Graphical User Interfaces in Dynamic Software Product Lines

Dean Kramer, Samia Oussena, Peter Komisarczuk
School of Computing and Technology
University of West London
London, United Kingdom
firstname.lastname@uwl.ac.uk

Tony Clark
School of Engineering and Information Sciences
Middlesex University
London, United Kingdom
t.n.clark@mdx.ac.uk

***Abstract*—Dynamic Software Product Line Engineering has gained interest through its promise of being able to unify software adaptation whereby software can be configured at compile time and runtime. While previous work has concentrated on language support and other platform support, little attention has been placed on graphical user interface variability. In this paper, we present the motivation for handling dynamic graphical user interface variability and the challenges that require tackling to enable this.**

***Index Terms*—Software Product Line Engineering, Dynamic Software Product Lines, Graphical User Interfaces**

I. Introduction

Smart phones in recent years have seen high proliferation, allowing more users to stay productive while away from the desktop. It has become highly predictable for these devices to have an array of sensors including GPS, accelerometers, digital compass, proximity sensors, sound etc. Using these sensors with other equipment already found in phones, a wide set of contextual information can be acquired.

This contextual information can be used in Context-Aware Self Adaptive (CASA) software. This software can monitor different contextual parameters and dynamically adapt at runtime to satisfy the user's current needs [1]. These behavioural variations can be seen to share similarities with features in Software Product Lines (SPL), where product commonality and variability is handled, providing higher asset reuse. Within SPLs, Feature Oriented Software Development (FOSD) has emerged as a method for modularising the features of a system [2]. The one fundamental difference between these two concepts is that while SPLs conventionally manage static variability which is handled at compile time, adaptive software requires dynamic variability to be handled at runtime.

Dynamic Software Product Lines (DSPL) enables the SPL to be reconfigurable at runtime [3]. By using DSPLs, variability can be static, adapted at compile time, or dynamic and adapted at runtime. This allows for greater reuse as variability can be implemented for both static and dynamic adaptation, as different products may require the adaptation to be applied at different times [4]. Previous work has enabled the program logic refinements to be handled at runtime, but there is little known how to handle graphical user interface (GUI) variability statically or dynamically. In this paper, we consider the problem of GUI variability in the context of DSPLs and explore some of the challenges that require attention, and provide insight into our research direction.

The remainder of this paper is structured as follows: Motivation to our research is presented in Section 2. In Section 3, we outline the challenges we aim to tackle in our research. Related work is then discussed in Section 4. Final discussion and conclusion is then presented in Section 5.

II. Motivation

When using SPLs, variability and commonality are expressed in terms of *features*. A feature of an SPL has been defined as *"a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems"* [5]. Features can be implemented in feature modules, incrementing software functionality [6], by extending the base feature of a program with additional logic.

Functionality in terms of program logic may not be the only crosscutting software artefact in a SPL. Just like specific business logic may be associated with a given feature, so too can GUIs or GUI refinements. By handling GUI variability within features, greater reuse can be achieved with this type of artefact.

Unlike conventional SPLs, DSPLs can reconfigure at runtime, driven by context changes. Context can be described as "*information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves*" [7]. On reconfiguration, the running application is altered by the addition and removal of program logic. Additionally the GUI may need to be altered to suit the new configuration. Without altering the GUI, events caused by user interaction are altered without the user being aware. By altering the GUI also, the user can be more aware about how the action has changed due to a context change.

A. Scenario Application

To illustrate our motivation, consider a content store application for a mobile device as an example DSPL. This application may provide different content for the user including applications, movies, music etc. Different content is organised into

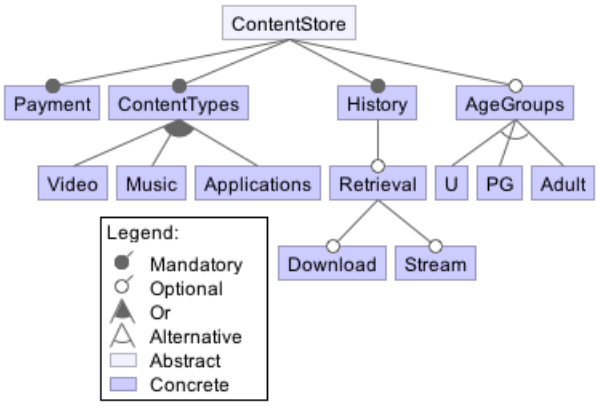

Fig. 1. Scenario Application

different categories. A simplified feature model of the DSPL can be seen in Figure 1. This application provides content for different age groups, and also the application can be tailored to suit these different groups.

Different contexts can affect this system, including contexts regarding the user, including age, and user preferences. Other contexts that can affect the system include device contexts, including connection type currently active, and amount of storage available on the device. As an example, if the current level of free storage on the device is below a particular level, content is streamed and is not downloadable. This change should be reflected in the GUI by only having a streaming button visible. Also, distribution of certain content types may not be allowed in certain counties, because of content licensing etc. Because of this, the GUI can be changed to represent that, by only having buttons for content types that are available. Lastly, since this store can bring content to multiple age groups, depending on the age group, different GUI styles can be used and content that is available to them.

III. Challenges

As stated above, DSPL work has predominantly investigated how to handle logic changes. Therefore the methods used are not well suited to GUI variability. In this section we outline the main challenges that our research attempts to resolve.

A. GUI Implementation

In mobile and web application development, it is common to not have the entire application developed using a single language or technology. Because of the heterogeneous nature of mobile devices, it is suggested that GUIs should not be programmed along with business logic. This is to help alleviate some of the difficulties resulting from tailoring to suit multiple device screen sizes and device capabilities. Also, when developing user interfaces along with business logic, code is less maintainable and is harder to reuse when interface elements can be reused in multiple places of an application.

To improve this maintainability and reuse, the Model-View-Controller pattern has been proposed. The Model-View-Controller (MVC) design pattern [8] aims to improve development of graphical applications by separating data from its representation, and how the user interacts with it.

Within recent years, we have seen the emergence of GUI representation being implemented using documents instead of code [9]. Using this approach, GUI representation is implemented in a more declarative fashion, commonly in markup based languages [10]. Examples of these languages includes Mozilla XUL, QML used in QT, Microsoft XAML, Apple Nib, and Android XMLBlock.

In existing approaches, unified refinement implementation has been proposed [4], [11]. This allows the developer to implement variability using a single technique or technology, which then can be composed statically or dynamically. The same should be true also for GUIs, whereby a single refinement can be applied statically at compile time or dynamically at runtime, without the need for a different implementation technique or technology. Our goal is to achieve this, while using GUI documents in the implementation.

B. Composition Approaches

When considering these user interfaces, one challenge is how composition should be carried out. For composition we see predominantly four different paths including compile-time, runtime, pattern and code transformation, and a hybrid of paths.

Compile-time composition is the process of deriving all foreseeable variations of a given resource at or before application compilation. Then at runtime, when a given GUI resource is needed, depending on the DSPL configuration, the correct resource variant is used. Using this method, mainly different variations of the same GUI are added to the application to be chosen at runtime. While this method can lead to faster reconfiguration, because no composition is required, it can lead to potential scalability issues regarding the time it takes to compile the application, and also the size of the application if a product contains a high amount of variability.

Runtime composition is the process of deriving a correct variation of a given GUI resource at runtime. By using runtime composition, the program is compiled with the GUI refinements for each of the selected features. Then at runtime, depending on the configuration, each of the refinements from the active features are composed together to obtain that GUI. This method unlike the compile-time method removes the possible exponential amount of GUI variations needing to be installed with the program. On the other hand, there is then a static overhead involved in this method, because all tools needed to compose the artefacts need to be included with the application. Also, with some platforms, some GUI artefacts require preprocessing during compilation, therefore requiring these be included too, which in some cases is not possible due to proprietary development kits.

While the MVC pattern can aid the developer by enabling separation of concerns and easier maintainability, it is not the only way a GUI can be created. Many platforms do allow GUI creation in the application programming language. Because of this we can transform the MVC pattern to the Model-View

(MV) pattern. By transforming this pattern, we have to process that GUI refinement and output the appropriate code. This code can then be included in the controller implementation. While this method will not incur the application size bloating found in the compile-time method, or the overhead seen in the runtime composition method, there are other potential issues regarding how the application handles multiple screen types, lack of software reuse, and platform compatibility.

Finally, it is possible to foresee a hybrid approach that combines more than one of the previous stated methods. Using this method, we can combine for example compile-time and pattern transformation to be capable of handling different configuration timing, a challenge discussed next. This method also may prove a method of reducing some of the exponential bloat caused by a purely compile-time only approach.

C. Configuration Timing

When a DSPL configuration alters, the running application is adapted. In previous work this has been handled either by the weaving or removing aspects [12], adding or removing decorators [11] etc. These changes are carried out straight away and can affect the system at the moment of reconfiguration. With GUI changes, depending on the granularity of the changes, it may not be best for every refinement to be added at anytime.

Within the lifecycle of a GUI, there are two main phases in which a refinement should be applied, on *inflation* or while it is *active*. The inflation of a GUI can be regarded as when the GUI is being constructed during the transition from one screen to another. An example taken from the scenario is a transition going from a screen showing a list of Horror movies to a screen showing the detail for a movie chosen from that list. In contrast, the active phase of a GUI is while the screen or GUI is currently active and visible to the user.

If we consider the scenario application presented earlier, when buying content that cannot be instantly downloaded because of its size and the device connection type, it could be useful for instead of just altering the behaviour of the download button, but to also update its label to represent the altered behaviour.

D. GUI Checking

As the MVC pattern proposes the separation of the model, view, and controller, this means that there are multiple linked artefacts that may contain variability. Because of this, it is important to make sure that no inconsistencies between the different elements occur. Examples of these errors include attempting to add an event listener to a non existent button, attempting to alter the appearance of GUI element, or even declaring an event listener that is not implemented within that controller. These can happen due to view lookups not being statically checked at compilation.

While checking for program bugs is an important aspect, the GUI should also be checked for unintended and unwanted changes. Examples of these types of unwanted changes include elements being moved to unwanted places because the addition or removal or a GUI widget. This type of check has become a research interest in Feature Oriented Programming (FOP). Methods of product verification have been proposed in FOP including the use of the Java Modelling Language (JML) [13]. By using JML, formal program specifications within the source code could be added, particularly allowing the addition of *method contracts* and *class invariant*. Method Contracts consist of preconditions that state what the method caller needs to ensure, and the post conditions that state what the method needs to ensure. Class invariants are properties that need to be maintained. Following this approach, there is room to see if a similar method can be applied across multiple differing document types.

IV. Related Work

Some work has looked at the variability of user interfaces, but GUI research within the scope of SPLs has been limited. In [14], a case study was carried out to investigate the GUI variability found in a commercial web-based information system. An analysis tool based on Selenium was used to extract data from the HTML and CSS documents mapping elements to the reference application. Their results show large amounts of non trivial variability in the GUIs, of which much could not be handled by stylesheet changes, with not one GUI element being used.

Other work considers how to re-engineer configurators [15]. In this work, the authors present challenges regarding the reverse engineering of existing configurators analysing GUI, webpage source, and code base to extract variability information. The second challenge was then regarding forward engineering and generating a tailored GUI and codebase.

To summarise, little attention has been paid to GUI variability, while what work has been done, has been a concentration only on static variability. In our research we wish to go further, and achieve the ability of handling this variability statically and dynamically.

A. DSPL Implementation Support

There has been two main methods proposed to help support DSPL implementation, by language extensions, or by larger services and component architectures.

Language support including Feature Oriented Programming (FOP) languages like FeatureC++ [11] have been proposed. In FeatureC++, logic within the dynamic feature modules is modularised using the decorator pattern. Using this pattern, decorators wrap classes at runtime to alter the behaviour of an application. Other work has included the emergence of Delta-Oriented Programming (DOP), particularly DeltaJava [16] and its dynamic form, Dynamic Delta Oriented Programming (DDOP) [17]. In DOP, feature refinements are implemented in *Delta Modules*. A delta module though similar to feature modules in that it can increment a base program functionality, it can also remove functionality. Delta Modules are implemented as single files, instead of each class refinement being a class file as in FOP.

While these language extensions bring very good support for runtime business logic and class member changes, they

support a single language only. Our work aims at considering GUI implementation using multiple languages, including GUI documents.

Service Oriented Architectures (SOA) recently has been a popular domain for DSPL research [4], [18], [19]. In [18], a mobile DSPL was proposed, but only considers GUI during screen transitions, and has no consideration for MVC pattern. Parra proposed a unified approach to implementing logic variability in [4], whereby it should not matter if the variability is compile time or runtime, it should be seen and implemented as the same, an approach we intend to follow for the GUI.

Other more general FOSD language tools included FeatureHouse [20], which brings FOSD to multiple languages and artefacts. FeatureHouse also does not require specific new keywords and syntax that is required by other FOP and DOP languages. Furthermore, FeatureHouse can be extended to new languages by providing a language grammar in FeatureBNF. This solution allows greater support for handling different languages and documents but has only been used at compile time, not at runtime.

V. Discussion and Conclusion

This paper discusses the need to handle graphical user interface (GUI) variability at runtime in dynamic software product lines (DSPL), and its related challenges. A tool for handling compile-time composition has been implemented to produce all runtime variations of the GUI layouts, but it currently does not handle different configuration times, only functioning at GUI inflation and not while the screen is active. Configuration timing is planned to be bound to features via feature attributes, using extended feature models. Then when the DSPL product is compiled, depending on the configuration timing, different code is generated for the respective controllers to handle these changes. Active time configuration using runtime or compile-time composition will require the screen to re-inflate. Consideration will need to be taken to avoid GUI state loss on re-inflation.

Our next plan is to develop a tool to handle pattern transformation, taking GUI refinements and generate the code needed to create the GUI directly in the controller including all references to event listeners. To handle the different configuration times, logic will have to be generated to add, remove, and alter GUI elements within the view hierarchy. Following this we want to compare each of the approaches using a case study. We believe that with this research we can improve how GUIs are implemented and handled in a DSPL.

References

[1] L. M. Daniele, E. Silva, L. F. Pires, and M. Sinderen, "A soa-based platform-specific framework for context-aware mobile applications," in *Enterprise Interoperability*, ser. Lecture Notes in Business Information Processing, W. Aalst, J. Mylopoulos, M. Rosemann, M. J. Shaw, C. Szyperski, R. Poler, M. Sinderen, and R. Sanchis, Eds. Springer Berlin Heidelberg, 2009, vol. 38, pp. 25–37.

[2] D. Batory, J. Sarvela, and A. Rauschmayer, "Scaling step-wise refinement," *IEEE Trans. Softw. Eng.*, vol. 30, pp. 355–371, June 2004.

[3] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid, "Dynamic software product lines," *Computer*, vol. 41, pp. 93–95, April 2008.

[4] C. Parra, "Towards dynamic software product lines: Unifying design and runtime adaptations," Ph.D. dissertation, INRIA Lille Nord Europe Laboratory, March 2011.

[5] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," *Technical Report CMU/SEI-90-TR-21*, 1990.

[6] M. Rosenmuller, "Towards flexible feature composition: Static and dynamic binding in software product lines," Ph.D. dissertation, Otto-von-Guericke-University Magdeburg, June 2011.

[7] G. D. Abowd, A. K. Dey, P. J. Brown, N. Davies, M. Smith, and P. Steggles, "Towards a better understanding of context and context-awareness," in *Proceedings of the 1st international symposium on Handheld and Ubiquitous Computing*, ser. HUC '99. London, UK: Springer-Verlag, 1999, pp. 304–307.

[8] G. E. Krasner and S. T. Pope, "A cookbook for using the model-view controller user interface paradigm in smalltalk-80," *J. Object Oriented Program.*, vol. 1, no. 3, pp. 26–49, Aug. 1988.

[9] D. Draheim, C. Lutteroth, and G. Weber, "Graphical user interfaces as documents," in *Proceedings of the 7th ACM SIGCHI New Zealand chapter's international conference on Computer-human interaction: design centered HCI*, ser. CHINZ '06. New York, NY, USA: ACM, 2006, pp. 67–74.

[10] J. Kim and C. Lutteroth, "Multi-platform document-oriented guis," in *Proceedings of the Tenth Australasian Conference on User Interfaces - Volume 93*, ser. AUIC '09. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2009, pp. 27–34. [Online]. Available: http://dl.acm.org/citation.cfm?id=1862703.1862707

[11] M. Rosenmüller, N. Siegmund, M. Pukall, and S. Apel, "Tailoring dynamic software product lines," *SIGPLAN Not.*, vol. 47, no. 3, pp. 3–12, Oct. 2011.

[12] C. Parra, X. Blanc, and L. Duchien, "Context awareness for dynamic service-oriented product lines," in *SPLC '09: Proceedings of the 13th International Software Product Line Conference*. Pittsburgh, PA, USA: Carnegie Mellon University, 2009, pp. 131–140.

[13] T. Thüm, I. Schaefer, S. Apel, and M. Hentschel, "Family-based deductive verification of software product lines," in *Proceedings of the 11th International Conference on Generative Programming and Component Engineering*, ser. GPCE '12. New York, NY, USA: ACM, 2012, pp. 11–20. [Online]. Available: http://doi.acm.org/10.1145/2371401.2371404

[14] A. Pleuss, B. Hauptmann, M. Keunecke, and G. Botterweck, "A case study on variability in user interfaces," in *Proceedings of the 16th International Software Product Line Conference - Volume 1*, ser. SPLC '12. New York, NY, USA: ACM, 2012, pp. 6–10.

[15] Q. Boucher, E. Abbasi, A. Hubaux, G. Perrouin, M. Acher, and P. Heymans, "Towards more reliable configurators: A re-engineering perspective," in *Product Line Approaches in Software Engineering (PLEASE), 2012 3rd International Workshop on*, june 2012, pp. 29 –32.

[16] I. Schaefer, L. Bettini, F. Damiani, and N. Tanzarella, "Delta-oriented programming of software product lines," in *Proceedings of the 14th international conference on Software product lines: going beyond*, ser. SPLC'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 77–91.

[17] F. Damiani and I. Schaefer, "Dynamic delta-oriented programming," in *Proceedings of the 15th International Software Product Line Conference, Volume 2*, ser. SPLC '11. New York, NY, USA: ACM, 2011, pp. 34:1–34:8.

[18] F. Marinho, F. Lima, J. Ferreira Filho, L. Rocha, M. Maia, S. de Aguiar, V. Dantas, W. Viana, R. Andrade, E. Teixeira, and C. Werner, "A software product line for the mobile and context-aware applications domain," in *Software Product Lines: Going Beyond*, ser. Lecture Notes in Computer Science, J. Bosch and J. Lee, Eds. Springer Berlin / Heidelberg, 2010, vol. 6287, pp. 346–360.

[19] H. Gomaa and K. Hashimoto, "Dynamic software adaptation for service-oriented product lines," in *Proceedings of the 15th International Software Product Line Conference, Volume 2*, ser. SPLC '11. New York, NY, USA: ACM, 2011, pp. 35:1–35:8.

[20] S. Apel, C. Kastner, and C. Lengauer, "Featurehouse: Language-independent, automated software composition," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 221–231.