On the Requirements and Design Decisions of an In-House Component-Based SPL Automated Environment

Elder Rodrigues*, Leonardo Passos†, Leopoldo Teixeira‡, Flávio Oliveira*, Avelino Zorzo*, Rodrigo da Silva Saad§
*Pontifical Catholic University of Rio Grande do Sul
Porto Alegre, RS, Brazil
{elder.rodrigues, flavio.oliveira, avelino.zorzo}@pucrs.br

†University of Waterloo Waterloo
Canada
lpassos@gsd.uwaterloo.ca

‡Universidade Federal de Pernambuco
Recife, PE, Brazil
lmt@cin.ufpe.br

§Dell Computers of Brazil Ltd.
Porto Alegre, RS, Brazil, RS, Brazil
rodrigo_saad@dell.com

***Abstract*—Software product line adoption has many challenges in industrial settings. A particular challenge regards the use of *off-the-shelf* tools to support this process, since these tools usually do not fully address some company's specific needs. To elicit concrete requirements and provide tool vendors and implementers with direct feedback, we avail from our experience in developing a software product line to derive testing tools for a laboratory of a global IT company (currently set as a pilot study). In this paper, we present such requirements and argue that existing tools fail to address all of them. In addition, we present our design decisions in creating an in-house solution meeting the specific needs of the partner company. We also highlight that these decisions help in building a body of knowledge that can be reused in different settings sharing similar requirements.**

I. Introduction

Software Product Line (SPL) has emerged as a promising technique to achieve systematic reuse and, at the same time, decrease development costs and time to market [13]. Although the use of SPL practices has significantly increased in recent years, with an extensive list of successful cases,[1] there are still many challenges when implementing SPLs in industrial settings [9].

One particular concern relates to the use of *off-the-shelf* tools (*e.g.*, pure::variants [2] and Gears [8]) supporting the SPL development life-cycle, since frequently these tools do not meet specific requirements of the companies aiming to adopt them. To overcome this, many companies set to develop in-house solutions to support their specific needs.

The SPL community, however, currently lacks evidence on the driving factors around custom-based solutions, which hinders tool vendors and implementers from having feedback from industrial clients outside their clientele. To mitigate this, we report our experience in implementing a SPL to derive testing tools for a laboratory of a global IT company. In particular, we describe the specific requirements of that company and argue that existing tools fail to support them. We also present the design decisions in creating a customized solution supporting the target SPL. Our contribution are twofold: (i) we identify a set of requirements that, although specific to our research context, already point needs currently not addressed by existing tools, either commercial or open-source. Thus, we elicit practical scenarios that tool vendors and/or implementers may consider supporting; (ii) we report our design decisions in fulfilling these requirements for an in-house solution developed for our partner company. These decisions, in turn, may be reused or adapted to improve existing tools or when devising solutions targeting similar needs.

This paper is organized as follows. Section II presents some context relative to the company in place, from which Section III builds on. The latest one then enumerates the elicited requirements, which we address with specific design decisions, discussed in Section IV. Section V briefly presents our custom-made tool and its usage workflow. Section VI revisits related work, while Section VII concludes the paper and points out final remarks.

II. Context

Our research group on Software Testing[2] works in cooperation with the Technology Development Lab (hereafter referred as TDL) of a global IT company, whose development and testing teams are located in different regions worldwide.

In the partner IT company, development is performed over different programming languages and IDEs (Integrated Development Environments), including Visual Studio (for Microsoft-based solutions), Eclipse (for Java-based implementations), Flash Builder, PHP and packaged applications (*e.g.*, Siebel and Oracle).

Testing teams use commercial and open source frameworks and tools to partially automate their testing activities. Frequently, however, due to the complexity of the testing in place, these teams create custom components to enable testing non-trivial applications. For example, testers may need to create web-service-based scripts to test back-end components that provide

[1] http://www.sei.cmu.edu/productlines/casestudies/

[2] www.cepes.pucrs.br

interfaces to the front-end of a given application, or components that simulate asynchronous messages for offline business processes. Currently, due to the distributed nature of the testing teams, custom components are rarely shared, thus leading to redundancy, little reuse, coding inconsistencies, higher development costs, etc.

To eliminate the effort of repeatedly creating custom infrastructure, we started a pilot study on SPL adoption with the TDL of our partner company. Upon success, we aim to replicate or apply it in different testing teams, if not all of them. In particular, we used a SPL called PLeTs [4] that aims to support the derivation of a particular testing infrastructure from a set of shared components (product configuration), which are then glued together (product derivation). In this SPL, derived products are testing tools that take behavioural models as input; these models denote specific test cases, thus leveraging testing teams to follow a model-based approach [18], a process that we describe elsewhere [4].

Prior to creating any tool to support product configuration and derivation in the target SPL, the TDL first considered the use of off-the-shelf solutions, provided that specific requirements were met. We describe this requirements list in Section III.

III. Requirements

This section enumerates the requirements (RQs) we collected from the TDL concerning the configuration (component selection) and derivation of products (gluing selected components), in addition to the evolution of the target SPL.

RQ1) The adopted tool must not be bound to any IDE. Although the TDL relies mostly on Microsoft-based solutions, they also use solutions from different distributors and vendors (*e.g.*, Eclipse for Java). In that case, the adopted tool must not impose the use of any IDE, as that would require a specific platform expertise that could limit TDL's way of working, and also lead to training costs unrelated to product line adoption.

RQ2) The adopted tool must support a graphical-based notation for designing feature models. This requirement follows from the fact that features are an effective communication medium across different stakeholders and feature models (FMs) are a widespread mechanism to capture variability. Thus, the TDL requires the use of FMs, with the particular need of a graphical-based notation. According to the TDL, the use of a graphical-based FM notation is likely to facilitate communication with different stakeholders outside testing teams, including non-technical staff (e.g., project managers). Moreover, graphical FMs provide testers with a quick visualization of the existing features in the current snapshot of the testing infrastructure, and how each feature relates to others. The TDL explicitly states that relying on textual-based notations is not an option. They argue that textual languages impose linearity, as one element is only known when another one ends, hindering an immediate grasp of the underlying structure.

RQ3) The adopted tool must support a graphical-based notation for designing structural architectural models. Since testing teams can be geographically distributed, changes to the underlying test infrastructure must be documented at all times to facilitate communication and future maintenance. In this case, in addition to keeping FMs, assets of shared components in the test infrastructure should also be documented in terms of structural architectural models (currently set to be component diagrams) that closely resemble their coding artifacts. These models capture fine-grained details that FMs alone would otherwise miss (e.g., a component port). The preference towards graphical notations is in tune with RQ2.

RQ4) Structural architectural models must be kept in synchronization with the FM and code base (and vice-versa). This requirement imposes a full round-trip between FMs and architectural models. As before, models should be presented and edited graphically.

RQ5) The tool must be extensible to support different structural architectural models and FM notations. The TDL states that structural architectural modeling should be centered around UML diagrams (see RQ3), but they point out the benefit of supporting other notations in the future (e.g., Domain Specific Languages). Likewise, one should also account for different FM notations, with extensions added as needed.

RQ6) FMs should be derivable from structural architectural models. In the TDL, most testers are familiar with standard UML models, but less so with FMs. To prevent initial mistakes and minimize the effort in extending the testing infrastructure, the tool must be able to derive a FM from the defined structural architecture, which in turn can be tuned accordingly.

RQ7) Structural architectural models should be derivable from FMs. As time progresses and FMs become more common among stakeholders, testers can start extending the testing infrastructure by first changing the FM, and then deriving the corresponding structural architectural model, which can be tuned accordingly (round-trip is already requested by RQ4).

RQ8) For each product of the testing infrastructure, it should be possible to derive its corresponding structural architectural model. Testing tools (products) are the result of selecting and combining components. For each product, it should be possible to generate its structural architectural model, which results from selecting specific elements from the architectural model of the whole SPL.

RQ9) Traceability links among models and implementation assets/elements should require minimal human intervention/effort. Traceability is an important concern for the TDL, as FMs and structural architectural models need to be mapped to implementation assets, and vice-versa. To prevent a high burden on manually keeping such links, any adopted solution must automate traceability as fully as possible.

RQ10) When extending the existing infrastructure with new features (components), their implementation must adhere to specific interfaces. To decrease coding effort and avoid human mistakes while enforcing coding styles, an initial skeleton implementation should be automatically derived from specific code templates. Currently, when testers evolve the testing infrastructure (e.g., when implementing the interface of a core capability of the testing infrastructure) they often copy and adapt an existing implementation or write a new one from scratch.

RQ11) The adopted tool must allow the creation of new glue code generators, that should be pluggable into the system without intrusiveness changes. The adopted tool must allow hooking code generators to produce glue code for specific target languages. This is aligned with the need to integrate with non-Microsoft solutions.

Summary: by evaluating existing tools for SPL adoption, we found that no existing off-the-shelf tool (either commercial or open-source) meets all of the presented requirements. Therefore, we and the TDL set to create an in-house component-based tool to support the target SPL. In Section IV, we describe our design decisions in building such a tool.

IV. Design Decisions

In this section, we report our design decisions (DDs) in creating an in-house tool supporting the requirements previously discussed. For each design decision, we refer to the associated requirements.

DD1) Build a plugin-based tool that is extensible (RQ5, RQ11). To allow extensibility, we employ a modular solution with a plugin-based architecture. The tool is currently implemented in C#, as the partner company relies mostly on Microsoft-based technologies. The use of C# facilitates integration and long term maintenance, as different employers can potentially enhance the tool in the future, either by developing new plugins, or improving its core.

DD2) No support for code editing (RQ1). This makes the tool IDE-independent, and as such, individual developers or entire testing teams can continue to use their preferred coding editors and/or development environments.

DD3) Provide an extensible environment for using different feature models notations, with the use of FODA-based notation [7] as the default one (RQ2, RQ5). Currently, the tool ships with a feature model editor plugin, that in addition to FODA's syntax, also supports definition of abstract features [17].

DD4) Support an extensible environment for using different structural models. In particular, we currently support UML component diagrams as the default structural architectural model (RQ3, RQ5). In the partner company, component diagrams are the most common artifact for documenting the structure of any given software architecture (at least in the TDL). Due to its widespread use within the company and from the fact that UML is taught in universities in the country where the TDL is located, its use dispenses extensive training, and thus reduces both costs and time. It is worth noting that the component diagram plugin targets a particular UML profile– SMarty [12], due to its support for variability encoding in UML diagrams.

DD5) Every concrete feature is mapped to exactly one component. In case of feature interactions, we rely on #ifdef-based annotations (RQ4, RQ9). Such a simple design dispenses the maintenance of explicit traceability links, thus eliminating the associated burden. In our pilot study, we observed that most features (components) are coarse-grained, making an 1:1 mapping a suitable solution. Since, #ifdef annotations are rare, the TDL is unlikely to face situations where maintenance is hindered by #ifdef complexity (known as #ifdef *hell* [15]).

Binding between components and features is performed by exact name matching, and the same occurs for #ifdef macros (one component matches exactly one macro name that cannot be redefined).

DD6) We synchronize the feature and structural models to maintain consistency after performing changes in each of them (RQ4, RQ6, RQ7, RQ9). Consistency, in this case, is eased by relying on a 1:1 mapping between feature and components, which makes transformation between models a straightforward task.

DD7) Derivation of a FM from a component diagram (and vice-versa) follows from a direct mapping between FM elements and UML elements in the SMarty profile (RQ8). The Smarty UML profile maps to FODA's syntax by a 1:1 correspondence. When deriving a FM from the corresponding component diagram (the inverse derivation), no abstract features will ever be created, as they do not have a corresponding element in a component diagram. Abstract features (see Section V) can still be preserved in the derived FM if one translates an existing FM into a component diagram, changes the latter, and derives the FM back. If the original FM contained abstract features, they will continue to exist, as long as they still root an element that maps to an element in the component diagram. This respect the full round-trip given by DD6.

DD8) After the generation of each project, one can generate a corresponding component diagram matching its architecture. By following the mapping between features and components, and with the component selection of a derived product, we take the product line component diagram (if non-existent, we derive it from the FM – see DD7), and resolve its variability according to the feature/component selection. That results in a component diagram documenting the architecture of the corresponding product.

DD9) Every component corresponds to a single compilation unit. Again, the binding is performed by matching names. Moreover, based on the product configuration and on the 1:1 mapping between features and components, the tool creates, a single Visual Studio project for each selected component, allowing developers to have all the information required to build, maintain, test and evolve selected components (RQ10). As most of the partner company relies on Microsoft-based technologies, our solution include a single plugin supporting Visual Studio-based projects. Other plugins can be developed for other IDEs, including Eclipse, Netbeans, among others.

DD10) We define an extensible environment for using different target languages (RQ11). Through a modular architecture, our tool support plugins that can work with different programming languages. As this is a pilot study, initial support has been developed for C# only.

V. The In-house Tool

This section presents the in-house tool (PlugSPL) developed to meet the requirements set by the TDL. PlugSPL is a plugin-based tool written in C#, from which users design an SPL and develop its components. From that, valid combinations of

components are selected and their matching products generated (testing tools). Figure 1 shows the tool's workflow, comprised of four main activities, namely, *SPL Design*, *Component Management*, *Product Configuration* and *Product Generation*. In this process, different stakeholders are involved and may overlap roles. Existing roles include testers, managers, developers, etc. For simplicity, we refer to them either as *domain engineers* (if related to the first two activities in PlugSPL), or as *application engineers* (if related to the last two activities).

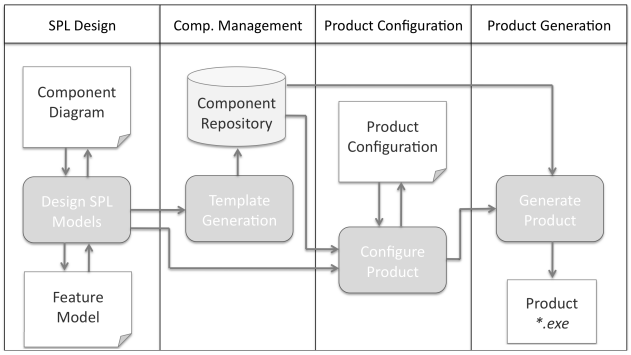

Fig. 1. PlugSPL activities

A. SPL Design Activity

The SPL Design Activity is the starting point in PlugSPL and aims to support the design of the SPL by means of a graphical FM notation or an UML component diagram.

Although PlugSPL supports FMs, the tool ultimately operates on the level of components, and requires the understanding of concepts such as components, interfaces and realization, as these drive later activities. For domain engineers with a strong background on FMs, but less so in component diagrams, the tool alleviates the modeling activity by supporting the automatic generation of a component diagram from the designed FM. In such case, engineers still manipulate components in other activities, but do not perform any modeling activity in terms of UML component diagrams. Similarly, for those with a strong background in UML component diagrams, but less so in FMs, the tool also supports the automatic generation of a FM from an existing component diagram. In both cases, edits in generated models are automatically synchronized with the models from which they are created, and vice-versa, along with their constraints (full round-tripping). By supporting both FMs and component diagrams and automatic conversion between them, PlugSPL allows an effective communication among different stakeholders in the TDL, with different modeling expertise.

To design an SPL using FMs, domain engineers rely on the FM graphical editor plugin (see Fig. 2). Besides supporting FODA elements (except or-groups), the editor allows marking features as abstract [17] (features are set to be concrete by default). Abstract features exist only to improve the organization of the FM, and are not mapped to any implementation element (class, interface, macro, etc.). Concrete features, on the other hand, follow a 1:1 mapping to a corresponding implementation component, and ultimately to a whole compilation unit. This mapping allows PlugSPL to trace a feature throughout its lifecycle.

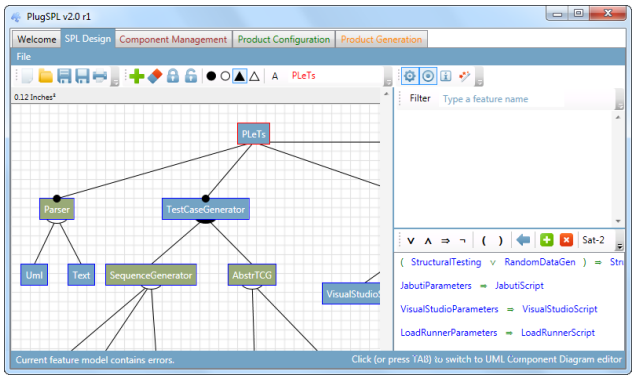

Fig. 2. PlugSPL feature model editor

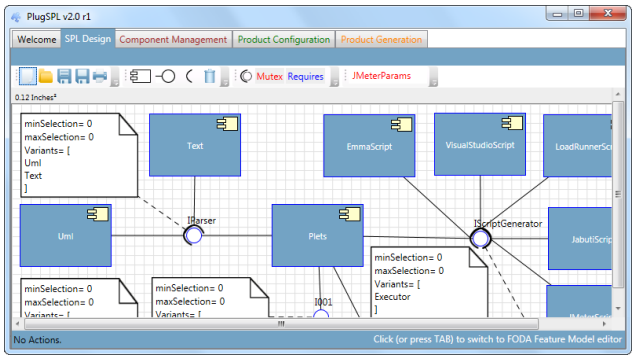

Fig. 3. PlugSPL component diagram editor

For the cases where domain engineers choose to model the SPL using component diagrams, they first select that diagram type (the UML component diagram editor is shown in Fig. 3). In this modeling approach, features are represented as components, which connect to other components by realizing their required interfaces. Since more than one component can realize a given interface, a required interface defines a variation point, and connecting components denote specific variants. These variation points can be further detailed by means of *tags* (UML comments, shown in Fig. 3) and *stereotypes* (*e.g.*, <<Mutex>> group, and <<Requires>> dependencies), allowing a fine-grain control over the variability in place. Tags allow engineers to control the cardinality of instances of each connecting component (captured as *minSelection* and *maxSelection*) and specify the set of possible variants. The value of *minSelection*/*maxSelection* is either zero or one, with the exception that *minSelection* and *maxSelection* are never both zero, and that *minSelection* is always less than or equal to *maxSelection*. Hence, this captures mandatory (*minSelection* = *maxSelection* = 1) and optional features (*minSelection* = 0, *maxSelection* = 1), but prevents the existence of or-groups. The absence of or-groups is currently a limitation, as PlugSPL cannot resolve which variant instance to use when integrating it with a given component. PlugSPL relies on the SMarty variability UML profile [12] as an annotation scheme.

Following a plugin-based architecture, the design activity in PlugSPL can be extended with other plugins supporting different FM modeling notations (*e.g.*, cardinality-based FM [5]) or UML diagrams (*e.g.*, class diagram). It can also be

extended to support different file formats (*e.g.*, SPLOT [11]).

B. Component Management Activity

In PlugSPL, the component management activity assists domain engineers in the implementation of the SPL components. Given the set of previously defined interfaces, domain engineers define their method signatures (operations) by importing external files (see Fig. 4). Not favoring any specific editor, even a built-in one, allows testing teams to continue using their preferred IDE or editor. Once the interfaces are defined, given the set of declared components, their interfaces, and their connections, PlugSPL generates an initial set of classes that conforms to them; still, these classes are not runnable, but rather skeletons whose associated methods are empty.

In the current C# plugin supporting this activity, each component results in a Visual Studio project, and each interface/class matches exactly one component. These projects are then distributed among different developers and/or testing teams, which then complete their implementation. In this process, developers instantiate interfaces through *fake* statements that are later replaced during *Product Generation*. This is due to the fact that developers cannot (and should not) predict which component can provide the contract of any given interface. Figure 6 illustrates this: instantiation of an `IParser` is done by instantiating the `DummyIParser` interface, a statement that is semantically incorrect, as interfaces cannot be directly instantiated (they only state a contract, and thus lack any behaviour on their own). This resembles the dependency injection pattern [14], while avoiding the burden of keeping XML configuration files, as required by many existing frameworks (*e.g.*, Spring [19]). The penalty, in this, case, is that variability is resolved at an early stage (during *Product Generation*), and not during runtime. Once the implementation of components is completed (they are now in the form of complete Visual Studio projects), they are fed back to PlugSPL, which in turn saves them in the component repository, provided no integration problem occurs.

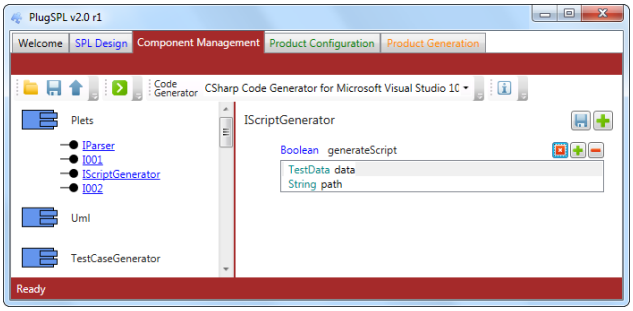

Fig. 4. PlugSPL Component management

C. Product Configuration Activity

In this activity, application engineers select the components that should comprise a target product (see Fig. 5). To allow such configuration, PlugSPL relies on the feature or component models previously designed, along with the components stored in the project workspace.

PlugSPL generates a tree view of the project's components and their interfaces, along with the set of components that

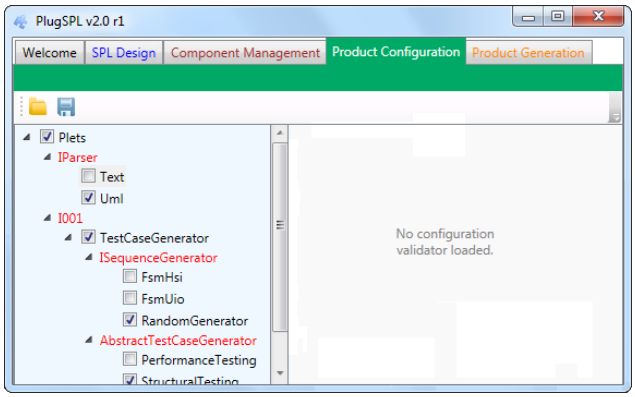

Fig. 5. PlugSPL Product configuration

can connect to each such interface. For instance, following the UML component diagram in Fig. 3, two components implement `IParser`, and serve as its variants: `UML` and `Text`. In that case, `UML` and `Text` appear as child nodes of `IParser` in the tree view in Fig. 5.

During configuration, application engineers select at most one component for each provided interface. In accordance with the constraints defined during the SPL design, PlugSPL automatically manages dependencies for selected components. The only exception occurs when configuration conflicts arise, which are then reported and must be manually fixed. Once a product is configured, the configuration is saved in the project workspace, and application engineers proceed to generate target products.

D. Product Generation Activity

In the product generation activity, from an existing product configuration and its chosen components, PlugSPL selects the corresponding Visual Studio projects generated during *Component Management*. PlugSPL then copies the source code of the selected components from the project workspace to a specified output folder, where components are then glued together. Gluing is performed by replacing extension points that instantiate interfaces (*fake* statements as previously discussed) by the instantiation of the concrete components in the configuration that support such interfaces. Figure 7 illustrates this: on line 5, the instantiation of `DummyIParser` (previously shown in Fig. 6) is replaced by the instantiation of the `UML` concrete class. Gluing also sets dependencies among different Visual Studio projects, *i.e.*, among related components. From the compilation of all components results a final product (executable testing infrastructure). As in other activities, the plugin supporting this activity is specific to C#-based projects.

VI. Related Work

The SPL community lacks studies that explicit state the requirements surrounding tool adoption and the corresponding design decisions in the case of custom-made solutions. The few studies attempting to tackle the first part (requirements) are based on collected interviews and surveys [1], [3], and aim to undercover particular challenges that could be the starting point for better tools and methodologies. Our study, although restricted to a single company and its specific requirements,

```
public class PLeTs{
    string fileName, newFileName;
    //ommited code
    Console.WriteLine("Initializing <Parser> component...");
    IParser parser = new DummyIParser();
    parser.LoadDocument(fileName);
    parser.ConvertStructures();
    parser.SaveDocument(newFileName);
    //ommited code
}
```

Fig. 6. Code before the replacement

```
public class PLeTs{
    string fileName, newFileName;
    //ommited code
    Console.WriteLine("Initializing <Parser> component...");
    IParser parser = new Uml();
    parser.LoadDocument(fileName);
    parser.ConvertStructures();
    parser.SaveDocument(newFileName);
    //ommited code
}
```

Fig. 7. Code after the replacement

provides an in-depth discussion over its requirements and context at place. Such requirements have not been fully exploited in the SPL literature, nor have they been fully addressed by existing tools (most notably, full traceability and round-trip over different models). Some teams report some of their design decisions when creating SPL-related tools, *e.g.*, Feature IDE [16]. However, decisions are not explicitly backed up by any industry-set requirements, but rather, from the creators own experience [16].

Other researchers investigate differences among existing tools [6], [10], but do not collect feedback based on industrial cases where these tools are used, or to which extent they succeed or fail when supporting SPL adoption.

On the tool development side, different solutions have been proposed, including both commercial and open-source. The two most popular commercial products today are pure::variants [2], from pure::systems, and Gears [8], from Big Lever Software Inc. Although they represent the most complete toolset for product line adoption, the specificity of the TDL's requirements make them unsuitable. The open-source arena is no different, although a plethora of solutions exist, ranging from web-based solutions [11], to Eclipse plugins [16]. A comprehensive list of existing tools, either commercial or open-source, is presented in [10].

VII. Conclusion and Future Work

This paper presented a set of requirements elicited in the context of an industrial partner and its Technology Development Lab. Through a pilot study, we collected specific needs targeting tool adoption for implementing a SPL-based solution for test products, and argue that existing tools, either commercial or open-source, do not meet the specificity of the requirements at hand. We then presented our design decisions when creating an in-house tool to fulfill our partner needs, along with a brief discussion of its supported workflow. We claim the reported requirements and design decisions as the two contributions of this work, as currently, few studies bring such discussion. Our work adds to that in the sense that the elicited requirements show practical scenarios that tool vendors and/or implementers may consider supporting; the design decisions, in turn, may be reused or adapted to improve existing tools or devise new ones targeting similar requirements.

As future work, we aim to keep track of our partner company needs and elicit new requirements as the SPL goes beyond the current pilot study, ideally being adopted by all testing teams.

VIII. Acknowledgements

Elder Rodrigues is a researcher at the Center of Competence in Performance Testing, a partnership between Dell and PUCRS.

References

[1] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, and A. Wkasowski. A survey of variability modeling in industrial practice. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, 2013.

[2] D. Beuche. Modeling and building software product lines with pure::variants. In *Proceedings of the 16th International Software Product Line Conference - Volume 2*, 2012.

[3] L. Chen and M. Babar. Variability management in software product lines: An investigation of contemporary industrial challenges. In *Software Product Lines: Going Beyond*, Lecture Notes in Computer Science. Springer, 2010.

[4] L. Costa, E. Rodrigues, R. Czekster, F. Oliveira, M. Silveira, and A. Zorzo. Generating performance test scripts and scenarios based on abstract intermediate models. In *Proceedings of the 24th International Conference on Software Engineering and Knowledge Engineering*, 2012.

[5] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 2005.

[6] M. Dammagh and O. Troyer. Feature modeling tools: Evaluation and lessons learned. In *Advances in Conceptual Modeling. Recent Developments and New Directions*, Lecture Notes in Computer Science. Springer, 2011.

[7] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature–oriented domain analysis (FODA) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, 1990.

[8] C. Krueger and P. Clements. Systems and software product line engineering with BigLever software Gears. In *Proceedings of the 16th International Software Product Line Conference*, 2012.

[9] C. W. Krueger. New methods in software product line practice. *Communications of the ACM*, 2006.

[10] L. B. Lisboa, V. C. Garcia, D. Lucrédio, E. S. de Almeida, S. R. de Lemos Meira, and R. P. de Mattos Fortes. A systematic review of domain analysis tools. *Information and Software Technology*, 52(1):1 – 13, 2010.

[11] M. Mendonca, M. Branco, and D. Cowan. S.P.L.O.T.: software product lines online tools. In *OOPSLA Companion*, 2009.

[12] E. Oliveira, I. M. Gimenes, and J. Maldonado. Systematic management of variability in UML-based software product lines. *Journal of Universal Computer Science*, 2010.

[13] K. Pohl, G. Böckle, and F. J. v. d. Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.

[14] D. R. Prasanna. *Dependency Injection*. Manning Publications Co., 1st edition, 2009.

[15] H. Spencer and G. Collyer. #ifdef Considered harmful, or portability experience with C news. In *USENIX*, 1992.

[16] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich. FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming*, 2012.

[17] T. Thum, C. Kastner, S. Erdweg, and N. Siegmund. Abstract features in feature modeling. In *Proceedings of the 15th International Software Product Line Conference*, 2011.

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

[19] C. Walls and R. Breidenbach. *Spring in action*. Manning Publications Co., 2007.