An Approach for Reusing Software Process Elements based on Reusable Asset Specification: a Software Product Line Case Study

Karen D. R. Pacini
and Rosana T. V. Braga

Institute of Mathematics and Computer Sciences
University of São Paulo
São Carlos, SP, Brazil
Email: karenr@icmc.usp.br, rtvb@icmc.usp.br

***Abstract*—Software reuse is becoming an important focus of both academic and industrial research since the rising demand for new software products and technologies is constantly growing. The short time to market, limited resources and lack of specialists are the main reasons for this investment on software reuse. As long as customers demand speed to deliver, there is a increasing special concern about software quality. In this context, we propose an approach to support both better time to market and software quality from reusing software process elements using the Reusable Asset Specification (RAS). This approach presents a mapping structure to represent process elements as reusable assets. The sharing of process elements among several projects aims to decrease time spent on defining the process model, as well as reducing the space used to store processes and their elements. Documenting these processes will also be facilitated, since it is possible to reuse a whole process or process's sub-trees that have already been documented or even certified. To illustrate our approach, we present a case study where a Software Product Line (SPL) process is mapped to RAS, highlighting the issues raised during the mapping and how we proposed to solve them.**

Keywords–Software Process; Process Reuse; Software Product Line; RAS; Reusable Asset Specification.

I. INTRODUCTION

The software industry has been adapting to the large increase of demand arising from the constant evolution of technology. The concept of software reuse gets an important role on this new way of software manufacturing, in which development time is reduced, while quality is improved [1]. Software product lines (SPL) emerged in this context, to support reuse by building systems tailored specifically for the needs of particular customers or groups of customers [1]. Reuse in SPL is systematic – it is planned and executed for each artifact resulting from the development process.

The most common SPL development approaches, such as Product Line UML-Based Software Engineering (PLUS) [2], Product Line Practice (PLP) [1], etc., are focused on the process to support the domain engineering and/or the application engineering, without considering the computational tools that support the process. Indeed, the choice and use of tools are made apart from the process and are strongly associated to variability management, i.e., dealing with the definition of the feature model and its mapping to the artifacts that implement each feature. Some examples of these tools include Pure::Variants [3], Gears [4], and GenArch [5].

To support a uniform representation of reusable assets, in 2005 the Object Management Group (OMG) has proposed the Reusable Asset Specification (RAS), which allows a common approach to be used by developers when storing reusable assets [6]. RAS offers a basic structure (CORE), but allows the creation of extension modules in order to adequate to the particular needs of each project. The specification is available via XML Schema Definition (XSD) and XML Metadata Interchange (XMI) files and its usage is defined by profiles. In the particular case of SPL, the use of RAS to model repositories contributes to make assets compatible to each other.

Several extensions to the original RAS profile have been presented, however their focus is on improving the representation of specific types of reusable assets [7][8][9]. However, we have not found any works showing how RAS could be used to represent the elements of a process, in particular in the SPL domain. This is not a trivial task, as there are several decisions to be made, for example, how process elements that are compositions of other elements should be stored using RAS. This also motivated this work, as we are interested in extending reuse to the process level, i.e., to facilitate the reuse of process phases, activities, or any other assets related to the process itself. In the particular case of SPL, it is important to consider approaches successfully applied in practice and well documented, such as the approaches proposed by Gomaa [2] and by Clements [1], for creating the case study to apply our approach. The use of these approaches is supported by their wide documentation and can illustrate scenarios for a variety of real applications.

So, the main motivation for this work is that, although SPL development approaches focus on establishing the process itself, only SPL artifacts are considered as reusable assets, rather than the process elements that could bring a number of benefits if appropriately reused. Indeed, current approaches do not motivate process reuse, which causes rework each time a process needs to be instantiated from the general processes. Additionally, experience gained from successful projects executed in the past are not taken into account when new similar projects arise, because they were not adequately stored for reuse.

Therefore, considering this context, this paper aims at proposing an approach that allows the storage of process elements using RAS, in particular in the SPL domain. This

 ISBN: 978-1-61208-438-1

can leverage the reuse of each SPL process element across several projects, in an independent way, potentially increasing reuse. As process elements can be composed of other process elements, reuse can be done both for single elements or elements in higher levels of the hierarchy, which contain one or more elements. So, the paper has also the objective of describing the main problems found when trying to map process elements to RAS and gives insights on how this has been solved by our approach.

Software & Systems Process Engineering Meta-model (SPEM) version 2.0 [10] was the process meta-model that inspired our approach. This is because SPEM is being widely employed as a software process modeling language, as indicated by Garcia-Borgonon [11]. It was used as basis to define our own structure, which represents several types of processes, as presented in the paper. It is important to notice that, although this paper presents a SPL process to exemplify the approach, any software processes that matches the model proposed in this paper could be used.

The remainder of this paper is organized as follows. Section II presents some background on SPL development and RAS. Section III presents our approach to store SPL process elements as reusable assets using RAS. Section IV presents a case study to illustrate our approach. Section V discusses related work and, finally, Section VI presents conclusions and future work.

II. BACKGROUND

In this section, we describe a SPL technique (PLUS) and a reuse standard (RAS) that have been used as a foundation for the proposed approach, so they are important to allow its understanding.

A. SPL Development Process: the PLUS approach

PLUS [2] is the SPL process chosen as a case study to illustrate the approach presented in this paper. However, any other SPL process could have been used, since the main idea is to illustrate how a SPL process can be represented using RAS.

PLUS employs methods based on the Unified Modeling Language (UML) [12] to develop and manage SPLs. Its main goal is to model features and variabilities of an SPL. The approach is based on the Rational Unified Process (RUP) [13] and each phase corresponds to a RUP work-flow with the same name.

The process used by PLUS is evolutionary and has two main activities: the product line engineering (or Domain Engineering) and the Application Engineering (configuration of the target system that results in a new product). For each activity, either in Domain or Application Engineering, there is a corresponding evolutionary process (Evolutionary Software Product Line Engineering Process - ESPLEP).

According to Gomaa [2], ESPLEP life cycle for Domain Engineering is composed of five activities with the three most important: requirements modeling, analysis modeling and design modeling, as can be observed in Figure 1. During requirements modeling, the SPL scope is defined, resulting in use cases and feature models. The analysis modeling includes static modeling, dynamic modeling, finite state machine modeling, as well as the construction of objects and analysis of dependencies between features and/or classes. The design modeling involves the definition of the SPL architecture.

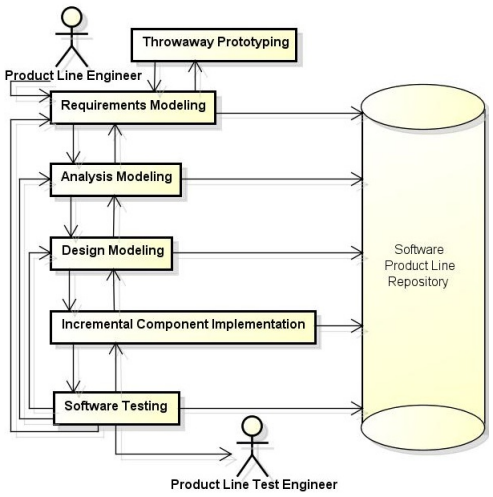

Figure 1. Process for applying the PLUS approach to SPL Engineering Phase - Adapted from Gomaa [2]

These are the basic activities, but variations can be added to the process. A characteristic of this approach is that stereotypes are used in the diagrams to identify different types of use cases, class diagrams or object diagrams (e.g., to denote mandatory or optional features).

B. Reusable Asset Specification

There is an increasing demand to ease software reuse, as it involves high costs associated to creating, searching, understanding, and using the assets found in a specific context. So, the creation of standards to organize and package assets is necessary. In this context, the OMG has proposed the Reusable Assets Specification (RAS) [6], which is a group of object management standards to allow the packaging of digital assets to improve their reusability.

RAS supplies, through a consistent standard, a set of guidelines that help to structure reusable assets. This can reduce conflicts that would arise when trying to reuse them. RAS models are based on UML [12] and Extensible Markup Language (XML).

The specification describes the reusable assets based on a model called Core RAS, which represent the fundamental elements of an asset. This core model can be adapted if necessary, by creating Profiles. RAS Profiles are a formal extension of core structure, which allows to add or improve information according to a specific context. They can be created to introduce more rigid semantics or constraints, however they must not change core definition or semantics. OMG supplies the default profile, based only in Core RAS, and also two customized profiles to be used in specific situations: the default component profile to support the principles and concepts of software components, and the default Web Service Profile that describes the client portion of a web service. Other extensions exist as well [7].

Core RAS packaging structure is split in five major sections (or entities): Classification, Usage, Solution, Related Assets and Profile, as shown in Figure 2.

 ISBN: 978-1-61208-438-1

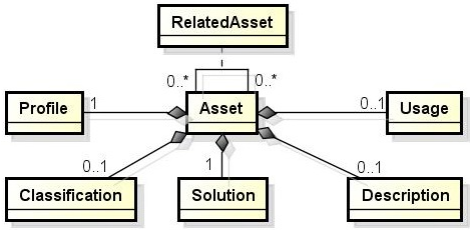

Figure 2. Core RAS Major Sections, Adapted from OMG [6]

Classification is a container entity used to allow the classification of assets to ease their further retrieval. It can contain descriptors, tags, and values, besides the context (domain, development, test, deployment, etc.). Usage contains usage instructions that improve the understanding of the asset before its usage, as well as how to perform the customization of the variation points. Solution contains the artifacts of the asset, which can be requirements, models, code, tests, documents, among others; Related Assets - describes other assets related to this one, together with the relationship type (e.g., aggregated, similar, dependent). Profile defines the version of the profile to which the asset refers to (e.g., the default profile, or any other extensions).

III. The Proposal

Although different SPL development processes share a common basis, they also contain variabilities. This motivates representing them as reusable assets. Therefore, in this paper, we propose to store process elements regarding SPL development as reusable assets using RAS (see Section II-B). In order to accomplish that, it has been defined a process modeling structure to represent processes.

A. Process Modeling Structure

The process modeling structure used in our approach is shown in Figure 3. It contains several elements: process model, phase, activity, artifact, etc. This structure was inspired on OMG SPEM 2.0 [10] concepts, and aims to represent all process elements for a software development process, i.e., it must be capable of representing processes from different development approaches existing in the literature (in the SPL domain we can mention ESPLEP - see Section II-A).

It is important to notice that this model is used to represent both the process template (i.e., the process model as defined by its authors) and the process instance, which is derived by instantiating the process template for particular purposes. A process instance refers to a template but has its own elements, according to the process execution. This is important because we may want to reuse not only the templates, but also the instances that were successful in a particular context and thus can be recommended when similar situations occur. For example, PLUS is a process model (template) that can be reused in a concrete SPL project, resulting in a process instance. Later, when a new project begins in a similar context, instead of reusing PLUS, we might want to reuse the instance instead, because it is already customized to the new context.

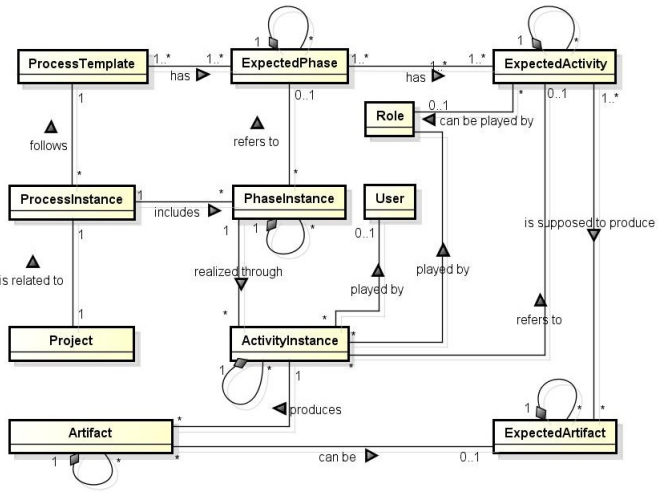

Figure 3. Process Model derived from SPEM

B. Modeling Structure Applied to Process Template and Instance

To illustrate the usage of the model proposed in Figure 3 we show, in Figure 4, the example of a random software development enterprise that is developing a SPL to ease the development of applications for hotel management (this example was chosen as the business is simple and it helps to understand the difference between process template and process instance). The development process was instantiated from ESPLEP (Figure 1) and here we focus only on the domain engineering phase. The domain engineer starts by creating a specific Project (*Hotel SPL Project*) and, associated to it, a Process instance (*Hotel SPL Process*), which in turn is associated to ESPLEP. In the figure, we use stereotypes to identify the roles played by each class in the example and [...] to express that other analogous instantiations can take place.

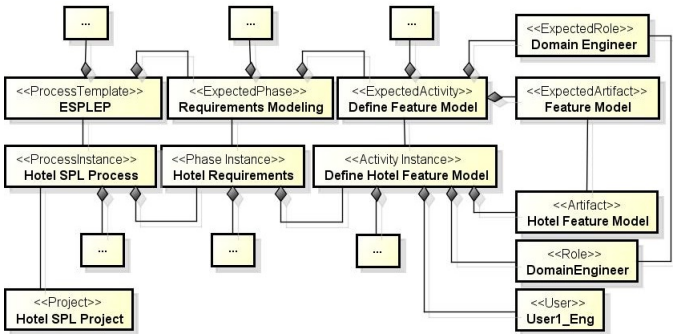

Figure 4. Example of ESPLEP instantiation on the proposed Process Model.

For each ESPLEP process element, we have to analyse whether it is adequate to our Hotel SPL process and, if so, create the respective instance. Additionally, the Hotel SPL process can be adapted to the particular context of the enterprise, for example adding new activities or artifacts, skipping optional activities, etc., as long as ESPLEP allows these adaptations. This is possible because, during the definition of ESPLEP modeling, it has been defined which elements are mandatory or optional. Mandatory elements need to have an instance, while

 ISBN: 978-1-61208-438-1

optional elements can be omitted.

C. Mapping Process Elements to RAS

We propose to map each process element to an independent RAS *Asset*, as shown in Figure 5. We discuss the rationale for that later in Section III-D. In the figure, this mapping is done considering the example of a template process and its associated elements. At the left-hand side of the figure, we have the process template, in the middle we have RAS, and at the right-hand side, examples of values assumed by the properties.

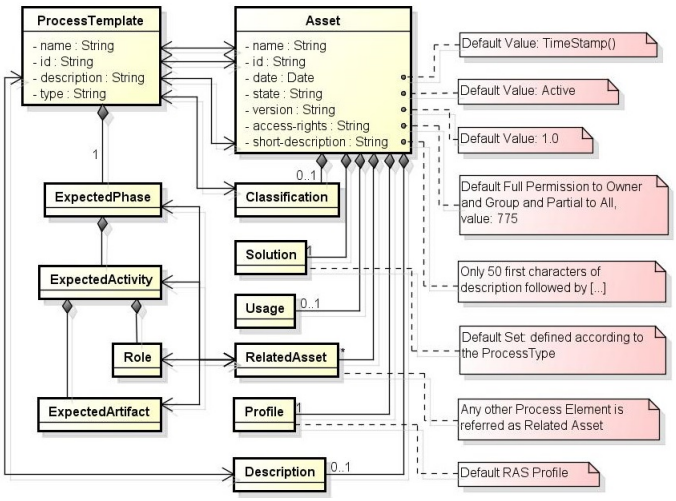

Figure 5. Example of a Mapping Structure from Process Template to RAS.

As presented in the figure, the attributes *name* and *id* have a direct relationship in both structures (*Process Template* and *Asset*). The *description* attribute is mapped to the *Description* RAS attribute, but is limited to the first 50 characters of the *short-description* attribute from the RAS *Asset*. The *type* attribute is mapped to the *Classification* element in RAS. The other process elements are treated as independent *Assets*, so they are created separately in the RAS structure and then related to each other through the *RelatedAsset* RAS element.

After doing this mapping, all the process information is stored in a RAS structure, however, according to RAS, some mandatory elements still need to be filled in. For example, the *Asset* has attributes: *date*, *state*, *version* and *access-rights*, which can be completed with default values as shown in Figure 5. The *Profile* element is mandatory and identifies which profile is being used to represent the *Asset*. In this case, we are using the *DefaultProfile*, version 2.1, as shown later in Section IV-A.

Another important mandatory element in the RAS structure is the *Solution*, which has to be filled in with information and related documents corresponding to the process element being stored. For example, in the Process Template the representation could be a file containing the complete structure of the process in an XML document. There are other optional elements exemplified in the case study.

D. Increasing the Potential Reuse

An important goal of the proposed approach is to enhance the potential reuse of processes, as well as decreasing time, effort, and storage space when creating and instantiating processes. By storing each element independently as a reusable asset, it is possible to share it among different processes and process instances, as illustrated in Figure 6. When an element is shared among processes, the whole tree with related elements associated to the root element is shared as well.

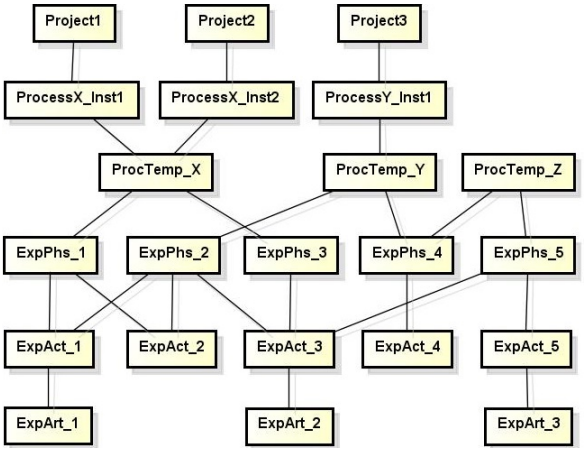

Figure 6. Example of Potential Reuse

As can be observed in Figure 6, the process element identified by ExpAct_3 represents an expected process activity, which is being referenced by three different expected phases. These phases are referenced by two different processes. The reuse of process elements as suggested by our approach allows we to share elements both in the process template and in the process instances, although the example in Figure 6 illustrates reuse only in template elements. This leads to a greater reuse potential, besides the fact that any repository or tool based on RAS can be easily used.

IV. Case Study

As described in the previous section, our approach allows the representation of process elements according to RAS, both for the process template and its execution. ESPLEP is composed of five expected phases (see Section II-A): Requirements Modeling, Analysis Modeling, Design Modeling, Components Incremental Implementation, and Software Testing. Each phase is composed of expected activities and their expected artifacts.

RAS is flexible regarding the possible ways to use, extend, and represent assets according to each project needs. Our approach is one of many possible ways to use it. We recommend to follow this usage pattern to ease the retrieval of assets later, i.e., client applications for searching assets will be easier to implement if they know that the underlying structure is based on RAS. However, it is important to observe that there is a minimum set of information required for a reusable asset to be considered in conformance with RAS: it has to indicate the used profile, at least one artifact and the basic information about the asset, as shown in Figure 5. Additionally, there are other information that can also be stored, so this case study aims at showing a possible way of using RAS elements to store process elements. This is shown in the following subsections.

A. Profile

The *Profile* element (Figure 7) refers to the asset representation structure that is being currently used. The RAS (*Core*)

 ISBN: 978-1-61208-438-1

is abstract, thus only profiles are instantiated and all of them are derived from the core. The derived profile that is closest to the core, and was chosen to be used in this work, is called *Default Profile*. More specifically, we adopted version 2.1, as it is compatible to any profile extending it.

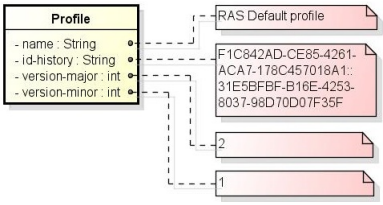

Figure 7. Example of use of Profile RAS Element.

B. Solution

The *Solution* element refers to the artifacts that compose the reusable asset. An asset can be seen as a set of artifacts (at least one artifact is required). The artifacts are the main reuse goal, and they can be classified into several types, like documents (doc, pdf, txt), code (java, sql, php, C#), descriptors (XML, XSD, HTML), and others.

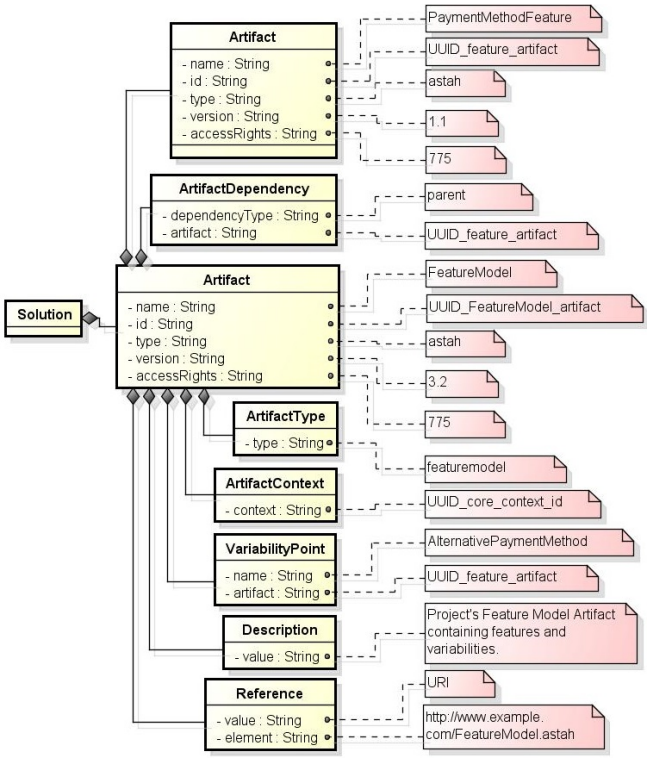

Figure 8. Example of use of Solution RAS Element.

Figure 8 presents an instance of the *Solution* element. Considering the hotel SPL introduced on Section III-B, the solution contains the Feature Model, defined during the *Define Hotel Feature Model* activity of the *Hotel Requirements* phase.

The main artifact stores (in fact it is a reference to where the real object is) the model itself, and has attributes such as name, identifier, type (file type, e.g., .astah), version, and access rights of the artifact, which we have defined as 775, following the Unix model, i.e., the artifact owner and the group to which he belongs to have total rights, while other users can only read and execute. RAS also recommends the use of Universally Unique IDentifiers (UUIDs).

The *ArtifactType* represents the artifact logical type, indicating what the artifact represents in the model (in the example, it is a SPL Feature Model). The *ArtifactContext* represents the context in which this artifact is useful. In the example, as the feature model is essential for the asset, it is classified with the *Core* type, as shown in the figure.

The *VariabilityPoint* element describes artifact variabilities. In the example, the artifact has a feature called PaymentMethod, which has several different alternatives. In this case, the feature that has variability is presented in this element, while the corresponding alternative features and variability rules are defined in the *Usage* element described later. Also, this artifact has other dependent artifacts representing each feature of the feature model. This allows the reuse of the feature model itself and the corresponding artifacts that implement them, not only during application engineering, but also in the domain engineering of other SPLS of the same domain (for example, PaymentMethod could be used in many different SPLs).

C. Classification

The *Classification* element refers to the asset descriptors or classifiers. It can include more than one descriptor or even schemas to describe the asset. It is also possible to define the contexts that will be referenced by artifacts and by the *Usage* and its activities.

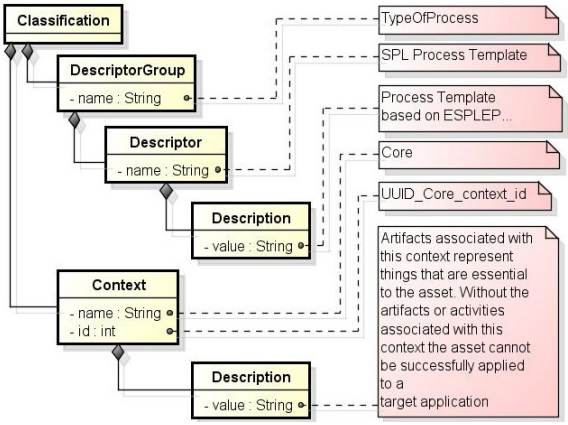

Figure 9. Example of use of Classification RAS Element.

Figure 9 presents an instance of the *Classification* element. Assume that we want to classify the *SPL Process Template* process element. So, the "SPL Process Template" is a *Type of Process* kind of classification. To map this information into RAS, first we need to define a group of classifiers that represent what kind of classification we are making, in this case, we are classifying as *TypeOfProcess*, so this will be the name of our *DescriptorGroup* RAS element. Defined that, we can define the value for this type, each value is defined as an instance of *Descriptor* RAS element, which in this case is *SPL Process Template*. An asset may have many classifiers, for example this same asset could be classified by DescriptorGroup *ProcessElement* and Descriptor *ProcessTemplate*.

 ISBN: 978-1-61208-438-1

In an analogous way, all the other process elements can be classified using this structure, for instance *SPL Process Instance* which is also part of *TypeOfProcess* DescriptorGroup; *Expected Phase*, *Expected Phase* and *Expected Artifact*, which are also part of *ProcessElement* DescriptorGroup; and so on.

The *Context* element is used to refer to artifacts and activities in the Asset context. As shown in the figure, the *Core* context represent that the related artifact/activity is essential to the asset.

D. Usage

The *Usage* element is used to keep information about how to use the asset (manuals for example), as well as which tasks have to be executed in order to that asset works correctly. This information can be relative to the context, to a specific artifact, or to the whole asset.

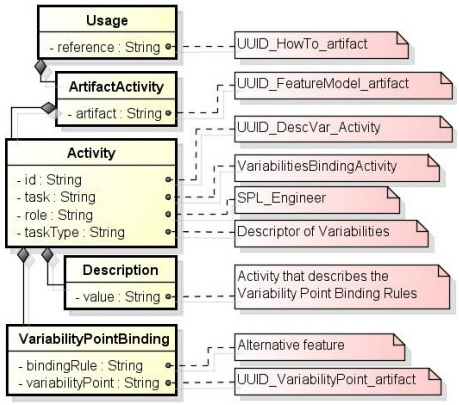

Figure 10. Example of use of Usage RAS Element.

Figure 10 presents an instance of the *Usage* element, in which the *reference* attribute refers to an artifact contained in another asset, identified by *UUID_HowTo_artifact*. This artifact represents a document with instructions on how to use and interpret the components of the *Usage* element. Even though this artifact is not mandatory, it can be useful to improve reuse. Schemas and other types of artifacts can also be referenced.

In the hotel SPL example, as mentioned before, there is a feature named PaymentMethod that represents a variability. Figure 10 presents the rules for using this variability from instances of the *VariabilityPointBinding* element. These instances are contained in an *Activity* element, which itself is contained in an *ArtifactActivity* element. This means that the activity is relevant only in the context of the referenced artifact. The activity (*VariabilitiesBindingActivity*) describes the rules to be followed for binding variabilities of the Feature Model artifact.

In the example, the artifact identified by *UUID_feature_artifact* has a variability called *AlternativePaymentMethods*. According to the rule defined in *VariabilityPointBinding*, the dependent artifacts (children) refer to alternative features of the Feature Model as defined in the *bindingRule* attribute.

E. Related Assets

The *Related Asset* element specifies the relationships among reusable assets. From these relationships, it is possible to assemble a dependency tree with all related assets.

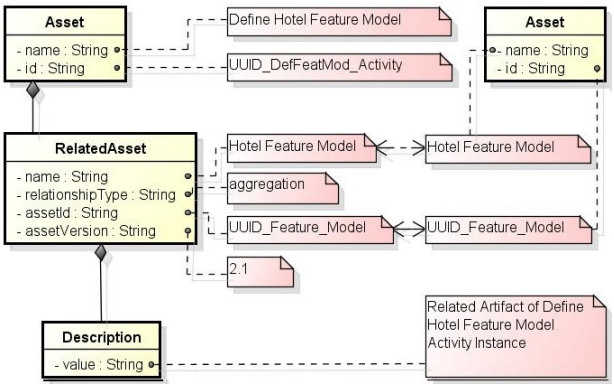

Figure 11. Example of use of Related Asset RAS Element.

Figure 11 presents an instance of *Related Asset*. It has a *name* attribute that corresponds to the name of the asset being related with, as well as *assetID* and and *assetVersion* representing the ID and version of the related asset, respectively. RAS defines some types of relationship (e.g., an *aggregation* in the figure) and allows other types to be created.

In this example, there is a relationship between <<ActivityInstance>> Define Hotel Feature Model and <<Artifact>> Hotel Feature Model. In this scenario, only the activity is related to the artifact, not the opposite way. This means that only the activity has visibility of the artifact. If the visibility was supposed to be both ways the artifact should have a relationship with the activity as *parent* type.

Navegability on relationships is very important on the reuse context. For example, when selecting an element for reuse, all the dependencies of this element will be loaded as well. In this case, if someone wishes to reuse the activity of Figure 11, the artifact would be loaded with it. But if they want to reuse just the artifact, it is possible because the artifact has no relationships (dependencies). Thus, if an element depends on another element and they must be loaded together, the relationship must be defined in both elements.

V. Related Work

While searching for RAS related studies on the literature, we could not find any descriptions or examples of how they use RAS to represent and to pack their reusable assets. Most of the studies focus on presenting how to identify and use the assets, rather than on how to map them into the RAS structure.

Part of the studies found on our research proposes RAS extensions to fit to several different purposes. The application of RAS in these studies are usually very specific to each case, for example to store components, or services, or process generated artifacts and so on. One of proposed extensions is presented by Mikyeong Moon et al. [8]. They propose an extension of the RAS Default Profile to store, manage, and trace variabilities on a Software Product Line.

Another example of using RAS to represent reusable assets is proposed by Islam Elgedawy et al. [14]. They propose to use the specification to represent Component Business Maps (CBMs) to allow the early identification of reusable assets in a project. They do not specify which RAS Profile they use or if they created their own extension to represent their assets.

 ISBN: 978-1-61208-438-1

An example of RAS application that does not use new extensions is proposed by Nianjun Zhou et al. [15]. In their approach, they present a legacy reuse analysis and integration method to support modeling legacy assets in a SOA context. To store the assets extracted by their approach they use the IBM Rational Asset Manager Repository (RAM), which is typically used for storage of unstructured assets (jar, war and ear) and documents specified using RAS.

There are other online repositories based on RAS in the web, one example is LAVOI created by Moura [7] and OpenCom created by Ren Hong-min1 et al. [9]. Both extended the RAS profile to adapt it to a wide range of types of assets and to facilitate assets classification, search and use.

Although there is a number of works related to RAS, none of them brings explicit examples of how to use RAS and to map the attributes as this work does. In addition, no studies were found that suggest process elements reuse based on RAS.

VI. Conclusions and Future Work

This work presented an approach to represent process elements as reusable assets using the RAS. For this, a mapping of these elements into the RAS structure was presented. The mapping not only represents the main information of process elements into RAS mandatory structure but also guides the user on how to use several other structures that are available at the RAS Default Profile. This representation makes possible the creation of a repository of process elements, which may highly increase the potential of reuse. Reusing processes and process elements has lots of benefits, such as improving time to market, decreasing time spent and staff effort, increasing quality. Besides that, a repository may be built with mechanisms to recommend process and process elements according to the user type and application context.

The contribution of this paper is applicable not only in the SPL context, but in any software processes in other contexts, as long as they follow our proposed meta-model structure derived from SPEM. For processes with different structures, a mapping analogous to that provided here can be done.

Another advantage of using the proposed approach is that process elements can also be shared among different project contexts, both for template and instance applications. An additional contribution of this paper is to serve as a documented example of how to use RAS in a practical way, since no example of usage details was found on our research in the literature.

As future work, we will implement a Service Based Tool for representing process elements into RAS Structure. This tool will be able to get input parameters relative to each element information (attributes) and generate its RAS mapping to store them into any repository that can read RAS files as input. In addition to process elements, this tool will be able to map any reusable artifact generated from the development process to RAS Structure. Thus, this tool will provide to the users, services to support the management of assets of SPL development, from process to maintenance.

Acknowledgements

Our thanks to Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) and University of São Paulo (USP) for financial support.

References

[1] P. Clements and L. Northrop, Software Product Lines: Practices and Patterns. Addison Wesley Professional, 2002, the SEI series in software engineering.

[2] H. Gomaa, “Designing software product lines with uml 2.0: From use cases to pattern-based software architectures,” in Reuse of Off-the-Shelf Components. Springer, 2006, pp. 440–440.

[3] D. Beuche, “Modeling and building software product lines with pure::variants,” in 16th International Software Product Line Conference-Volume 2. ACM, 2012, pp. 255–255.

[4] R. Flores, C. Krueger, and P. Clements, “Mega-scale product line engineering at general motors,” in Proceedings of the 16th International Software Product Line Conference-Volume 1. ACM, 2012, pp. 259–268.

[5] E. Cirilo, U. Kulesza, and C. J. P. de Lucena, “A product derivation tool based on model-driven techniques and annotations.” Journal of Universal Computer Science (JUCS), vol. 14, no. 8, 2008, pp. 1344–1367.

[6] O. M. Group, “Reusable asset specification,” OMG, 2005. [Online]. Available: http://www.omg.org/spec/RAS/2.2/ [Retrieved: Sep, 2015]

[7] D. d. S. Moura, “Software profile ras: extending ras and building an asset repository,” Master’s thesis, 2013. [Online]. Available: http://www.lume.ufrgs.br/handle/10183/87582

[8] M. Moon, H. S. Chae, T. Nam, and K. Yeom, “A metamodeling approach to tracing variability between requirements and architecture in software product lines,” in 7th IEEE International Conference on Computer and Information Technology (CIT). IEEE, 2007, pp. 927–933.

[9] R. Hong-min, Y. Zhi-ying, and Z. Jing-zhou, “Design and implementation of ras-based open source software repository,” in 6th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD), vol. 2. IEEE, 2009, pp. 219–223.

[10] O. M. Group, “Software & systems process engineering metamodel specification,” OMG, 2008. [Online]. Available: http://www.omg.org/spec/SPEM/2.0/ [Retrieved: Sep, 2015]

[11] L. García-Borgoñón, M. A. Barcelona, J. A. García-García, M. Alba, and M. J. Escalona, “Software process modeling languages: A systematic literature review,” Inf. Softw. Technol., vol. 56, no. 2, Feb. 2014, pp. 103–116.

[12] O. M. Group., “Unified modeling language,” OMG, 2011. [Online]. Available: http://www.omg.org/spec/UML/2.4.1/ [Retrieved: Sep, 2015]

[13] R. S. Corporation, “Rational unified process,” IBM, 1998. [Online]. Available: http://www.ibm.com/developerworks/rational/library/content/03July/1000/1251/1251_bestpractices_TP026B.pdf [Retrieved: Sep, 2015]

[14] I. Elgedawy and L. Ramaswamy, “Rapid identification approach for reusable soa assets using component business maps,” in IEEE International Conference on Web Services (ICWS). IEEE, 2009, pp. 599–606.

[15] N. Zhou, L.-J. Zhang, Y.-M. Chee, and L. Chen, “Legacy asset analysis and integration in model-driven soa solution,” in IEEE International Conference on Services Computing (SCC). IEEE, 2010, pp. 554–561.

 ISBN: 978-1-61208-438-1