in the $toAdd$ set, an `ADD` operation is added to the set of add operations (lines `19-21`).

```
OUTPUT:
      Host 1:
        REPLACE(Bv5,Bv10,H1), REPLACE(Cv9,Cv12,H1),
        ADD(Dv9,H1).
      Host 4:
        REPLACE(Bv5,Bv10,H4),
        REMOVE(Ev5,H4)
```

Listing 2: Resource operations for the running example

The result of step 1 for the running example is shown in Listing 2. There are two `REPLACE` operations and one `ADD` operations defined for Host 1. In addition, one `REPLACE` and one `REMOVE` operation is defined for Host 4.

[2] http://people.cs.kuleuven.be/danny.weyns/UpdateModelAlgorithm.pdf

Step 2. Determine Dependent Processes

To ensure consistency, every running process of the deployed system that uses a resource with a `REPLACE` or `REMOVE` operation has to be shutdown during the execution of these operations. The algorithm defines two sets of dependent processes: $processes_{rst}$ contains all processes that depend on at least one resource in the `REPLACE` operation set, and $processes_{stp}$ contains all processes that depend on at least one resource in the `REMOVE` operation set. Due to space constrains, we omit the specification of this part of the algorithm.

```
INPUT:
      Locked at Host 1:
          (P1 : Bv5, Cv9), (P2 : Cv9), (P3 : Av12)
      Locked at Host 4:
          (P1 : Bv5),  (P2 : Bv5, Dv9), (P3 :  Ev5)
OUTPUT:
      processesrst[Host 1] : P1,P2
      processesrst[Host 4] : P1,P2
      processesstp[Host 4] : P3
```

Listing 3: Dependent processes for the running example.

Listing 3 shows the resource dependencies of the processes for the running example. All the processes with a dependency to a resource with an `REPLACE` operation are added to the $processes_{rst}$ sets for both hosts. Process P3 at Host 4 is added to the $processes_{rst}$ set since it has a dependency to resource E_{v5} which will be removed. Process P3 at Host 1 is not affected because there is no operation defined for resource A_{v12} at this location.

Step 3. Build Change Sets

In the last step of the algorithm, the sequence of proper update operations is generated. We explain the subsequent sub-steps and illustrate them with excerpts of the running example.

The sequence of update operations are collected in the $updateScript$. First, all `ADD` operations are added to the update script in lines `8, 9`. Since `ADD` operations do not influence running processes no process operation need to be defined. Figure 9 shows the `ADD` change set for the running example which contains only the operation ADD(D_{v9},H_1) that is included in the update script shown in Listing 5.

Lines `11-19` describe the preparation stage for the composition of the `REPLACE` change sets. For each resource, the number of replace operations on different locations is determined. We assume that the names of resources can be ordered. For the running example, resources are ordered alphabetically (A...E). To minimize the impact of replace operations, the replace change sets will ordered in the update script according to decreasing number of affected resources. The result of this preparation stage for the running example is shown as the first part of Listing 5.

Next, in the lines `20-53` the `REPLACE` change sets are generated. First, the sets of affected resources and processes with dependencies are computed for each location (lines `22-32`). The computation starts with the resource that needs to be replaced on a maximum number of locations. In the running example, this is resource B, which has to be replaced on both hosts. At Host 1, resource B has dependencies with the local processes P1 and P2, and indirectly with resource C. At Host 2, resource B has dependencies with the local processes P1 and P2. Figure 9 shows how all these resources and processes belong to the single `REPLACE` change set.

Next, for each generated `REPLACE` change set, operations are added to the update script (lines `33-52`). Subsequently, `STOP` operations for the processes are added, followed by the `REPLACE` op-

Initialization:
1: $affected : Resource\{\}[|locations|]$
2: $rs_{count} : \mathbb{N}[]$
3: $r_{max} : Resource$
4: $r_l : Resource$
5: $toRestart : Process\{\}[|locations|]$
6: $toRestart` : Process\{\}$
7: $toModify[l] : Resource\{\}[|locations|]$
Procedure:
8: **for** $l \in locations$ **do**
9: $updateScript \leftarrow operations_{add}[l]$
10: **end for**
11: **for** $l \in locations$ **do**
12: $affected[l] \leftarrow \{r : r \in getResources(operations[l]_{rep})\}$
13: **end for**
14: $rs_{count} \leftarrow []$
15: **for** $l \in locations$ **do**
16: **for** $r \in affected[l]$ **do**
17: $rs_{count}[r.name] \leftarrow rs_{count}[r.name] + 1$
18: **end for**
19: **end for**
20: **while** $\sum_{l \in locations} |processes_{rst}[l]| > 0$ **do**
21: $r_{max} \leftarrow r : r \in \bigcup_{l \in locations} affected[l] \wedge \max rs_{count}[r.name]$
22: **for all** $l \in locations$ **do**
23: $r_l \leftarrow r : r \in affected[l] \wedge r.name = r_{max}.name$
24: $toRestart[l] \leftarrow \{\}$
25: $toRestart' \leftarrow \{p : p \in processes_{rst}[l] \wedge lockedBy(l, r_l, p)\}$
26: **while** $|toRestart[l]| < |toRestart'|$ **do**
27: $toRestart[l] \leftarrow toRestart'$
28: $toModify[l] \leftarrow \{r : r \in affected[l] \wedge \exists p \in toRestart[l] : lockedBy(l, r, p)\}$
29: $toRestart' \leftarrow$
30: $\{p : p \in processes_{rst}[l] \wedge \exists r \in toModify[l] : lockedBy(l, r, p)\}$
31: **end while**
32: **end for**
33: **for all** $l \in locations$ **do**
34: $processes_{rst}[l] \leftarrow processes_{rst}[l] \setminus toRestart[l]$
35: $processes_{stp}[l] \leftarrow processes_{stp}[l] \setminus toRestart[l]$
36: **for all** $p \in toRestart[l]$ **do**
37: $append(updateScript, STOP(p, l))$
38: **end for**
39: **end for**
40: **for all** $l \in locations$ **do**
41: **for all** $o \in getOperations(toModify[l], operations_{rep}[l])$ **do**
42: $append(updateScript, o)$
43: **end for**
44: **for all** $r \in toModify[l]$ **do**
45: $rs_{count}[r.name] \leftarrow 0$
46: **end for**
47: **end for**
48: **for all** $l \in locations$ **do**
49: **for all** $p \in toRestart[l]$ **do**
50: $append(updateScript, START(p, l))$
51: **end for**
52: **end for**
53: **end while**
54: **for all** $l \in locations$ **do**
55: **for all** $p \in processes_{rst}[l]$ **do**
56: $append(updateScript, STOP(p, l))$
57: **end for**
58: **for all** $o \in operations_{rem}[l]$ **do**
59: $append(updateScript, o)$
60: **end for**
61: **end for**

Listing 4: Build update scripts.

erations for the resources, and finally, START operations are added for the stopped processes. The set of operations for the single REPLACE change set in the running example are shown in Listing 5. The sequence consists of the STOP operations for processes P1 and P2 at both hosts, followed by the REPLACE operations for the various resources, and finally the START operations are added.

Finally, the REMOVE change set is generated (lines 54–61) for all the resources for which REMOVE operations are defined. The operations are added to the update script, preceded by STOP operations for the dependent processes that are no longer needed after the update. Figure 9 shows that the REMOVE change set comprises only

```
PREPARATION STAGE:
       rs_count[A] : 1      rs_count[B] : 2
       rs_count[C] : 1      rs_count[D] : 0
       rs_count[E] : 1

OUTPUT: UPDATE SCRIPT:
       ADD(D_v10, H_1)

       STOP(P_1, H_1)              STOP(P_2, H_1)
       STOP(P_1, H_4)              STOP(P_2, H_4)
       REPLACE(C_v9, C_v12, H_1)   REPLACE(B_v5, B_v10, H_1)
       REPLACE(B_v5, B_v10, H_4)
       START(P_1, H_1)             START(P_2, H_1)
       START(P_1, H_4)             START(P_2, H_4)

       STOP(P_3, H_1)
       REMOVE(E_v5, H_4)
```

Listing 5: Update script for the running example

resource E at host 4 together with process P3. The corresponding operations conclude the update script as shown in Listing 5.

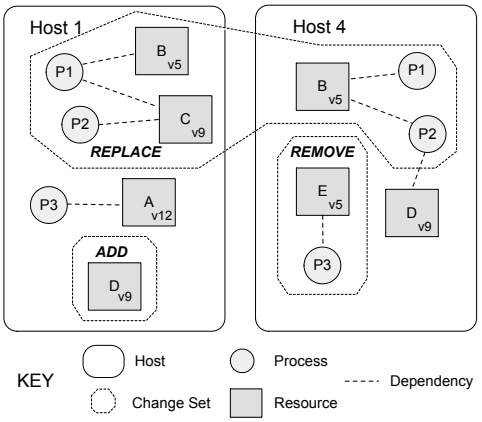

Figure 9: Change sets for the running example.

5. EVALUATION

We have evaluated the approach for online evolution of SPL in an empirical study. In this paper, we briefly summarize the results of this study. A detailed report of the study is available for the interested reader[3]. We evaluated a total of 68 updates of industrial logistic systems of Egemin performed by 17 professionals, half of them with Egemin's traditional update approach, the other half with the tool that automatically generates update scripts. We formulated hypothesis with respect to the correct execution of product evolutions (R1) and the availability of services during the updates (R2). Statistical analysis revealed significant differences for the tested hypothesis. In particular, the results demonstrate that 44% of the updates with the traditional approach contain errors, while all the updates with the tool were performed correctly. Furthermore, the results show that with the traditional approach 58% of the process shutdowns were unnecessary, while there were only 7% redundant process shutdowns with the tool.

In addition, we probed whether the availability of architectural knowledge changed the maintainers' attitude to evolution tasks. Therefore, we used a questionnaire that the subjects completed after the update tasks. The results indicate that maintainers felt more confident that the updates were performed correctly when they use the tool with explicit update scripts. The fact that the number of system modifications was almost seven times higher in the traditional approach as with explicit update scripts confirms this finding.

[3] http://people.cs.kuleuven.be/danny.weyns/EmpiricalStudyOnlineUpdates.pdf

6. RELATED WORK

In theory, the choice of an architectural knowledge management strategy should be made explicitly and tailored to the organization's needs. However, in practice, there never seems to be enough time to document architecture knowledge with enough rigor to be useful[4]. Researches from the GRIFFIN project list a number of issues related to architecture knowledge management in a organization [11]. They report lack of consistency between documentation and the actual system, communication overhead between stakeholders, and lack of explicit collaboration between maintenance teams. Our experiences with Egemin confirm these findings.

Several authors have pointed to the complexity of managing SPL evolution. [17] argues that SPL evolution is complex process because of various interdependencies between software assets. [13] points to the complexity associated with the typical mismatch between architectural variability and the actual variability.

To deal with the lack of detailed architecture knowledge of SPL evolution, a number of methods and tools have been proposed. [16] proposes architecture reconstruction as a means to integrate product features in the platform and to maintain the architectural integrity of the platform. Tools such as Ménage [12] (xADL 2.0 based representation), SELECTA [10] (which uses composition of meta-models) and pure::variants [6] (industrial tool) help with the recovery of architecture knowledge. Whereas the described approaches focus on evolution of the design and implementation time artifacts, we address the codification of architecture knowledge for deployment time evolution of the SPL products.

Our approach benefits from the research in the domain of software architecture reconstruction. [8] provides a general overview of the state of the art in this area. Comparing to existing approaches, we derive as-is models from heterogeneous set of information sources of the deployed products. We also extract the to-be models, i.e. the valid installation bundles of updated products.

Several methods for differenting architecture models have been proposed. [9] uses source code level differencing to analyze the variability in SPL in a reverse engineering context. [1] proposes a tree-to-tree correction algorithm for differencing and merging architectural models. In our work, differencing models has a specific and practical objective. In particular, the algorithm presented in this paper uses harvested architecture knowledge to generate automatically the update scripts for evolving deployed products of a SPL.

7. CONCLUSIONS AND FUTURE WORK

In a join effort with Egemin, we faced the problem with a lack of explicit architecture knowledge to evolve deployed products of a SPL correctly and efficiently. To tackle this problem, we developed an automated approach to support maintainers of product lines. Together with Egemin's key stakeholders, we codified the architecture knowledge required to evolve SPL in an integrated meta-model. This meta-model offers the basis for a repository that can be populated with the relevant knowledge harvested from system artifacts. We also codified the architecture knowledge of the update procedure for evolving systems in an algorithm that generates the sequence of steps that maintainers have to perform to evolve a deployed system to a new version in a correct and efficient manner.

We developed a supporting tool for the approach and used it to perform a controlled experiment in which we evaluated the updates of 68 industrial logistic system, half of them performed with Egemin's traditional update approach, the other half with the tool. The results give evidence that using the tool significantly improves correctness and availability of services during system evolutions. Moreover, we observed a positive change in the maintainers' attitude to the evolution task, in particular with respect to their confidence in performing their tasks with the tool.

Architecture knowledge typically refers to design decisions, rationale, and the like. In this paper, we complement this traditional perspective with a specific form of architecture knowledge. In particular, we consider knowledge about the course grained structures of SPL products, dependencies between components of products, and the update procedure to evolve deployed products.

Although we successfully applied our approach to Egemin's SPL, several research challenges remain. First, we plan to formally proof the correctness and availability properties of the proposed approach. Second, we plan to challenge the generality of the approach by applying it to a different domain with different technology. Finally, we plan to extend our work by exploring evolution scenarios in which the asset base of the SPL evolves, rather than particular products. An example scenario in Egemin's context is the introduction of a new module that deals with timing issues of external logistic services.

8. REFERENCES

[1] M. Abi-Antoun, J. Aldrich, N. Nahas, B. Schmerl, and D. Garlan. Differencing and merging of architectural views. *Automated Software Engineering*, 15(1):35–74, 2008.

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

[3] N. Anquetil et al. A model-driven traceability framework for software product lines. *Software & Systems Modeling*, 2010.

[4] P. Avgeriou, P. Kruchten, P. Lago, P. Grisham, and D. Perry. Architectural knowledge and rationale: issues, trends, challenges. *Softw. Eng. Notes*, 32:41–46, July 2007.

[5] M. Babar, T. Dingsyr, P. Lago, and H. van Vliet. Software architecture knowledge management. *Springer*, 2009.

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

[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] S. Duszynski. Visualizing and Analyzing Software Variability with Bar Diagrams and Occurrence Matrices. In *14th Software Product Lines Conference, SPLC*, 2010.

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

[11] R. Farenhorst, P. Lago, and H. van Vliet. Prerequisites for successful architectural knowledge sharing. In *18th Australian Software Engineering Conference, ASWEC*, 2007.

[12] A. Garg et al. An environment for managing evolving product line architectures. In *Software Maint. Conf.*, 2003.

[13] S. A. Hendrickson and A. van der Hoek. Modeling product line architectures through change sets and relationships. In *International Conference on Software Engineering, 2007*.

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

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

[16] C. Riva and C. Del Rosso. Experiences with software product family evolution. In *Principles of Software Evolution*, 2003.

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

[18] F. Van Der Linden, K. Schmid, and E. Rommes. *Software product lines in action: the best industrial practice in product line engineering*. Springer, 2007.