ISA Packager: a tool for SPL deployment

Jesus García-Galán
Dept. Computer Languages and Systems
University of Seville (Spain)
jegalan at us dot es

Pablo Trinidad
Dept. Computer Languages and Systems
University of Seville (Spain)
ptrinidad at us dot es

Antonio Ruiz-Cortés
Dept. Computer Languages and Systems
University of Seville (Spain)
aruiz at us dot es

ABSTRACT

In software projects, and particularly in Software Product Line (SPL) projects, product composition and deployment are tasks that are not supported by open source tools. These tasks are repetitive and error-prone. Automation helps on reducing the errors while the productivity increases. In this paper we present a real-world experience through ISA Packager, a generic tool to package and deploy SPLs. In this experience we build a SPL of SCADAs (Supervisory Control And Data Acquisition). Each customized SCADA product evolves in time and ISA Packager is in charge of easing product maintenance and updating.

Keywords

software product line, feature models, deployment, product evolution, error analysis

1. INTRODUCTION

Product composition and deployment are mandatory tasks in any project. Their complexity relies on the project size and its architecture. On *Software Product Lines* (SPLs) [6], these tasks are even more important because an extra effort is usually needed to derive products from assets. Since many companies are adopting SPL, this problem is getting more and more interesting for the community.

In the case study we present in this paper, a local software company is specialized in building *Supervisory Control And Data Acquisition* (*SCADA*) systems. A SCADA system is a computer system that monitors and coordinates infrastructures or processes, generally industrial ones. For instance, a computer system that controls hydroelectric or wind power plants is a SCADA system. The company has already built a SPL to enhance their building process. Installing a basic SCADA software system with only three assets is not difficult. However, SCADAs have around 20 software assets, each of them having several versions. Moreover, each version is composed by many binaries, configuration files and scripts among others. Those artifacts should be packaged, and deployed in several machines with many different configurations. A deployment mainly consists of copying files, daemon installations, ordered script executions and OS-specific tasks. The problem of deployment is complex but it does not end here, because products can evolve, adding, removing or upgrading assets.

Although SPL approach has improved the software production process, packaging and deployment were still handmade and repetitive activities. It is a time-wasting and error-prone approach. We have built *ISA Packager*, a reusable tool to package, deploy and maintain SPLs in general. Using this tool reduces error commitment while time-to-market speeds up.

ISA Packager approaches the problem following four main sub-processes:

1. *Packaging*: taking resources from a version-control system and collecting them into a unique package.

2. *Installation*: the package is deployed and capabilities such as services, daemons or environment variables are set.

3. *Product Evolution*: once the product is operating, new assets can be added or existing ones removed or upgraded to a newer version.

4. *Product validation*: before installing or evolving a product, we must guarantee that a product configuration is valid.

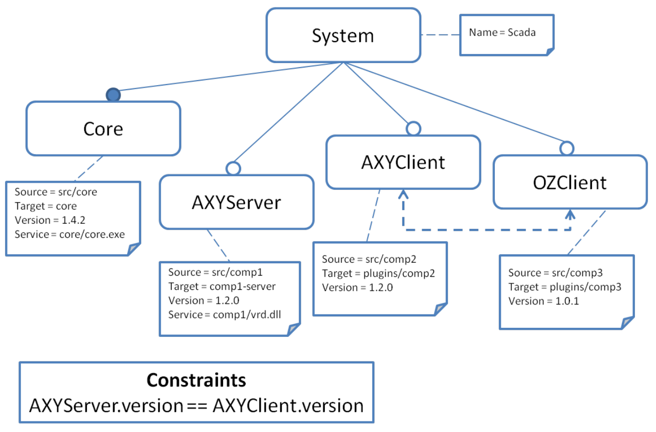

Figure 1: Example of a SCADA feature model

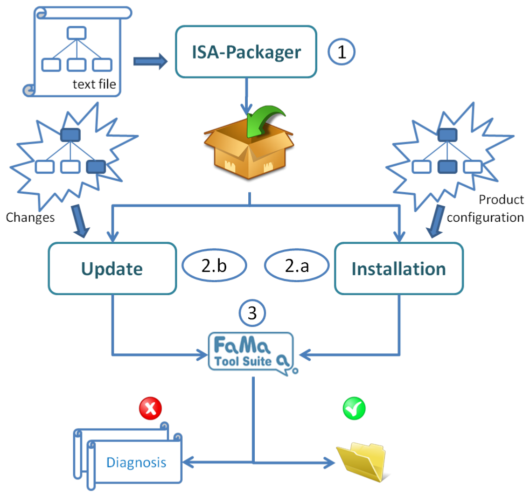

Figure 2: ISA Packager life cycle

As a result of the SPL approach, SCADAs are decomposed into software assets or features whose relationships are described in a *Feature Model* (FM) (Fig. 1). The FM links software artifacts such as files, scripts, daemons or environment variables to features. ISA Packager takes a FM and a configuration for a specific product as inputs. Firstly, ISA packager validates the product configuration by means of *FaMa Framework* [15], a feature model analysis tool. In case any error is detected, explanations are provided to aid the user in correcting the FM or the configuration. If inputs are error-free, artifacts are packaged and deployed in a system. The deployment information is kept to be used in further maintenance operations such as software updates or product reconfigurations.

Although the company has project-specific requirements such as Windows XP and Ubuntu compliance or plain input text file-formats for FMs, we have faced each sub-process in general terms so our solution can be reused in other projects of the same nature. Therefore, our solution is approached in two levels: a generalization level, where we solve the problem for SPLs in general, and a specialization level, where the solution is adapted to our particular project. Both levels are discussed in Section 2. A brief review for the related work can be found in Section 3. Finally, Section 4 presents our conclusions and how we envision ISA Packager will evolve to widen its scope and to apply it in other contexts.

2. ISA PACKAGER

2.1 Architecture overview

ISA Packager is a Java tool to package and deploy SPL assets in general, following the process depicted in Figure 2. Firstly, the tool receives a FM describing a SPL and a configuration as inputs. ISA Packager obtains information about assets to package the product into a unique file. This file has the ability of self-installing a product or updating a previous installation. Prior executing the installation process, ISA Packager validates the resulting configuration for containing no errors such as feature incompatibilities or dependences.

It has a modularized architecture, composed by five pieces, as depicted in Figure 3. *Packager* module packages a product into a file. *Installer* is the module in charge of installing and updating products. *Model* module transforms input files into extended feature models. To validate products, ISA Packager uses *FaMa Framework* [15], an external tool for SPL analysis, through the adapter module *ISAPackager2FaMa*. Finally, the tool provides for a command-line user interface (UI). These components are compiled into a single and distributable binary using *Maven*[1].

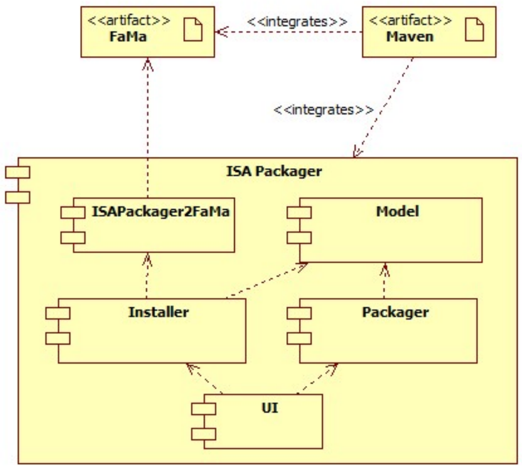

Figure 3: ISA Packager architecture

2.2 Packaging

Packaging is the first process in ISA Packager (figure 2, point 1). The tool receives an extended feature model as input. Currently, the FM is described in a single text file, with a proprietary format known as *ISA Installation Format* (iif). Features represent system assets, and each attribute represents relevant information about an asset, like versioning, or target or source folders. Moreover, we can define constraints over features and/or attributes. In Figure 1 we can see a sample input model with four fictional leaf features, a parent feature, and an exclusion constraints. Together with a model, ISA Packager has to be provided for a root location, and optionally the security credentials. As a result, ISA Packager processes the model and builds a custom installer.

2.3 Installation and Updating

Product installation is the second step on the life cycle (figure 4, point 2.a). A user can choose for a set of features to install using a command-line user interface. Installer translates each selected feature into a set of executable *commands*, through an analysis of features and their attributes. We classify commands in four types:

- *File*: copying files into a specified location.
- *Environment variable*: setting or modifying environment variables in a system.
- *Daemon*: installing a process as a daemon. Several processes in SCADAs have to run whenever the system is active.

- *Script*: executing scripts after or before another command execution. They are generally used for feature updating to remove files or to uninstall daemons.

The tool has a stack of command instances for each of the above kinds. All the commands in a stack are executed following the order depicted in Figure 4. For installation, environment variables commands are firstly executed. Then, for each asset, files are copied. If it is required daemons are installed and auxiliary scripts executed.

Figure 5 depicts a selection of features from the model in Figure 1. Three features (on the left) are selected for installation -system feature is always selected-. On the right side, we find the kind of commands to be executed for each feature selection. Firstly, an environment variable is set to save installation location. Core is deployed, copying files and installing a daemon. Finally, OZClient installation just implies a File command.

When a previous installation is detected in the system where a product is to be deployed, product updating process is launched instead of installation(figure 2, point 2.b). The most basic updating consists of adding new features. However, it usually implies removing features or updating an existing one which implicitly consists of removing an old version of a feature and adding a newer one. Therefore, the main difference between updating and installation is the ability of removing features, which implies undoing all the commands that were executed during installation. For this purpose, we have defined deletion commands for each of the installation commands:

- *File*: deleting files.
- *Environment variable*: unsetting or restoring a variable.
- *Daemon*: stopping and uninstalling a daemon.
- *Script*: executing an uninstall script

An opposite execution order is adopted for feature removal, as indicated in Figure 4 by the*uninstallation* arrow.

The above considerations are the result of a generalization of the problem for SPLs in general. For our particular case study, installers should support Windows XP and Ubuntu operating systems, so commands have been adapted to them. For instance, setting an environment variable in Windows XP, can be done by a command line tool. However, in Ubuntu you have to write into a variables file.

2.4 Validation

When we configure a product, even more on complex SPLs, we could choose an invalid set of features. A configuration if invalid if it violates at least one of the constraints represented within the FM of the SPL. In case a product is invalid, it cannot be installed, so the configuration must be validated before installation. At present, there are available several tools to analyze feature models that can be used to check if a configuration is valid for a given FM. We have chosen *FaMa Framework* [15] because it is an open-source tool and is able to analyze feature models containing attributes.

FaMa contributes to ISA Packager with two analysis operations: *Product validation* which analyses if a final product is valid for a given FM; and *Product explanation* which returns a set of explanations to assist on configuration restoration.

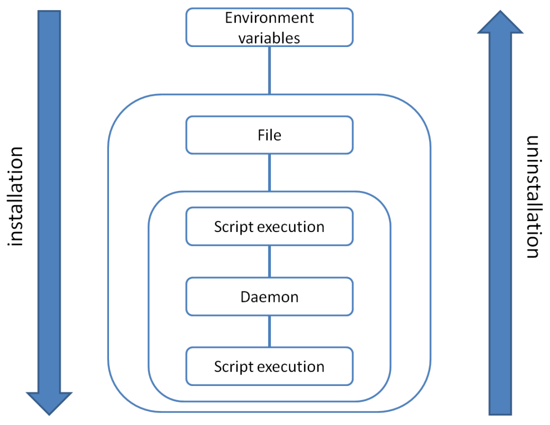

Figure 4: Commands execution for installation and uninstallation

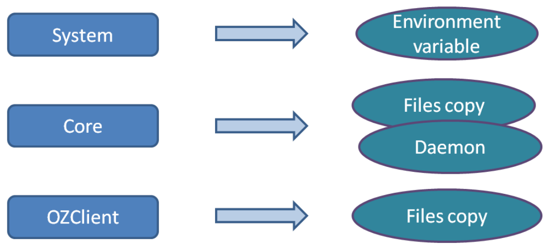

Figure 5: Selection of features to install, and translation to actions

For instance, if we try to install all features of the sample SPL in Figure 1, the resultant product is not valid since *AXYClient* and *OZClient* are mutually exclusive. FAMA informs that the configuration is invalid and provides for explanations such as 'remove *OZClient* feature from the product' or 'remove *AXYClient* from the product'.

However, product explanation has not a straightforward application for updating. When installing, we only have two choices to repair an error: to select or unselect features. But when updating, we can choose among several versions to install. At date, FaMa Framework is not able to work with versions of the same feature for explanations, but it can for product validation. We are working to incorporate recent work on this field, such as *Vierhauser et al.* [16]

3. RELATED WORK

Product deployment in general, is still an open issue on many organizations. There exist general-scope deployment tools that can be used for this purpose. This is the case of *Maven* [1] or *Cargo* [2], both open- source tools to assist product compilation and deployment. There are also tools which are specialized in software installation, such as *IzPack* [3] or *Nullsoft Scriptable Install System (NSIS)* [4], which are able to create customized and cross-platform installers.

Regarding SPLs, we can find product derivation approaches. Product derivation is the branch of Software Product Line Engineering (SPLE) that affronts product composition, integration and deployment from a SPL. We refer to three tools for product derivation:

- *AHEAD* tool suite, based on AHEAD methodology[5], supports the customization of the building process for feature-oriented SPLs.

- *GEARS* (*Krueger et al.*) [10] is a commercial tool to configure and derive products from a SPL.

- *COVAMOF* [7] [14] is a tool to define variability models. It provides for a Visual Studio plugin which permits deriving products, among other options.

Two product derivation approaches should be specially remarked: *Pro-PD* (Process reference model for Product Derivation) (*O'Leary at al.*) [11] [13], a process reference model for product derivation, developed at Lero; and *DOPLER Ucon* (*Dhungana et al.*) [8] [13] is a tool-supported product derivation approach, driven by industry needs developed at JKU, Linz. Related to DOPLER, *Grunbacher et al.* [9] discussed about using *Eclipse* to take advantage from its plugin architecture and deployment system.

For more information about existing approaches to product derivation, *Rabiser et al.* [12] is an excellent survey on this topic.

4. CONCLUSIONS AND FUTURE WORK

ISA Packager provides for a way to package SPL assets into an installer, and deploy and update products. The major contribution of ISA Packager is supporting product evolution, by means of updating existing products, thanks to the links established between features and de/installation commands. However, some limitations still exist that we plan to overcome in the next future:

- Supporting remote deployment.

- Using a graphical UI that substitutes the current command-line user interface.

- Updating validation, providing explanations when several versions can be installed and the configuration is invalid.

- Widen the kinds of installation commands, going further than daemon installation, environment variables or script executions.

- Integrating multiple asset versions on the same packager.

- Splitting input files in two files, one describing features tree -problem domain- and other describing feature attributes and constraints -solution domain-.

At the end of our project, we plan to release ISA Packager as an open-source tool under GPL v3 license.

5. ACKNOWLEDGMENTS

This work has been partially supported by the European Commission (FEDER) and Spanish Government under CICYT project SETI (TIN2009-07366), and by the Andalusian Government under ISABEL project (TIC-2533).

6. REFERENCES

[1] Apache maven. `http://maven.apache.org/`.
[2] Cargo. `http://cargo.codehaus.org/`.
[3] Izpack. `http://izpack.org/`.
[4] Nullsoft scriptable install system. `http://nsis.sourceforge.net/`.
[5] D. Batory, J. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 187 – 197, May 2003.
[6] P. Clements and L. M. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.
[7] S. Deelstra, M. Sinnema, and J. Bosch. Variability assessment in software product families. *Information and Software Technology*, 51(1):195 – 218, 2009. Special Section - Most Cited Articles in 2002 and Regular Research Papers.
[8] D. Dhungana, R. Rabiser, P. Grünbacher, and T. Neumayer. Integrated tool support for software product line engineering. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 533–534, 2007.
[9] P. Grunbacher, R. Rabiser, D. Dhungana, and M. Lehofer. Model-based customization and deployment of eclipse-based tools: Industrial experiences. In *Automated Software Engineering, 2009. ASE '09. 24th IEEE/ACM International Conference on*, pages 247 –256, 2009.
[10] C. Krueger, D. Churchett, and R. Buhrdorf. Homeaway's transition to software product line practice: Engineering and business results in 60 days. In *Software Product Line Conference, 2008. SPLC '08. 12th International*, pages 297 –306, 2008.
[11] S. T. Padraig O'Leary, Fergal McCaffery and I. Richardson. An agile process model for product derivation in software product line engineering. In *Journal of Software Maintenance and Evolution: Research and Practice*, 2010.
[12] R. Rabiser, P. Grünbacher, and D. Dhungana. Requirements for product derivation support: Results from a systematic literature review and an expert survey. *Information and Software Technology*, 52(3):324 – 346, 2010.
[13] R. Rabiser, P. O'Leary, and I. Richardson. Key activities for product derivation in software product lines. *Journal of Systems and Software*, In Press, Corrected Proof:–, 2010.
[14] M. Sinnema and S. Deelstra. Industrial validation of covamof. *Journal of Systems and Software*, 81(4):584 – 600, 2008. Selected papers from the 10th Conference on Software Maintenance and Reengineering.
[15] P. Trinidad, D. Benavides, A. Ruiz-Cortes, S. Segura, and A. Jimenez. Fama framework. In *Software Product Line Conference, 2008. SPLC '08. 12th International*, pages 359 –359, 2008.
[16] M. Vierhauser, P. Grünbacher, A. Egyed, R. Rabiser, and W. Heider. Flexible and scalable consistency checking on product line variability models. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pages 63–72, 2010.