Run-time Variability Issues in Software Product Lines

Alexandre Bragança[1] and Ricardo J. Machado[2]

[1] Dep. I&D, I2S Informática – Sistemas e Serviços SA, Porto, Portugal,
alexandre.braganca@i2s.pt
[2] Dep. Sistemas de Informação, Universidade do Minho, Guimarães, Portugal,
rmac@dsi.uminho.pt

Abstract. The Product line approach promises productivity and flexibility gains through reuse. In order to achieve higher levels of productivity and flexibility, companies may need to adopt run-time variability realization techniques. However, such an approach can raise issues that companies need to face in order to fully implement run-time variability in their product lines. This is the case of I2S; a Portuguese software-house specialized in software solutions for the insurance industry. In this paper, we present and discuss two major run-time variability issues that were identified in the insurance domain. We also present our approach to solve these issues and how it is being implemented in the I2S product line. We also relate our approach to other techniques documented in the field literature.

1 Introduction

Product families and product lines aim to promote reusability within a given set of software products [1]. Software product lines have achieved substantial adoption by the software industry. The adoption of product line software development approaches has enabled a wide variety of companies to substantially decrease the cost of software development, maintenance, and time to market and increase the quality of their software products [2].

To accomplish reusability among various software products there must be common characteristics among them. Normally, this means that the various software products must share the same domain. Therefore, an organization that has built several software systems in a domain has also acquired very good knowledge of such a domain. This knowledge can be used when building new software systems in the same domain. Nonetheless, to build diverse software systems within a domain we also need to specify variability, i.e., what can vary between the applications of a product line. Feature diagrams have been proposed as a way to model variability [3].

When building an application in a domain, the architecture of the application is an instance of a reference domain architecture, and it uses software components that fit the architecture. The software components can be selected from the features of the new application. Common features imply common components that are used in all applications in the domain. Variable features require support for variability. In order to implement the application variants, modelled by feature diagrams, variability realization techniques are needed.

Variability realization techniques are usually applied at pre-deployment time. By this we mean that the variation point is introduced and is bound to a variant in stages of the development cycle that precede deployment. These techniques limit the degree of variability of a product line because the introduction of new variation points or the binding of new variants cannot be made after deployment [4]. There are commonly used and documented run-time variability realization techniques that can help solving these limitations. Usually, the industry adopts run-time variability realization techniques based on standard commercial component infrastructures such as CORBA and COM. Nonetheless, these techniques still present issues that, in some cases, can become serious limitations. We will present such an example in the case of I2S; a Portuguese software house specialized in the development of software solutions for the insurance industry.

In this paper we will present an industry case of application of run-time variability techniques in a product line. Based on this case, we will identify and discuss run-time variability issues that are not well addressed by traditional run-time variability realization techniques. Based on the experiences resulting from the industry case, we will propose new approaches to run-time realization techniques.

The remainder of this paper is organized as following. The next section presents the case study of I2S and gives a brief description of the particularities of the insurance domain. Section 3 is dedicated to the discussion of the run-time variability issues that can emerge in domains such as the domain of the case study. Section 4 presents our proposed approach to solve these issues. In section 5 we give some insight in how we realize our approach in the case of I2S. In section 6 we present related work of other authors and in section 7 we conclude the paper.

2 The I2S Case

This section presents the case of I2S Informática – Sistemas e Serviços SA. I2S was founded in 1985 and during the following years it became specialized in the development of software products and solutions for the insurance market. It operates mainly in Portuguese spoken language countries but has also other markets like Poland. The company has more than 100 employees and about half of them work in the development of software products. One of the authors of this paper has been collaborating with this company, mainly with the recently created research and development sector, helping in finding and developing technical solutions for its problems.

I2S has developed a line of products aimed at the insurance market. During all these years, the company has faced and solved several problems regarding variability. The company applied some of the most common variability realization techniques like pre-processing of source code, linking directives, parameterisation, inheritance, code composition, dynamic library loading and component infrastructures (i.e., COM). There are, however, characteristics of the insurance domain that raise some new run-time variability issues that the company has to face.

There are some cases in which solving variability at compilation-time and even at deployment-time is not a satisfactory solution. This is the case of the I2S, and its product line aimed at the insurance industry.

Business products in the insurance market are very complex. A new insurance product may imply that new and variable data may be necessary to be registered for each new insurance policy of that product. New rules for risk assessment and claim processing may also be needed. These are some of the many variability points that such a system may need to cope with in the case of the market of a new insurance product. We can say that such a software system may have a significant change in behaviour for each insurance product.

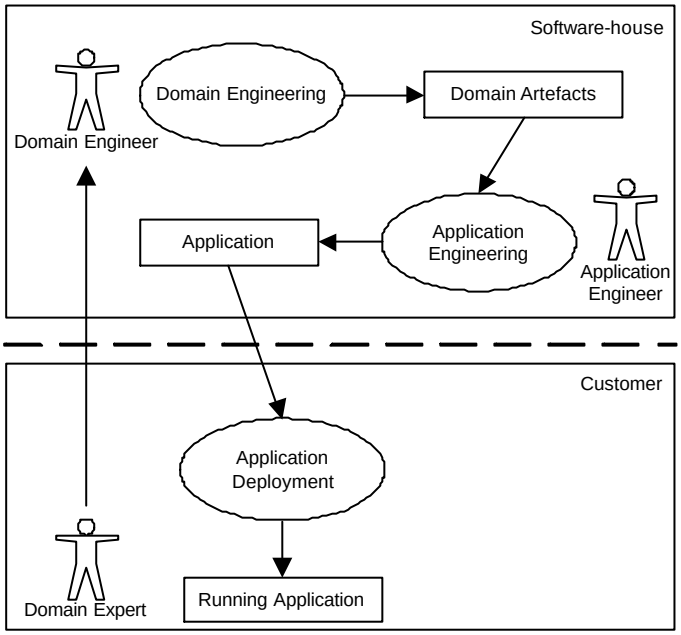

Fig. 1. Simplified product line engineering process

If, for instance, a company like I2S has 10 insurance companies using its software system and an average of 20 insurance products per company, in the case of software variability solved at compilation-time, we may end up with 10*20=200 variations of the software system.

Even if, technically, one could manage such a number of diverse versions of the system, a significant problem remains: when an insurance company needs to market a new insurance product, this may imply changes in the requirements of the application. According to figure 1, such changes need to be communicated to the domain engineer in order to be incorporated in the product line engineering process. This will encompass time to an engineering process and time to deploy the new application. Such a time frame may not be feasible because of the constraints of the time to market of the new insurance product. Thus, this case will typically benefit from run-time variability realization techniques if their adoption can provide support to avoid the traditional

software engineering process that may be necessary each time there is a change in an insurance product or a new insurance product is introduced.

For variability to be solved at run-time, the intervention of the application end-user may be necessary. The end-user intervention can be as simple as a run-time feature selection, for instance, by means of an option selection in a configuration screen of the application. More complex scenarios may also exist, namely in very dynamic domains, as the one presented in this section. In these cases, it may be necessary for the end-user to complete the functional specification of application's features at run-time. This functional completion ought to be made, if possible, by using a language as close as possible to the domain language. This requirement will allow an end-user with domain and system (application) knowledge to complete the functional specification of dynamic features. Clearly, the end-user who is going to complete the application functionality should also be a domain expert.

In the next section, we will present issues regarding runtime variability implementation techniques that we are facing in the I2S case. These issues will be presented in the context of domain engineering and product lines. We will also present our approach to deal with these issues.

3 Run-time Variability Issues

Feature modelling is a widely adopted technique used to specify feature variability (and commonality) in domain engineering and product lines [5]. According to [1], a feature is a "unit of behaviour that is specified by a set of functional and quality requirements". As such, features (at least capability features) are concepts normally very close to the end-user language. Domain analysis based on feature modelling has many benefits. Several methods, such as Feature Oriented Domain Analysis [3], use capability features in domain analysis and modelling. Since features are specified using terms very close to the end-user language, feature diagrams are a very good technique for domain knowledge interchange between end-user and developers. With the right mapping and dependency rules between capability features and other layers of features (technical and development features), it is possible to close the usual gap that exists between the problem space and the solution space in a software engineering process [6].

In domain engineering, the aim is to develop domain reusable artefacts [6]. These artefacts can then be reused in the development of new applications in the domain (see figure 1). The usual outcomes of domain engineering are software architectures and software components. Software architectures model the domain systems in terms of computational components and connections among those components [7]. The development of the computational components is also an objective of domain engineering. As we will see later, the result of that activity can be traditional software components (i.e., COM or CORBA components) or other artefacts like, for instance, domain-specific languages. The range of possible applications in the domain can be described using feature diagrams. An application of the domain will usually correspond to a selection of features of the domain (i.e., a subset of possible features of the domain). As such, application engineering can use feature selection to capture the

software components that can be reused in the development of a new application in the domain. In this manner, user requirements are "translated" into feature selections that help in identifying possible reusable components. These feature selections can also be used to select an adequate architecture for the application (from reference domain architectures). So, features are used to model variability at domain engineering level and remove that variability at application engineering level, when developing a particular application of the domain.

One major issue is raised when realizing feature variability at the application level. Given the fact that variations in applications can have diverse binding times, there are, in fact, very limited proposed approaches to implement variability at run-time. Research has been done widely with pre-deployment techniques. For instance, GenVoca allows a very good mapping between features and implementation of variability but it is supported by compile-time composition of features [8]. Other techniques, like generative programming, are based on static programming, i.e., they use techniques like C++ templates to compose features [9]. These academic techniques seem well suited as variability realization techniques but have a serious limitation: they are usually not usable at run-time. Component infrastructures are commonly used by the industry, and can be used as run-time realization techniques, but, comparing with the academic techniques, they have limited characteristics. Several authors recognize the limitations of these techniques [9]. They also seem to agree that it should be possible to specify components and their composition, manipulation and modification at different times.

In our approach to run-time variability implementation techniques we see two (almost) orthogonal issues:

1. how to combine features at run-time, i.e., how to enable user selection of features at run-time;
2. how to enable behaviour specification *holes* in the applications for the user to fill at run-time.

4 Proposed approach

Static Run-time Variability. The first issue mentioned above regards techniques for run-time feature selection and composition. I2S, and the software industry in general, already use techniques such as dynamic library loading and component infrastructures to implement this kind of run-time variability. So, what is the problem with these techniques? We shouldn't say that there is a problem, but that there are limitations. Our point regards the fact that the kind of feature selection and composition achieved is very limited compared to the possibilities that compile-time techniques, like GenVoca, offer.

To reach the level of capabilities offered by compile-time techniques regarding feature selection and composition, run-time environments for applications need also to evolve. For instance, GenVoca offers the capability of layer composition by using a compile-time technique similar to inheritance. The interesting characteristic of this technique is that it was conceived to apply *inheritance* into components that can be composed from multiple classes. Each component (layer) implements some feature. To offer this kind of capabilities at run-time we propose an approach based on extend-

ing the run-time capabilities of the operating run-time environment. Our argument is that if the operating run-time environment exposes capabilities similar to the ones existing at build-time, feature selection and composition similar to the one made at compile-time should also be feasible. At this initial exploring phase of our work, we propose that the operating run-time environment should be extended with reflection and run-time compilation. We assume that, for the first issue of our approach to run-time variability, the application user will only execute feature selection and composition, and so, there isn't a crucial need for a high-level feature composition language. At present, it seems satisfactory to have a composition language similar to the one used with GenVoca. In this way, we advocate that it is possible to extend the usual plug in model of run-time variability adopted by applications into an extended plug in model with similar characteristics to the ones found in compile-time variability realization techniques.

Dynamic Run-time Variability. Regarding the second issue, the context in which feature specification is done suffers a very significant change. In fact, what we are advocating in this issue of run-time variability is that domain engineers and application engineers will leave 'holes' in their specification for post-deployment completion (more precisely run-time completion). This is a scenario where one or more features do not have possible implementations specified before run-time. We call application 'holes' these features that don't have implementation before run-time. This only occurs in very dynamic domains like the I2S case. In this context, application users will have to play a special role because they will have to fill the holes left in application engineering. This filling of holes corresponds to implementing features. In order to achieve this, the application user will need to do some kind of program specification. This entails several implications: (1) the application user needs to assume a special role; (2) development methods may have to be adapted to support such need; (3) a feature implementation/specification language has to be developed, or adapted, for this special application user.

So far, our work suggests that this role of run-time specification/implementation of features has to be played by a domain expert that also needs to be an application expert. We suggest the name of *application domain specialist* for this new role. This role is different from that of a domain engineer and is also different from an application engineer. An *application domain specialist* is a domain expert that is also an application expert, i.e., he/she can adapt/program an application in a post-deployment time (usually run-time). An application engineer is someone who builds applications in a domain using domain artefacts.

An *application domain specialist* is neither a programmer nor a software engineer. He is a domain expert that also needs to have a very high application expertise. Since run-time variability can imply run-time feature programming, this means that the *application domain specialist* needs to have some kind of programming skill. This should not be like a generic programming skill but more like a domain programming logic skill. An *application domain specialist* should be able to use a domain-specific language with grammar and semantic very close to the *natural* domain language and concepts. In our work, we propose the adoption of domain-specific languages as a mean to specify the implementation of features at run-time. This means that a domain with run-time feature implementation needs a domain-specific language. Therefore,

the domain engineering process needs to produce a domain-specific language. Concerning this aspect, we propose that the development of domain-specific languages should be based on features and feature diagrams. In fact, the adoption of feature diagrams to support the development of domain-specific languages has already been documented [10].

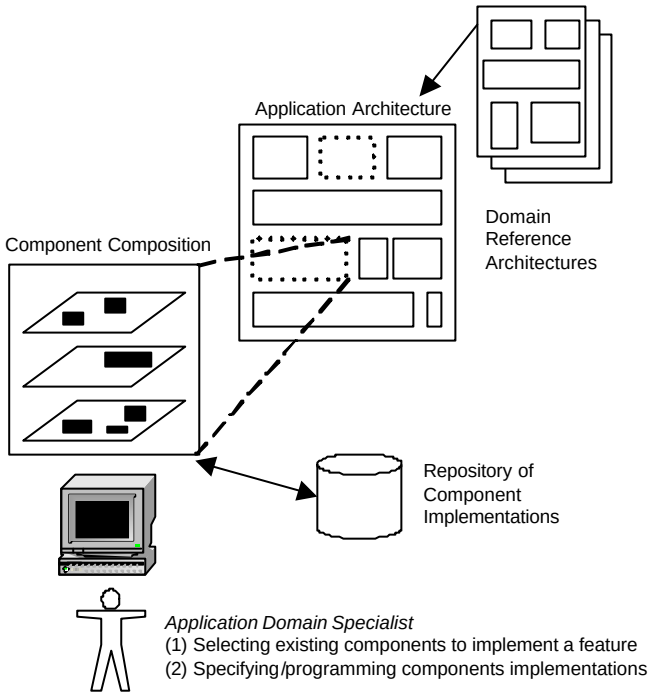

Fig. 2. Proposed approach to run-time feature implementation using variability realization techniques.

So far, our study indicates that run-time features could be implemented (in a majority of cases) by very simple programming logic connecting existing high-level domain concepts, which in turn, are also features of the domain. If this is true, then feature diagrams could be used to extract the design requirements of the domain specific language and also to design a library of resources to be used in the language programs (i.e., components).

Figure 2 is a schematic representation that resumes our approach to run-time variability. It represents the role of the *application domain specialist* in the context of the run-time variability issues of the application: the selection/composition of features and the implementation of components/features.

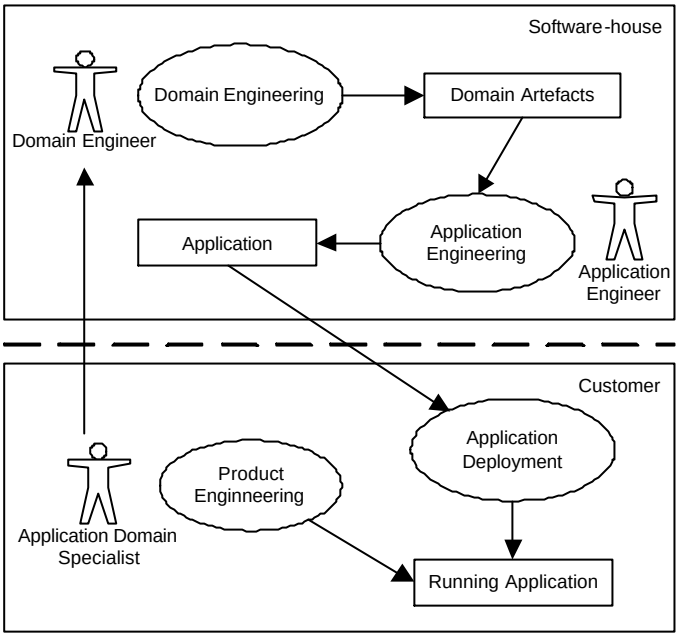

Fig. 3. Business product engineering in the software product line process.

5 Proposed Approach Realization

In the previous section, we presented our technical proposal for two run-time variability issues that can be raised in very dynamic business domains. Our technical proposals are based on our experience within I2S and the insurance domain.

The technical solutions we presented must be applied in the context of a particular business domain. To do so, the activities of the application domain specialist, presented in figure 2, must be done in some business domain context. In the insurance domain, this context is provided by the insurance product engineering activity. This activity must be supported by the application at run-time. Insurance products represent the functional variability of the application. The application should support multiple insurance products. As presented in figure 3, insurance products are developed by the application domain specialist. Only in the case where the variability provided by the concept of insurance product is not enough, the application domain specialist must communicate the limitation to the domain engineer in order to be incorporated in the application using the regular engineering process.

The insurance product development process requires support very similar to a regular software development process. In fact, each insurance product will be like a 'program' within the insurance application. As such, tools and mechanisms to support product debug, test or logging should be present. The presentation of techniques and methods to support the development of such tools is not in the scope of this paper.

We have developed a run-time system (domain-specific language + run-time execution machine) based on our approach that is being used at I2S. This first version was incorporated in the I2S product line. It is being used in some customers of I2S to support the commercialisation of insurance products via web-based channels. The insurance products are engineered at the back office of the insurance companies (using one of the applications of the I2S product line) and deployed to a web server. In the web server, another application of the I2S product line uses the run-time system to support the variability of each published insured product.

6 Related Work

There are documented examples of successful development and adoption of domain-specific languages [11]. RISLA is an example of a domain-specific language that was used (and according to the authors remains in use) to specify financial products in a bank [12]. In RISLA, the specifications of financial products are transformed into COBOL code, which is compiled and integrated in the bank system. Our approach significantly differs from RISLA because we propose that the specification of variability (i.e., insurance products) be made at run-time, without the need of the traditional software development process. The need of this traditional development process also limits the adoption of an approach similar to RISLA in a scenario like the one of figure 3. In fact, our approach aims to support run-time variability in a product line of a software-house and, to our knowledge, RISLA was only applied as a bank in-house developed application.

There is limited scientific literature regarding insurance software systems. Nonetheless, some analysis and design patterns regarding the insurance domain have been presented [13, 14]. These patterns focus on the specific design requirements that insurance applications have. Especially the patterns present approaches to the representation of insurance domain entities and documents. There are also patterns regarding the realization of the insurance business product concept. These patterns are also based domain-specific languages and run-time support. However, to our knowledge, the authors don't provide further information regarding on how to develop such artefacts. They also don't clarify if (and how) such patterns can be applied in a product line approach such as the one presented in this paper. There are also no references to practical applications of the patterns so it becomes difficult to compare them to our work.

7 Conclusions and Future Work

The Product line approach promises productivity and flexibility gains through reuse. In order to achieve higher levels of productivity and flexibility companies can adopt run-time variability realization techniques. However, such an approach can raise several issues that companies need to face in order to fully implement run-time variability in their product lines. We have presented the variability techniques used by the in-

dustry and their limits regarding the run-time variability issues that a company may face.

Two major issues regarding run-time variability were identified and characterized in the context of our experience at I2S. We have also presented our approach to solve the identified run-time issues.

As part of our future work, we plan to further validate our approach in the I2S product line. In the context of our work, I2S has developed a domain-specific language and a run-time infrastructure. Application domain specialists already use this language as a tool to specify several business rules within the I2S product line. Hence, this seems to be a very rich and representative run-time variability case to experiment our approach. The results from this work will help us to further understand the run-time variability implications in product line's development processes.

References

1. Bosch, J., *Design and Use of Software Architectures Adopting and Evolving a Product-Line Approach*. 2000: Addison-Wesley.
2. Bosch, J. *Maturity and Evolution in Software Product Lines: Approaches, Artifacts and Organization*. in *Second Software Product Line Conference (SPLC2)*. 2002: Springer-Verlag.
3. Kang, K.C., et al., *Feature-Oriented Domain Analysis (FODA) Feasibility Study Technical Report*. 1990, Software Engineering Institute, Carnegie Mellon University.
4. Jaring, M. and J. Bosch. *Representing Variability in Software Product Lines: A case study*. in *Second Software Product Line Conference (SPLC2)*. 2002: Springer-Verlag.
5. Griss, M.L., J. Favaro, and M. d'Alessandro. *Integrating Feature Modeling with the RSEB*. in *Fifth International Conference on Software Reuse*. 1998. Victoria, Canada: IEEE Computer Society Press.
6. Kang, K.C., et al., *FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures*. Annals of Software Engineering, 1998. 5: p. 143-168.
7. Jacobson, I., M. Griss, and P. Jonsson, *Software Reuse: Architecture, Process and Organization for Business Success*. 1997: Addison Wesley Longman.
8. Batory, D., R.E. Lopez-Herrejon, and J.-P. Martin. *Generating Product-Lines of Product-Families*. in *ASE'02*. 2002. Edinburgh, Scotland: IEEE Computer Society.
9. Czarnecki, K., *Generative Programming Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models*, in *Department of Computer Science and Automation*. 1998, Technical University of Ilmenau.
10. Deursen, A.v. and P. Klint, *Domain-Specific Language Design Requires Features Descriptions*. Journal of Computing and Information Technology, 2002. 10(1): p. 1-17.
11. Deursen, A.v., P. Klint, and J. Visser, *Domain-Specific Languages: An Annotated Bibliography*. ACM SIGPLAN Notices, 2000. 35(6): p. 26-36.
12. Deursen, A. and P. Klint, *Little languages: Little maintenance?* Journal of Software Maintenance: Research and Practice, 1998. 10(2): p. 75-92.
13. Keller, W. *Some Patterns for Insurance Systems*. in *Pattern Languages of Programs 98*. 1998.
14. Johnson, R.E. and J. Oakes, *The User-Defined Product Framework*. 1998.