

XToF – A Tool for Tag-based Product Line Implementation

Christophe Gauthier, Andreas Classen,*
Quentin Boucher, Patrick Heymans

PRECISE Research Centre
Namur, Belgium

Email: chistophe.gauthier@student.fundp.ac.be
{acs,qbo,phe}@info.fundp.ac.be

Margaret-Anne Storey

University of Victoria
Victoria, Canada

Email: mstorey@uvic.ca

Marcílio Mendonça

University of Waterloo
Waterloo, Canada

Email: marcilio@csg.uwaterloo.ca

Abstract—This tool demo paper describes a tool called XToF which is being developed through a collaboration between the University of Victoria, the University of Namur and the University of Waterloo. The purpose of the tool is to let programmers define, maintain, visualise and exploit precise traceability links between a feature diagram and the code base of a software product line. The resulting tool supports automated configuration of a Java or C code base and is minimally intrusive with respect to development practices.

Index Terms—tool demo; software product line; feature diagram; tagging; programming language; C; Java

I. INTRODUCTION

The tool described in this paper is being developed as part of the Masters thesis of the first author. The tool extends a toolchain that was previously assembled/developed by the University of Namur and Spacebel, a Belgian company that develops software for space missions.

The purpose of both tools (the old and the new one) is to let programmers define, maintain, visualise and exploit precise traceability links between a feature diagram (FD) and the code base of a software product line. Both tools are meant to be minimally intrusive with respect to development practices. The new tool, called XToF¹, provides enhanced functionality by leveraging on two new components: (1) TagSEA, an Eclipse plug-in developed at University of Victoria, which purpose is to support navigation and knowledge sharing in collaborative program development, and (2) S.P.L.A.R. a Java library developed at University of Waterloo that automates various FD analyses.

The remainder of the paper is structured as follows. It starts in Section II with a description of the requirements (Section II-A) and implementation (Sections II-B and II-C) of the initial toolchain, together with a list of its limitations (Section II-D). Then, in Section III, we present the contribution of this paper: XToF, the new prototype designed to overcome the aforementioned limitations. We describe in turn its components and principles (Section III-A), its functionalities (Section III-B) and on-going as well as future development (Section III-C).

*FNRS Research Fellow.

¹XToF stands for cross(X)-Tagging of Features

II. THE INITIAL TOOLCHAIN

A. Context and requirements

The assembly/development of the initial toolchain took place as a collaboration between the University of Namur and Spacebel. The goal of this collaboration was to turn the implementation of a flight grade satellite communication software library into a software product line that would support the following requirements:

- allow *mass-customisation* of the library: meaning to be able to efficiently derive products that only contain the features required for a specific space mission,
- *be compliant with quality standards and regulations* in place for flight software,
- have a *minimal impact on current development practices*,
- *automate* the solution as much as possible.

The first and second requirements stem from the strict constraints that are imposed on flight grade on-board software. Components for space usage are developed to deal with extreme environmental conditions such as cosmic rays, temperature variation and vibration. This type of hardware is usually very expensive and several evolutionary steps behind the consumer hardware we more commonly know. Therefore, CPU usage and memory footprint typically have to be minimised. Also, developers often have no other choice than programming in C and obey strict rules that prohibit usage of ‘dangerous’ mechanisms, such as dynamic heap memory allocation, or general-purpose third-party libraries [1]. Along the same lines, *dead code* is also to be avoided. In our case, since specific missions only require part of the protocol’s functionality, it is important to only deploy those parts (or features) of the protocol that are going to be used.

The rationale for the third and fourth requirements was to facilitate adoption of the solution by the company. A first version of the toolchain was then elaborated jointly by the academic and industry partners. It is described in detail elsewhere [2]. In the rest of this section, we just recall its most important features and limitations to introduce our new contribution, XToF, presented in Section III.

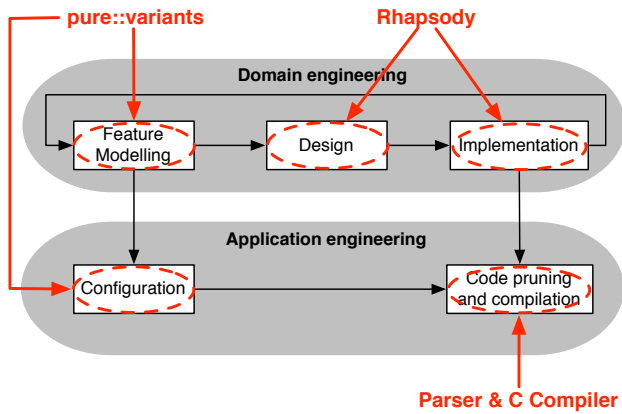


Fig. 1. Toolchain as deployed at Spacebel

B. Deployed toolchain

Guided by the above requirements, a first version of the process and toolchain were developed and successfully deployed in the company. The tool-supported process is depicted in Figure 1. It is organised after the classical software product line engineering process [3] which consists of two main streams: *domain engineering* (the creation of reusable artefacts or core assets) and *application engineering* (the usage and adaption of the core assets to create final products). In our case, the core assets are the Feature Diagram (FD), the system architecture (in UML) and the C code base, made of 6224 lines of code, the two last assets being decorated with tags pointing to the features of the FD. Application engineering starts with a configuration step during which some features are selected and some are discarded. This information is then used to remove code related to discarded features before the mission-specific product is compiled. A technical report [2] describes each of these steps in greater detail. It also gives an extensive definition of the syntax and semantics of our tagging language, demonstrates its correctness, provides an illustration and compares it with other annotative approaches to software product line implementation such as CIDE [4] and `#ifdef` pre-processing statements.

Figure 1 also shows how this process is supported by tools:

- design and implementation are supported by the tools already in use at the company, namely the *Rhapsody* UML CASE tool, and the C compiler;
- feature modelling and configuration are supported by *pure::variants*, a commercial off-the-shelf FD-based tool;
- a parser that was developed specifically for this project. The parser takes two inputs: (1) a valid list of features provided by *pure::variants* after configuration, and (2) an ANSI C source file annotated with tags written in our tagging language. It returns an ANSI C source file with no tags and no unnecessary code. The parser is encapsulated in a make-file and run on every single file of the codebase.

C. The tagging languages

Basically, a feature tag is an annotation of a block of C code with the names of the features that require the block to be present. If none of the features listed in a tag is included in a particular product, then the tagged code block will not be part of the source code generated for this product. Tags can be nested and a whole file can be tagged with a special annotation. Untagged code is assumed to be needed for all features.

Syntactically, a feature tag is a comment that follows a predefined pattern. As such, it is displayed in the same colour as comments in code editors. The syntax of feature tags is:

```

<fcomment> ::= "/*@feature:" <flist> "@*/" [<filetag>]
<flist> ::= <featurename> ( ":" <flist> ) *
<filetag> ::= "/*@!file_feature!@*/"
  
```

where `<featurename>` identifies a feature of the FD. The scope of a tag is the functional block, which we define as a group of statements that belong together, and that can be removed as a whole without violating the syntax or grammar of the language. For instance, it would be impossible to remove only the signature of a function without also removing its body. Functional blocks thus correspond to elements of the abstract syntax tree (AST), an idea previously found in [4]. With this approach, we can guarantee that the pruned code will always be syntactically correct. The functional block corresponding to a code tag is determined by the instructions that follow the tag. More details can be found in the technical report [2].

D. Limitations of the toolchain

The tool-supported process described in the previous sections turned out to be effective in meeting the requirements set out by the company. A detailed evaluation [2] revealed that there was still space for improvements (in order of priority):

- *Tighter integration*: communication between the tools was performed only through file exchange. Although this did not impede usage of the toolchain, it was recognised that an integrated environment, where loosely coupled tools play together, could be a significant enhancement. An important improvement, for example, could be that the feature editor/configurator could point directly to the code fragments a feature corresponds to in the code editor, and vice versa.
- *Legibility*: according to the company's developers, the legibility of the source code was not reduced by the tags. Indeed, the tagging language was designed to be concise and is rendered in a different colour in most code editors. However, the developers found it sometimes hard to determine the feature(s) corresponding to a specific source fragment, especially in the presence of nested tags. Tag-based filtering and visualisation techniques could alleviate this problem.
- *Portability*: although pruning dead code is most usually required in embedded systems where C dominates, C is not the only language used in embedded systems. Additionally, our "tag and prune" approach has a wider

applicability than embedded systems, hence the idea of extending the approach to other languages.

- *On-the-fly tag generation*: the programmers who used the toolchain estimated that the overhead due to the tags during the domain implementation phase was 20 to 25% with respect to tag-free implementation of a ‘maximal’ product (the return on this investment being delayed to the application implementation phase). However, they also recognised that the overhead could be decreased if the tags were systematically captured at the time the feature is programmed rather than after the fact.

III. THE NEW PROTOTYPE: XTOF

Functionally, XToF, the new prototype, is meant to support the activities depicted in Figure 1 in a single integrated environment, and overcome the limitations described in the previous section.

A. Components and principles of XToF

The opportunity for re-implementing the original toolchain came from the discovery of an open-source Eclipse plug-in called TagSEA. TagSEA was developed by Storey *et al.* [5] to support asynchronous and collaborative program development. It enhances navigation and knowledge distribution in the code based on tags placed by the programmers. The approach and tool are originally unrelated to software product lines, but turned out to be applicable in this context.

XToF uses the capabilities of TagSEA to manage tagging and tags. TagSEA defines *waypoints* as “locations of software model elements”[6]. The notion of waypoint as a *point of interest* has been extended to a *design area* of interest in order to capture blocks of code associated to feature tags. TagSEA provides mechanisms to filter tags, list waypoints and navigate to a waypoint. XToF then links TagSEA waypoints to features and blocks of code.

One of the main enhancements to the first toolchain is that the FD is now displayed directly in the programming environment. The FD is used as an index to code fragments and as the configuration interface. One can select a set of features to obtain specific views of the program and to configure it. XToF adopts the classical layout of Eclipse (see Figure 2): the FD is displayed as a directory tree (A), some buttons (B) trigger actions like configuration or tag filtering, while TagSEA constantly displays the list of waypoints (D) for each tag (C).

To allow tighter integration of TagSEA with the FD-related functionalities, we needed full access to their source code. This was provided by SPLAR² a powerful Java library that automates various FD analyses, by which we replaced *pure::variants*.

B. Current functionalities

We now take a closer look at the tool’s currently supported functionalities:

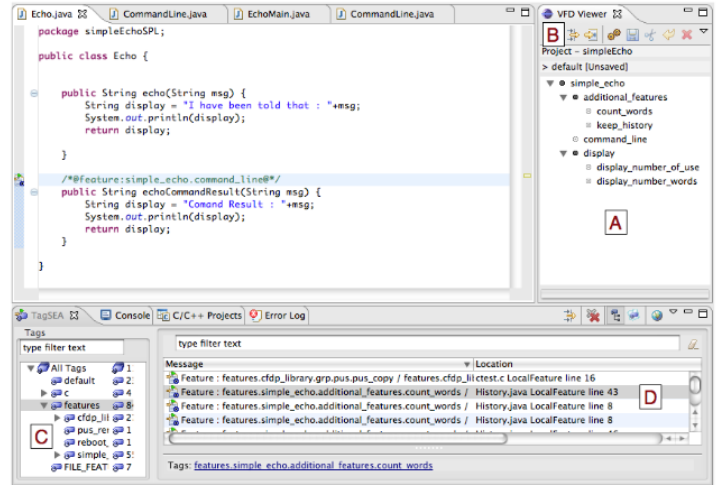


Fig. 2. XToF’s main screen

- *Loading the FD*: To be displayed and configured in the tool, the FD has to be loaded. XToF expects it as an XML file in the *SXFM* format.³ The file can be created in any text editor, but can be more easily produced by the web-based visual FD editor SPLLOT [7], the front-end to SPLAR. Once the FD is loaded, XToF displays it and lets the users add tags, navigate and configure. The loaded FD is copied to the project folder and its path is saved as a property of the project. The FD is thus made available to all project contributors who can work in parallel.
- *Tagging code fragments*: To reduce the time needed to tag blocks of source code, XToF uses auto-completion from Eclipse. While typing a tag, feature names are displayed, and when selected, directly added to the tag.
- *Navigation and visualization*: XToF feature tags behave like regular TagSEA waypoints. The user can list the location of feature tags, navigate to a tagged code fragment and display it. Some visualisations have been developed to answer simple questions such as “Which blocks are associated to a set of tags?” and “Which set of tags is associated to a line of source code?”. To answer the first question, the user can select the set of tags in XToF and the tagged block of source code is highlighted. Another mechanism provides the opposite function, i.e. answers the second question: the features corresponding to the current line in the active editor window are highlighted in the FD. Additionally, XToF filters packages and classes that contain blocks tagged with selected features from the FD. Finally, we reuse a ‘cloud’ visualization from TagSEA that shows how tags are used.
- *Configuring and pruning*: Configuration and pruning are now integrated. The configuration interface is based on the FD. Clicking on a feature allows the user to toggle it from deselected to selected and conversely (see Figure

²See <http://www.splot-research.org>

³See <http://gdansk.uwaterloo.ca:8088/SPLLOT/sxfrm.html>

```

import java.util.NoSuchElementException;
/*@feature:simple_echo.additional_features.keep_history@**/!file!@*/
public class History {

    private Vector<String> history;
    /*@feature:simple_echo.additional_features.count_words@*/
    private int word_count;

    public History() {
        history = new Vector<String>();
    }

    /*@feature:simple_echo.additional_features.count_words@*/
    public void computeWordsCount(String echo) {
        String[] words_msg = echo.split(" ");
        word_count = words_msg.length + word_count;
    }

    public void addEcho(String echo) {
        /*@feature:simple_echo.additional_features.count_words@*/
        computeWordsCount(echo);
        history.add(echo);
    }

    public String getLast() {

```

Fig. 3. Code highlighting in XToF

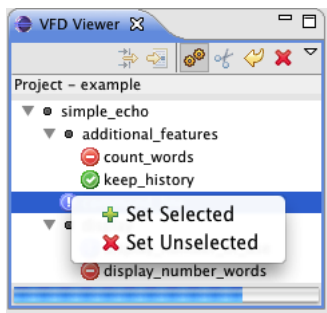


Fig. 4. Product configuration in XToF

4). Each decision made on the diagram is propagated by SPLAR to ensure the validity of the configuration. Once configuration is completed, the mission-specific implementation can be generated. To do this, XToF will clone the project to a new one with a name provided by the user. It will then prune the source code and remove code according to the valid list of features.

- *Portability*: XToF takes advantage of the plug-in platform provided by Eclipse to support other languages than Java. Two languages are currently supported: Java and C.

C. On-going and future work

Additional functionalities will be implemented in the future:

- *Editing the FD*: The current version of XToF does not support editing of the FD. The objective would not only be to create it from the same front-end as the rest of the functionalities, but also maintain the consistency between the FD and the feature tags. For example, if one modifies the name of a feature, each tag that uses the feature will be modified too, thereby supporting co-evolution of the FD and the program.
- *High level visualization*: The current visualization is limited. It is hard to determine which files and packages are tagged with given features (although not impossible if one reads the list of tagged blocks). XToF will provide high level visualization to answer the questions: “Which files and packages are associated to a set of features and

conversely?” and “Which packages or classes (in the case of an OO project) have features in common?” XToF will display links between the files (or classes), packages and features, and provide mechanisms to restrict the view to a set of features or files.

- *Improve declaration tagging*: When the partner company used the first prototype, they reported that one of the major sources of errors was incorrect tagging of variable, function and type declarations [2]. This can occur, for example, when a variable tagged with feature *A* is required by a feature *B* without the developer updating the tag. The product generated with feature *B* selected but feature *A* deselected will not compile. To prevent such an error, the ‘using’ feature must declare the variable, or the ‘declaring’ feature must always be present with the ‘using’ feature. XToF will help the user avoid such issues by offering a pruning based on the set of features that are always present with one selected feature [2]. Another possibility that we will investigate is to take into account dependencies in the code for automatic generation of tags, in the spirit of change-oriented programming [8].

IV. CONCLUSION

In this paper, we introduced XToF, a tool prototype supporting tag-based product line implementation in Java and C. XToF is an extension of a toolchain that was initially developed as joint university-industry project and which has been deployed in the company. XToF will supersede this toolchain by improving it in various ways: better tool integration, visualization, portability to other programming languages and on-the-fly tag generation. Once finished, XToF will provide an integrated tool to create a feature diagram, develop a tagged ‘maximal’ product, navigate through the features and the tagged blocks of code, classes and packages (in the case of OO programs), support feature-based configuration and generate products automatically by pruning the source code.

V. ACKNOWLEDGEMENTS

This work is sponsored by the Interuniversity Attraction Poles Programme of the Belgian State, Belgian Science Policy (MoVES project), FEDER, BNB and FNRS.

REFERENCES

- [1] MISRA, *MISRA-C: Guidelines for the use of the C language in critical systems*. Motor Industry Research Association, 2008.
- [2] Q. Boucher, A. Classen, P. Heymans, A. Bourdoux, and L. Demonceau, “Tag and prune: A pragmatic approach to software product line implementation,” PRECISE Research Centre, Univ. of Namur, Tech. Rep., 2009.
- [3] K. Pohl, G. Böckle, and F. J. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.
- [4] C. Kästner, S. Apel, and M. Kuhlemann, “Granularity in software product lines,” in *Proceedings of ICSE '08*, 2008.
- [5] M.-A. Storey, L.-T. Cheng, I. Bull, and P. Rigby, “Shared waypoints and social tagging to support collaboration in software development,” in *Proceedings of CSCW '06*, 2006.
- [6] —, “Waypointing and social tagging to support program navigation,” in *In Proceedings of CHI '06*, 2006.
- [7] M. Mendonca, M. Branco, and D. Cowan, “S.P.L.O.T.: Software product lines online tools,” in *Proceeding of OOPSLA'09*, 2009.
- [8] P. Ebraert, A. Classen, P. Heymans, and T. D’Hondt, “Feature diagrams for change-oriented programming,” in *Proceedings of ICFI'09*, 2009.