Summary of: On Checking Delta-Oriented Software Product Lines of Statecharts

Summary of: On Checking Delta-Oriented Software Product Lines of Statecharts

Michael Lienhardt[1], Ferruccio Damiani[2][0000−0001−8109−1706], Lorenzo Testa[2], and Gianluca Turin[3]

[1] ONERA —The French Aerospace Lab, France (michael.lienhardt@onera.fr)
[2] University of Turin, Turin, Italy (ferruccio.damiani@unito.it)
[3] University of Oslo, Oslo, Norway (gianlutu@ifi.uio.no)

Abstract. A Software Product Line (SPL) is a set of programs, called variants, which are generated from a common artifact base. Delta-Oriented Programming (DOP) is a flexible approach to implement SPLs. This short paper summarises the contributions published in [10]. A foundation for rigorous development of delta-oriented product lines of statecharts is provided by defining: a core language for statecharts, DOP on top of it, an analysis ensuring that a product line is well-formed (i.e., all variants can be generated and are well-formed statecharts). An implementation of the analysis has been applied to an industrial case study.

Keywords: Core calculus · Delta-oriented programming · Software product line analysis · Statechart.

1 Background

A *Software Product Line* (SPL) is a set of programs, called *variants*, which have well documented variability and are generated from a common artifact base [6]. *Delta-Oriented Programming* (DOP) [11, 5] [4, Sect. 6.6.1] is a flexible approach to implement SPLs. A delta-oriented SPL consists of a *feature model*, an *artifact base*, and *configuration knowledge*. The feature model provides an abstract description of variants in terms of *features*—each feature represents an abstract description of functionality and each variant is identified by a set of features, called a *product*. The artifact base provides language dependent artifacts that are used to build the variants—it consists of a *base program* (that might be empty or incomplete) and of a set of *delta modules* (*deltas* for short), which are containers of modifications to a program. For example: for Java programs, a delta can add, remove or modify classes and interfaces; for statechart programs, a delta can add, remove or modify states and transitions. Configuration knowledge connects the features in the feature model with the artifacts in the artifact base by associating to each delta an *activation condition* over the features and specifying an *application ordering* between deltas. Once a user selects a product, the corresponding variant is derived by applying the deltas with a satisfied activation condition to the base program according to the application ordering. Thus

configuration knowledge defines a mapping from products to variants, and DOP supports the automatic generation of variants based on a selection of features.

SPL analysis approaches can be classified into three main categories [13]: product-based, family-based and feature-based. *Product-based* analyses work only on generated variants (or models of variants). *Family-based* analyses work on the artifact base, without generating any variant or model of variant, by exploiting feature model and configuration knowledge to derive results about all variants. *Feature-based* analyses work on the reusable artifacts in the artifact base (base program and deltas in DOP) in isolation, without using feature model and configuration knowledge, to derive results on all variants.

2 Contributions of [10]

The toolchain of the HyVar project [2] supports the development of delta-oriented SPLs of statecharts [8] expressed in the format supported by YAKINDU STATECHART TOOLS [3]. A YAKINDU statechart consists of: an *interface definition part*, which declares the elements (e.g., events and typed operations) used by the statetechart to interact with the external environment; and a *state definition part*, which defines the structure of the statechart (i.e., a hierarchical state machine that can use the elements declared in the interface definition part). This toolchain has been used to develop product lines of car embedded software systems [2]. It provides: automatic derivation of a statechart variant, C/C++ and Java code generation, linking to external code artifacts, compilation, and support for guaranteeing that all the statechart variants can be generated and are well formed.

In delta-oriented programming, the generation of a variant fails when attempting to apply a delta that contains an operation that cannot be executed (e.g., for an SPL of YAKINDU statecharts, adding an already existing event to the interface definition part, or removing or modifying a non existing state in the state definition part).

In YAKINDU STATECHART TOOLS, a statechart is well formed if the interface definition part is well-formed (e.g., there are no duplicated declarations) and the state definition part is well formed, that is: (i) there are no structural flaws (like, e.g., a dangling transition); (ii) all the elements used to interact with the external environment are declared in the interface definition part; and (iii) the use of each of these elements is correct with respect to its declaration (e.g., each operation is used according to the type declared for it).

The paper [10] provides a formal account of the SPL family-based analysis technique implemented in the HyVar toolchain. Namely, it:

1. defines FEATHERWEIGHT STATECHART LANGUAGE (FSL), a core textual language that captures the key ingredients of YAKINDU statecharts (much as Featherweight Java [9] captures the key ingredients of class-based object-oriented programming);
2. formalizes for FSL (by a means of a set of typing rules) the well-formedness checks supported by YAKINDU STATECHART TOOLS;

3. defines Featherweight Delta Statechart Language (FDSL), a core textual language for delta-oriented SPLs where variants are written in FSL, that captures the key ingredients of the delta operations on Yakindu statecharts supported by the HyVar toolchain;
4. defines (on top of the formalization in points 1, 2 and 3 above) a family-based analysis for guaranteeing that all the variants can be generated and are well formed; and
5. illustrates how the implementation of the analysis in the HyVar toolchain has been applied on the HyVar case study.

The Yakindu statecharts language is defined as an ECore metamodel [1]. In the HyVar toochain, the language of deltas on Yakindu statecharts is defined by the DeltaEcore tool suite [12], which supports developers in defining delta languages for Ecore metamodels. The core languages FSL and FΔSL, which capture the key ingredients of delta-oriented programming on Yakindu statecharts, have been designed in order to enable providing a formal account of the SPL analysis (cf. point 4 above).

The proposed family-based well-formedness checking mechanism for delta-oriented SPL of FSL statecharts is inspired by the type checking approach for delta-oriented SPLs of Java-like programs proposed in [7]. In [7], starting from a set of typing rules for IFJ [5] (an imperative version of Featherweight Java [9]), it is shown how to define a family-based type-checking analysis for SPLs written in IFΔJ (a language for delta-oriented SPLs of IFJ programs). In order to enable using the technique proposed for IFΔJ SPLs to define a family-based well-formedness analysis for SPLs written in FΔSL, the notion of well-formed FSL statechart has been formalized by a means of a set of typing rules. Since the structure of an FSL statechart is more complex than the structure of an IFJ program, the technique proposed in [10] had to address this additional complexity. In particular:

- An IFJ program has only classes and attributes, while statecharts have a recursive structure where composite states can contain composite states that can contain states themselves—in the formalization of the analysis this has been addressed by introducing a notion of path to identify where an element is placed in a statechart.
- The elements of an IFJ program have only one dependency slot (classes depend on their super classes, and fields and methods depend on the types and method they use in their declaration), while the elements in a statechart have a more fine grain structure where several parts can be changed (for instance, each part of a transition can be changed)—in the formalization of the analysis this has been addressed by introducing the notion of dependency slots.

3 Conclusion and Future Work

The paper [10] originated in the context of the HyVar project, while enhancing the preliminary version of the HyVar toolchain (that supported delta-oriented

SPLs of YAKINDU statecharts) by adding support for an SPL analysis that automatically checks that all the variants can be generated and are well formed. The paper [10] provides a formal account of the well-formedness SPLs analysis technique integrated into the toolchain, and illustrates how the analysis has been applied to an industrial case study.

In future work we would like to further evaluate the implementation by considering other case studies. We also plan to define other static analyses for delta-oriented SPL of YAKINDU statecharts (like, e.g., model checking) and to incorporate them into the HyVar toolchain.

References

1. Eclipse Modeling Framework (EMF). www.eclipse.org/modeling/emf/
2. The HyVar home page. www.hyvar-project.eu
3. Yakindu statechart tools. www.itemis.com/en/yakindu/state-machine/
4. Apel, S., Batory, D., Kästner, C., Saake, G.: Feature-Oriented Software Product Lines: Concepts and Implementation. Springer (2013)
5. Bettini, L., Damiani, F., Schaefer, I.: Compositional type checking of delta-oriented software product lines. Acta Informatica **50**(2), 77–122 (2013). https://doi.org/10.1007/s00236-012-0173-z
6. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison Wesley Longman (2001)
7. Damiani, F., Lienhardt, M.: On type checking delta-oriented product lines. In: Integrated Formal Methods: 12th International Conference, IFM 2016. Lecture Notes in Computer Science, vol. 9681, pp. 47–62. Springer International Publishing, Cham (2016). https://doi.org/10.1007/978-3-319-33693-0_4
8. Harel, D.: Statecharts: a visual formalism for complex systems. Science of Computer Programming **8**(3), 231 – 274 (1987). https://doi.org/10.1016/0167-6423(87)90035-9
9. Igarashi, A., Pierce, B., Wadler, P.: Featherweight Java: A minimal core calculus for Java and GJ. ACM TOPLAS **23**(3), 396–450 (2001). https://doi.org/10.1145/503502.503505
10. Lienhardt, M., Damiani, F., Testa, L., Turin, G.: On checking delta-oriented product lines of statecharts. Science of Computer Programming **166**, 3 – 34 (2018). https://doi.org/10.1016/j.scico.2018.05.007
11. Schaefer, I., Bettini, L., Bono, V., Damiani, F., Tanzarella, N.: Delta-oriented programming of software product lines. In: Software Product Lines: Going Beyond - 14th International Conference, SPLC 2010. Lecture Notes in Computer Science, vol. 6287, pp. 77–91. Springer (2010). https://doi.org/10.1007/978-3-642-15579-6_6
12. Seidl, C., Schaefer, I., Aßmann, U.: Deltaecore - A model-based delta language generation framework. In: Modellierung 2014, 19.-21. März 2014, Wien, Österreich. LNI, vol. 225, pp. 81–96. GI (2014), `http://subs.emis.de/LNI/Proceedings/Proceedings225/article2.html`
13. Thüm, T., Apel, S., Kästner, C., Schaefer, I., Saake, G.: A classification and survey of analysis strategies for software product lines. ACM Comput. Surv. (2014). https://doi.org/10.1145/2580950