Architectural Evolution of a Software Product Line: an experience report

Marcelo Schmitt Laser, Elder Macedo Rodrigues, Anderson Domingues, Flávio Oliveira, Avelino F. Zorzo
School of Computer Science (FACIN) - Pontifical Catholic University of Rio Grande do Sul
Porto Alegre – RS – Brazil
Email: {marcelo.laser, anderson.domingues}@acad.pucrs.br
{elder.rodrigues, flavio.oliveira, avelino.zorzo}@pucrs.br

***Abstract*—This work presents an experience report on the architectural decisions taken in the evolution of a Software Product Line (SPL) of Model-based Testing tools (PLeTs). This SPL was partially designed and developed with the intention of minimizing effort and time-to-market during the development of a family of performance testing tools. With the evolution of our research and the addition of new features to the SPL, we identified limitations in the initial architectural design of PLeTs' components, which led us to redesign its Software Product Line Architecture (SPLA). In this paper, we discuss the main issues that led to changes in our SPLA, as well as present the design decisions that facilitate its evolution in the context of an industrial environment. We will also report our experiences on architecture modifications in the evolution of our SPL with the intention of allowing easier maintenance in a volatile development environment.**

I. Introduction

Over a few decades, more and more software development companies have been using some software engineering strategies, such as reuse-based software engineering, to develop software with less cost, faster delivery and increased quality. Reuse-based software engineering is a strategy in which the development process is focused on the reuse of assets and on a core architecture, reducing the development effort and improving the software quality. In recent years, many techniques have been proposed to support software reuse, such as Component-based development and Software Product Lines (SPL) [11]. Component-based development is centered on developing a software system by integrating components, where each component can be defined as an independent software unit that can be used with other components to create a system module or even a whole software system. In another way, Software Product Lines are focused on developing a family of applications based on a common architecture and a shared set of software assets, where each application is generated, in accordance with the requirements imposed by different customers, from these assets and shares a common architecture [11].

In past years, SPL has emerged as a promising technique to achieve systematic reuse and at the same time to decrease development costs and time-to-market [9]. One of the main SPL development sub-processes is to use the domain requirements and the product line variability model to define the Software Product Line Architecture (SPLA). The SPLA is a common, high level and generic structure that will be used for all the products derived from the SPL.

In order to take advantage of this approach, we have adopted the SPL concept to support the development of our applications [1] [4] [10]. We have found that, although this enabled the reuse of artifacts, thus reducing the time and cost of development, it incurred in a cost related to the evolution of each artifact, as well as that of the SPLA used to manage this evolution.

In this work we report and discuss our experience in implementing and evolving the architecture of a component-based SPL to derive Model-based Testing (MBT) tools. In particular, we describe how we applied software design patterns [6] to map and to instantiate components, these having their variability managed by another component [5]. Finally, we describe two methods of implementing the variable components (features).

This paper is organized as follows. Section II describes the context where PLeTs SPL was designed and developed, as well as briefly presenting its Product Line Architecture (PLA). In Section III we discuss the main PLA limitations identified along the SPL evolution. In Section IV we present and discuss our approach to mitigate these limitations, as well as describe our PLA in accordance with that approach. In Section V we discuss the related work and Section VI presents the lessons learned along the PLA evolution. Finally, the conclusion and the future work are presented in Section VII.

II. Context

Our research group on Software Testing[1] has been working to design and develop several testing tools. Our research focus is to investigate innovative ways to mitigate the effort of repeatedly creating custom solutions to apply performance, functional and structural testing. After developing several testing tools, either from scratch or using a limited opportunistic reuse, but which had several features in common, we started a collaborative study with the Technology Development Lab of our partner company to investigate the use of SPL concepts to generate these testing tools. As a result of this study we consolidated our SPL called PLeTs [1] [4] [10]. In this SPL, derived products are testing tools that receive behavioural models as input and give either automatic or manual test scripts as output; these models denote specific test cases, thus leveraging testing teams to follow a model-based approach [14], a process that we describe elsewhere [1].

[1] www.cepes.pucrs.br

(DOI Reference Number: 10.18293/SEKE2015-057)

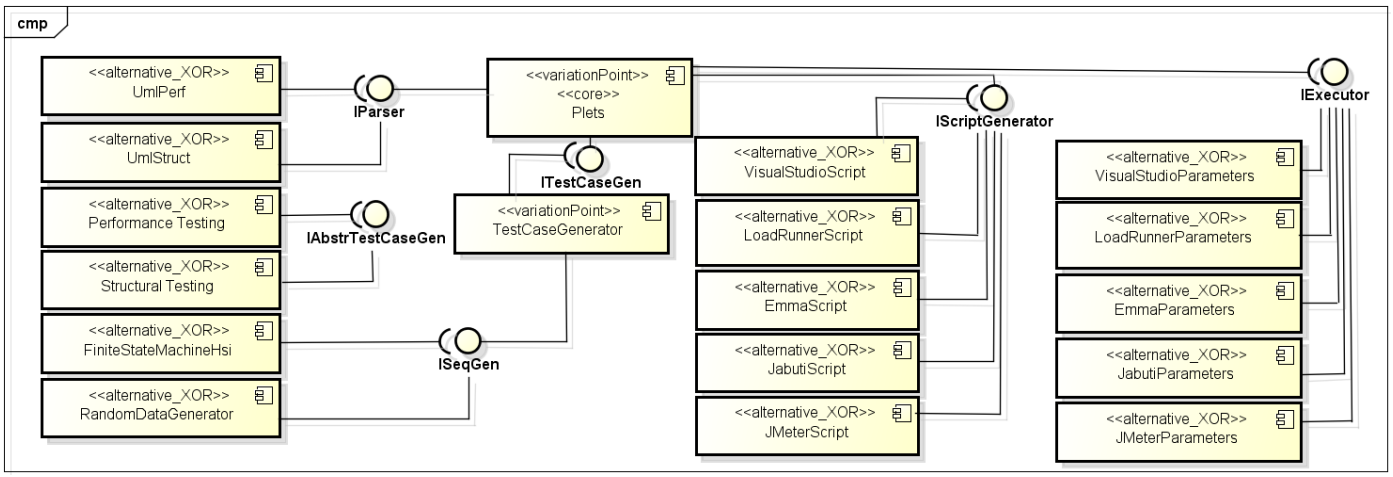

Fig. 1. PLeTs UML Component Diagram

A. PLeTs Architecture

PLeTs was initially designed to support the derivation of a particular testing tool from a set of shared software components, which are then glued together with minimal changes. We defined the use of a replacement mechanism to develop each concrete feature [13] of the PLeTs feature model [1]. In this way, an MBT tool derived from PLeTs is assembled by selecting a set of components and a common software base. We chose this approach to generate PLeTs products because it presents some advantages, such as high-level of modularity and a simple 1:1 feature to code mapping. Figure 1 shows the PLeTs component model.

Since we are using a component replacement mechanism, each provided interface represents a variation point and each variable component implementation represents a variant. The interfaces provided by the *PLeTs* components are (see Figure 1): *a*) *IParser*: this interface is a mandatory variation point that has two exclusive variant components, *UmlPerf* and *UmlStruct*; *b*) *ITestCaseGen*: this interface is a mandatory variation point that has one mandatory component: *TestCaseGenerator*. This component provides two interfaces: *IAbstractTestGen* and *IseqGen*. The former interface can be realized by one of the following components: *PerformanceTesting* or *StructuralTesting*. The latter interface can be realized by one of the following components: *FiniteStateMachineHsi* or *RandomDataGenerator*; *c*) *IScriptGenerator*: this interface is an optional variation point that can be realized by one of the following components: *VisualStudioScript*, *LoadRunnerScript*, *EmmaScript*, *JabutiScript* and *JmeterScript*; *d*) *IExecutor*: is an optional variation point that can be realized by one of the following components: *VisualStudioScript*, *LoadRunnerScript*, *EmmaScript*, *JabutiScript* and *JmeterScript*. For a detailed description of component functionalities, see [3].

In accordance with Figure 1, a valid configuration of a MBT tool derived from PLeTs could have the following components: *PLeTs*, *UmlParser*, *TestCaseGenerator*, *PerformanceTesting*, *FiniteStateMachineHsi*, *LoadRunnerScript*, *LoadRunnerParameters*. Based on the selected components, the generated tool supports the extraction of test information from UML models (*UmlParser* component), then generates test cases (*TestCaseGenerator* component) using a Finite State Machine and the HSI method (*FiniteStateMachineHsi* component) to apply performance testing (*PerformanceTesting* component). The generated test cases could be used as a input to generate scripts to the LoadRunner testing tool (*LoadRunnerScript* component) and then the scripts could be loaded in the tool and run the test (*LoadRunnerParameters* component).

III. PLeTs Architecture Limitations

During the design and development of an SPL, some of the main issues faced by an SPL Architect are caused by changes in the requirements, as these may result in the inclusion of unpredicted features to the SPL (reactive approach [7]), *i.e.* features that may not fit into the initial design of the SPLA [12]. In our experience, this has often meant that some of the core components of the SPL must be altered to make use of this new feature, which in some cases could imply in changes to the SPLA. This may in turn result in the propagation of these modifications to other unrelated software components in order for them to comply with the changes in the SPLA.

In the early versions of PLeTs [1] [3] [4] [10], we have attempted to solve the problem of SPLA volatility with two different approaches: *Component Interfaces* to map the access between components and, *compile-time definitions* to isolate statements that instantiate variability [2].

For Component Interfaces, we assumed that we would be able to create a coarse-grained relationship between a Feature Model and a Component Diagram, that is, a 1:1 mapping between features and components. Though we were able to create certain components that could be shared between different systems, our final result required high maintenance due to the volatility of our SPLA. Furthermore, any features that could not be directly translated into a single component inevitably became entangled in dependencies, sometimes limiting reuse in the SPL.

To address the difficulties imposed by a coarse-grained relationship, we allowed for the finer-grained representation offered by compile-time definitions. By doing this, we were able to create a more suitable mapping between software features and implemented code. We were also able to concentrate most of the instantiation of variability management within a (comparatively)

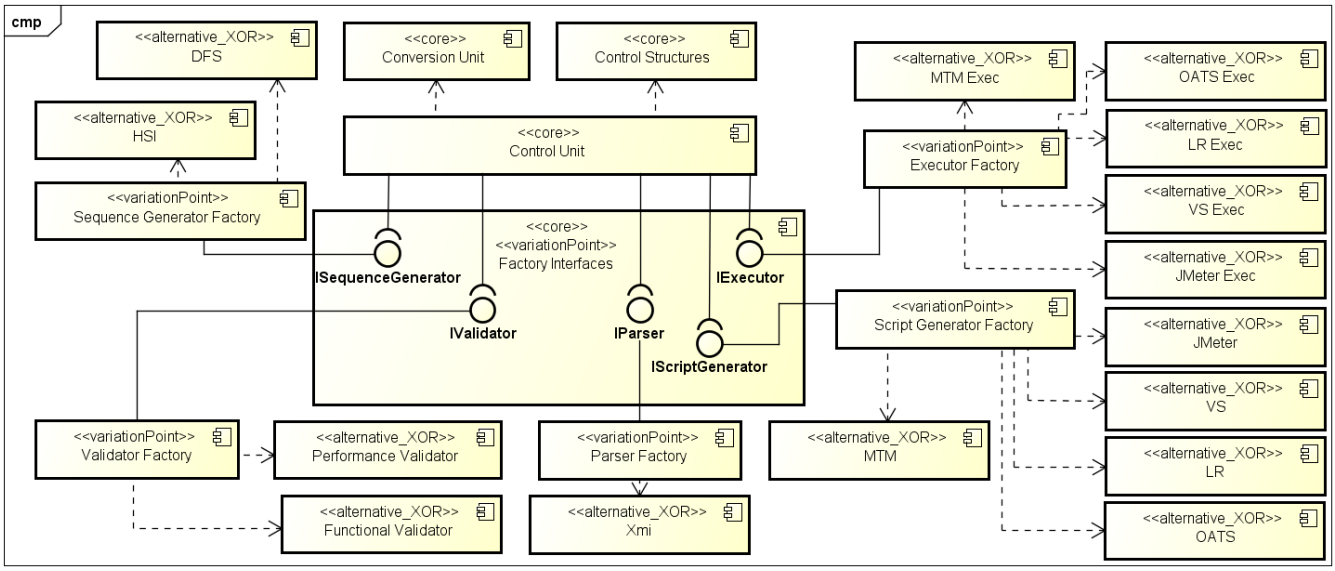

Fig. 2. PLeTs Improved - UML Component Diagram

small section of the code. Before the use of compile-time definitions, we had variability management spread across the entire source code, whereas now we are able to contain it within a well-defined section with few statement blocks.

Another difficulty we constantly faced was the maintenance caused by any changes in a data structure used by more than one component of the SPL. Such changes would often impose the modification of several components. Sometimes these modifications would be of such a degree that the effort required to update all related components was higher than that of creating a new data structure. We soon came to realize that this option was merely mitigating the update effort, as several adapters and converters had to be made over time to suitably enable the new data structure to work with certain product configurations.

Though each of the two approaches described had their merits, neither fully addressed the difficulties we found in evolving the SPLA in our research center environment. This environment is volatile due to the need to assign team members to different projects during certain parts of the development phase and the shifting of team members throughout the development cycle. For an SPL framework to fit into this environment, we found it necessary to establish certain guidelines (see Sections IV-1 and IV-5) and mechanisms (see Sections IV-2, IV-3 and IV-4) for the maintenance and extension of the SPLA.

IV. PLeTs Architecture Evolution

In order to tackle the issues raised by our previous development paradigms, we have adopted a mixed approach that is largely based on the use of the Factory Method design pattern [6] to externalize variability points from the implementation of concrete features [13]. We kept the component-based approach. The difference is that our SPL now has a well-defined core that centralizes the variability management, as well as serving as a starting point to the execution of any derived product.

The core of our SPL is composed by four components (see Components marked with the stereotype *core* in Figure 2), described below: *Control Unit*; *Factory Interfaces*; *Control Structures* and; *Conversion Unit*. Our goal has been to simplify variability management as much as possible by ensuring the independence between the SPL core and the feature instantiations, specifying sections (*i.e.* Factory Interfaces component) of the core to manage variability.

1) Control Unit: The Control Unit component is responsible for orchestrating the execution of the system, providing access to the functions of those components that implement features and organizing the data structures necessary for the proper execution of the system. It is designed and implemented without any dependencies on components external to the SPL core, which protects it from modifications in them.

2) Factory Interfaces: In order to access the components that are external to the core, that is, all features, the Control Unit makes use of the Factory Interfaces component, which is an abstract representation of the variability points of the SPLA. It contains interface definitions for each variability point, each serving as a connection point for a variable component. Each variability point in the SPL Architecture is represented here by one interface.

In Figure 2, we can see that within the Factory Interfaces component we have five interfaces represented. Each of these interfaces, for example *IExecutor* [3], contains a signature to all operations that must be available in a component that fulfills the role of the equivalent abstract feature[2], for example, the Executor feature. All references contained in the Control Unit with the intention of accessing a variable component will be made with one of these interfaces, so that the Control Unit may know what it is able to do without relying on access to the libraries of the variable components.

3) Control Structures: To access the data structures held by the components external to the core without establishing dependencies to the libraries that define these data structures,

[2]In our approach, abstract features represent the variability points of the system.

the Control Unit makes use of the Control Structures component. Like the Factory Interfaces component, Control Structures is an abstract representation of part of the SPLA, in this case, the data structures required by the SPL. While the Factory Interfaces component represents variability, the Control Structures component represents commonality, that is, any representations that are common to two or more data structures of the system. These are abstract data structures that serve as a surface representation of the concrete structures of the SPL. These abstract structures can either be a number of detailed ones that make use of inheritance or a single structure to serve as a placeholder for all others. To enable the identification of a specific instance of a Control Structure by the Control Unit without creating dependencies between it and the data structure components, we have specified that all Control Structures must have an identification attribute. Specifically, we have used an enumerator, created inside the Control Structures library itself, listing the data structures present in the system.

4) Conversion Unit: The Conversion Unit is responsible for parsing structures that are equivalent, i.e. any parsing process that does not change the content of a structure, such as the refactoring of a structure to execute different functions or the updating of a structure to a newer version. The Conversion Unit is a subsystem within the SPL, serving as a centralizer to the inclusion and updating of data structures. This part of the SPL core has been essential to the evolution of our project, greatly lowering the effort required to adapt product configurations to changes in data structures and vice-versa. By having all of the converters available to one another, we are able to create a directed graph of possible structural conversions and identify entry points to easily include new structures to the system. Rather than having to create adapters and converters for all combinations of data structures, the Conversion Unit makes use of the commonalities between them.

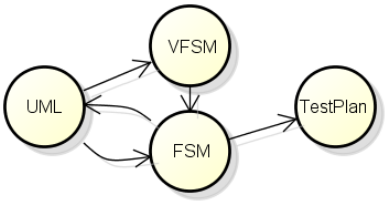

Fig. 3. Conversion Unit Diagram for PLeTs

Figure 3 shows the conversions currently available in the Conversion Unit. We have a single modeling format, based on UML diagrams, and a single test script format, called Test Plan. Additionally, we have the state transition models Finite State Machine (FSM) and (VFSM), from which the test sequences are generated. When the inclusion of VFSM was made, our requirement was that it be convertible to and from the UML and FSM formats, and to the model Test Plan, resulting in a total of five converters. As shown by the cycles in the graph, by implementing two of these converters ("UML to VFSM" and "VFSM to FSM"), we were able to fulfill all five of the requirements for the inclusion of this new structure. For example, the conversion "VFSM to Test Plan" is made by first converting the VFSM into an FSM, and then converting the resulting FSM into a Test Plan. We are aware that this composite conversion of structures may be detrimental to performance, but in our experience this has not been an issue. Should performance be critical to a certain conversion, a specific converter can be added.

For every abstract structure defined in the Control Structures component, the Conversion Unit has a factory capable of reading its type, as well as the return type desired, and forwarding it to the appropriate concrete structure converter. If a new structure component is developed for the system, specific converters will have to be implemented for that structure in order to convert both to and from it. All of these converters are contained within the Conversion Unit component itself, and are therefore accessible by the SPL Core.

Figure 4 presents an example of the process executed by the Conversion Unit. The Control Unit makes a request to the Conversion Unit. This request sends both the structure to be converted and the desired return type. The Conversion Unit identifies the type of the input structure and forwards the request to the appropriate Converter Factory, in this case the UML Factory. The Converter Factory infers the required converter based on the return type, and forwards the request to it. Finally, the converter casts the resulting structure into a general purpose structure described in the Control Structures component and returns it to the Control Unit.

The key factor that enables us to do this is the extensibility of a single library by partitioning it. The only library upon which the other core components are dependent on is the one containing the factories responsible for the first phase of the conversion, that is, the reading of input type. The concrete converters are each implemented in their own packages, compiled into their own libraries and kept outside of the SPL Core. They are implemented as extensions to the Conversion Unit namespace and included as needed by the compiler, resulting in a single logical unit that is distributed among various libraries with varying dependencies. The result is a final product where the converters, being plugged into the SPL Core as needed, are accessible from all concrete feature implementations while the SPL Core, being guarded from these concrete converters, can function in their absence.

5) Variable Components: We have found that the reuse of features for the creation of new product configurations becomes simpler and requires less refactoring effort if the variable components that implement these features are developed autonomously, that is, to be capable of execution as an atomic software unit if given an appropriate input. To make this possible without requiring the SPL core to depend on the libraries of all known variable components, an intermediate entry point is represented in the SPLA in the form of Factory components.

These factories make use of both compile-time and runtime logical operations to return an instance of the correct main feature realization. The compile-time operations are used to define what variable components are available to a given product configuration. The runtime logical operations are used when more than one option is available as a realization of a main feature, evaluating input from the user and the current state of the software to return the correct option. More about the Factory Method design pattern can be found in [6].

Ideally, each new feature added to our SPL as an alternative to represent a variability point in the SPLA will be developed as

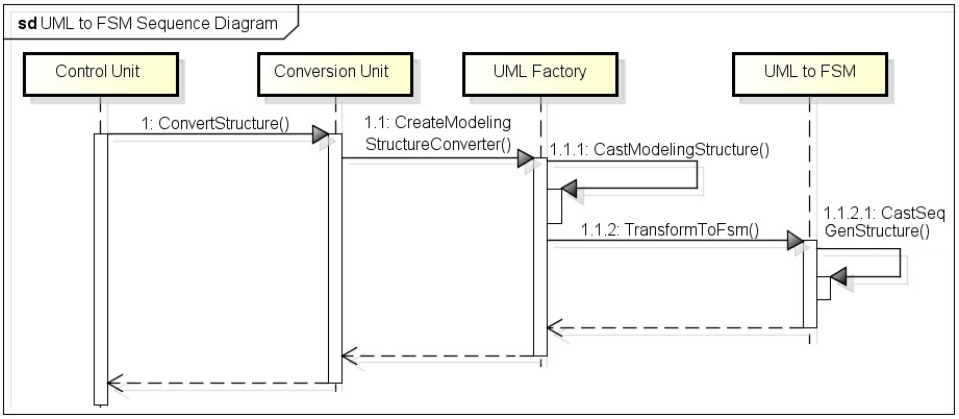

Fig. 4. PLeTs UML to FSM Sequence Diagram

its own component. This results in several diverse components (variants) to resolve each variability point. Alternatively, we have identified the option of representing a variant as an extensible component, in a manner similar to the Conversion Unit (See Subsection IV-4). Based on our experience, this can be useful during particularly hectic periods of development, during which hotfixes are required constantly and there is not enough time to build an entirely new component.

Our experience with the Sequence Generation feature of PLeTs ("Sequence Generator Factory" in Figure 2) serves as a practical example of the application of this option. This particular feature has proven to require adjustments each time a new product configuration was to be derived. Given the speed at which new versions were required and the sometimes mutually-exclusive nature of these adjustments (what would work with one configuration could not work with another, and vice-versa), we found the creation of an entire new component too costly. Instead, we developed small extensions to the existing component that would either include new operations or override existing ones. The original component, along with the extension component, would then be packaged as a single logical unit within a certain configuration, essentially becoming one possible implementation of a certain feature. We used this method purely for reasons of compatibility, and they were used only until such a time as we were able to develop a more robust and versatile component to represent that feature (less demand, more resources available).

In the case of developing new components for each feature, we have the advantage of finer granularity. Since we began using this approach, we have found that the maintenance of our SPL has become simpler than the alternative identified, as the independence of components is strictly enforced by their isolation. This means that the failure or replacement of a concrete feature has little impact to the rest of the system. On the other hand, the stricter isolation requires that the Factory component (responsible for managing the variability point) be kept up to date with modifications to the SPLA, requiring recompilation of the SPL Core. We are aware that this means either that a new Factory component must be written every time a new feature is added to this variability point or that the original Factory's source code must be available to anyone working on the SPL. In our experience, due to the entire SPL being worked on within a single environment and therefore all code being available during development, this has not been an issue.

In the case of extensible components we have the advantage of faster development time and greater inter-feature accessibility. Dependencies can be formed between different alternatives to a single variability point, allowing for an incremental extension of libraries without access or change to their source code. This approach also allows a developer to make alterations to a component without necessarily having access to the original factory (indeed, the factory related to this variability point should ideally remain unchanged). This means that independent developers are able to alter and extend the SPL without being given full access to its source code. This approach does not, however, offer any assurances in regards to system granularity. A component built with several codependent sub-components using this approach incurs on the liability to complete system failure if a single unit of the variable component should fail.

V. Related Work

Although Software Product Lines is an active research field, few works present new approaches for the implementation of an SPL, as well as discuss difficulties and limitations faced by SPL design and development teams.

For example, in [15], the authors speak extensively on techniques for Variability Management and present the case study of the Mercure PL, in which the Abstract Factory design pattern is used as a decision model, with each of its concrete factories being related to one product. Our approach has similarities with the one presented in this particular work, but diverges from it in that our use of the Factory Method design pattern is extended to deal with each of the variability points of the SPL.

In [8], this topic is also discussed. A two-dimensional model is proposed for the representation of the issues in variation management, with "files", "components" and "products" in one axis and "sequential time", "parallel time" and "domain space" in the other. The author argues that the nine smaller issues defined by this model can be tackled using a divide-and-conquer strategy.

VI. Lessons Learned

This section presents the lessons learned from the development and evolution of PLeTs and the subsequent evolution of its SPLA.

- One of the limitations of PLeTs' previous SPLA was the presence of strong dependencies between components. This often meant that changes in one component resulted in a chain maintenance through all components dependent upon it. To mitigate this problem we have proposed a change to the SPLA to add an SPL core that contains all the basic operations supported by the SPL. Rather than simply making a separation between the concepts of commonality and variability, we found it advantageous to add components into the core for managing the execution and communication of the variable components, *i.e.* the Control and Conversion units. In doing this, we were able to give autonomy to the variable components, avoiding the dependencies that may have arisen between them.
- In some situations, we had difficulty adding new components to the SPL due to the absence of a well-established entry point. In using the Factory Method to automate the process of instantiating components during execution, we were able to isolate the code referent to the majority of variability decisions into small sections that are easy to maintain. This has facilitated the expansion of the SPL by creating an entry point for the adding of new variable components.
- Both SPLAs used benefited from the component-based approach to the realization of variation points in the SPLA. Connecting a new component to the SPL through one of the pre-existing factory interfaces is a simple process, requiring only the packaging of input and output in accordance to the Control Structures component of the core. The new SPLA did not negatively impact in this.
- Should changes in requirements result in new variability points in the SPLA, the SPL core will require modifications. A new factory interface and variation point factory will be necessary, and this will incur in modifications to the Control Unit. We are aware of the implications of such modifications, but have not had the opportunity to test whether or not modifications in the Control Unit would in turn incur in changes to the variable components. Given the well-defined nature of the MBT technique, such radical changes to the SPLA are unlikely in our environment.

VII. Conclusion

In this paper we report our experience on the design, development and evolution of a Software Product Line of Model-based Testing tools - PLeTs. We have focused on techniques to simplify the process of managing the evolution of components by use of a software design pattern. We have also presented a description of the core components of PLeTs, which were designed to simplify the communication among variable components, as well as increase the degree of autonomy they have between one another.

Despite the completion of the development of our SPL in accordance to the current SPLA, we are certain that there are further issues to be resolved related to the growth of the SPL. This work details the maintenance and evolution given to address specific issues identified during the evolution of the SPLA, but over the course of this maintenance the SPL did not have any important features added to it. It is important to evaluate how well the SPLA design proposed here withstands the problems imposed by the addition of new variability points. It would also be valuable to investigate how well a new team member would adapt to this new approach, considering that we work in a volatile environment and new developers might not have prior knowledge of the design pattern used to develop PLeTs.

Acknowledgments

Study partially developed by the Research Group of the PDTI 001/2012, financed by Dell Computers of Brazil Ltd. with resources of Law 8.248/91. We thank Dell for the support in the development of this work.

References

[1] L. T. Costa, R. Czekster, F. M. Oliveira, E. M. Rodrigues, M. B. Silveira, and A. F. Zorzo. Generating Performance Test Scripts and Scenarios Based on Abstract Intermediate Models. In *24th International Conference on Software Eng. and Knowledge Eng.*, pages 112–117, 2012.

[2] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[3] E. de M. Rodrigues, L. Passos, F. Teixeira, A. F. Zorzo, and R. Saad. On the Requirements and Design Decisions of an In-House Component-based SPL Automated Environment. In *26th International Conference on Software Eng. and Knowledge Eng.*, pages 483–488, 2014.

[4] E. de M. Rodrigues, L. D. Viccari, A. F. Zorzo, and I. M. Gimenes. PLeTs-Test Automation using Software Product Lines and Model Based Testing. In *22nd International Conference on Software Eng. and Knowledge Eng.*, pages 483–488, 2010.

[5] C. Gacek and M. Anastasopoules. Implementing product line variabilities. In *Symposium on Software Reusability: Putting Software Reuse in Context*, pages 109–117, New York, NY, USA, 2001. ACM.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[7] C. W. Krueger. Easing the transition to software mass customization. In *4th International Workshop on Software Product-Family Eng.*, pages 282–293, 2002.

[8] C. W. Krueger. Variation management for software production lines. In *Software Product Lines*, pages 37–48. Springer, 2002.

[9] K. Pohl, G. Böckle, and F. J. v. d. Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer–Verlag New York, Inc., 2005.

[10] M. B. Silveira, E. M. Rodrigues, A. F. Zorzo, H. Vieira, and F. Oliveira. Model-Based Automatic Generation of Performance Test Scripts. In *23rd International Conference on Software Eng. and Knowledge Eng.*, pages 1–6, Miami, FL, USA, 2011. Knowledge Systems Institute Graduate School.

[11] I. Sommerville. *Software Engineering*. Pearson/Addison–Wesley, 2011.

[12] M. Staples and D. Hill. Experiences Adopting Software Product Line Development without a Product Line Architecture. In *Asia-Pacific Software Eng. Conference*, pages 176–183, 2004.

[13] T. Thum, C. Kastner, S. Erdweg, and N. Siegmund. Abstract features in feature modeling. In *15th International Software Product Line Conference*, pages 191–200, 2011.

[14] M. Utting and B. Legeard. *Practical model-based testing: a tools approach*. Morgan Kaufmann, 2006.

[15] T. Ziadi, J.-M. Jézéquel, F. Fondement, et al. Product line derivation with uml. In *Software Variability Management Workshop, Univ. of Groningen Departement of Mathematics and Computing Science*, 2003.