In Proceedings of the 31st International Conference on Software Engineering (ICSE), Companion Volume, New Ideas and Emerging Results (NIER) Track, Vancouver, Canada, May 16-24, 2009.

Towards Safer Composition

Andreas Classen[†*] Patrick Heymans[†] Thein Than Tun[†‡] Bashar Nuseibeh[‡]

[†]PReCISE Research Centre, Faculty of Computer Science, University of Namur - FUNDP
5000 Namur, Belgium
{acs,phe,ttu}@info.fundp.ac.be

[‡]Centre for Research in Computing, The Open University
Walton Hall, Milton Keynes MK7 6AA, UK
{t.t.tun,b.nuseibeh}@open.ac.uk

Abstract

Determining whether a set of features can be composed, or safe composition*, is a hard problem in software product line engineering because the number of feature combinations can be exponential. We argue that synergies between current approaches to safe composition should be exploited and propose a combined approach. At the heart of our proposal is a merge operation that creates a behavioural description for the entire product family from a feature diagram and descriptions of individual feature behaviour. As a result, we intend to verify more efficiently safe composition for an exponential number of feature combinations.*

1 Introduction

> *"Having started with one problem, we must end with one solution. So the concerns we have separated must be recombined: having divided to conquer, we must then reunite to rule."* [8]

Decomposition has long been advocated as a means to deal with the complexity of software intensive systems [1]. While decomposition allows to reduce the complexity by breaking a big problem into small and understandable sub-problems, it does so at the expense of later steps in the development where the solutions to the various sub-problems need to be composed. Composition thus becomes another fundamental problem of software engineering. *Safe composition* [20] is the problem of deciding whether the solutions to the sub-problems (such as components or aspects) are valid with respect to a specific objective. More generally, since there may be different possible combinations of sub-problems, checking safe composition is to determine whether the compositions of the solutions to sub-problems are valid for all intended combinations. The problem is hard, since the number of intended combinations might be exponential for the number of sub-problems. Approaches to decomposition that guarantee safe composition from the start need to make assumptions about the nature of the decomposition. Here, no such assumptions are made.

In previous work [3], we proposed a conceptual solution to the problem of *safe composition*. Unfortunately, the solution, as many other solutions, is not efficient because of the exponential number of combinations it explicitly considers. Other solutions suffer from this same restriction [12, 18, 13], or tend to rely on the questionable assumption that there is a single model encompassing all sub-problems [6, 5]. In this paper, we identify synergies between these methods (Section 2) and propose a combined approach, free from most existing drawbacks (Section 3). This comes, however, at the expense of the generality of our previous work [3] since we restrict the approach to behavioural properties expressed with linear or modal transition systems (Section 4).

More concretely, the different intended combinations of solutions to sub-problems form a software product line [15] (SPL) and are commonly represented with a feature diagram [19] (FD), i.e. a decomposition hierarchy of features that formally expresses the set of intended feature combinations, the *variability* of the SPL. The term *feature* here denotes any type of solution to a sub-problem (a component, an aspect, etc.). At the heart of our proposal is a behaviour-preserving merge operation. The operation, given an FD and behavioural descriptions for each feature, builds an overall behavioural description, encompassing all features and all combinations thereof. The idea is that the resulting model incorporates both behaviour and variability so that each property it satisfies (e.g. some safety property) is also satisfied by all individual feature combinations.

[*]FNRS Research Fellow.

2 Approaches to safe composition

The safe composition problem occurs in various subdisciplines of software engineering, such as aspect-oriented [10, 9], feature-oriented [20, 11, 14], component-based [17] and model-based development [4, 18], and in Software Product Line Engineering (SPLE) [6, 12, 5, 16]. It also occurs in the well-studied problem of feature interaction management [2]. Our proposal draws on the following approaches.

Larsen *et al.* [12] introduce the notion of modal I/O automaton in order to model a configurable component where the refinements represent different configurations. Two configurable components can be safely composed iff it is possible to obtain two full refinements (without modalities) of each component's automaton whose composition is non-empty. Since it considers each feature as a separate entity with its own automaton, then if the number of features grows, each automation remains manageable. However, Larsen *et al.* do not consider how the definition of safe composition generalises to systems consisting of an arbitrary number of features.

Fischbein *et al.* [6] suggest using modal transition systems (MTS) to define the behaviour of the SPL, which can be instantiated into several normal linear transition systems (LTS). Safe composition is considered only implicitly in the sense that if the combined behaviour of some features is an implementation of the SPL, then the composition is safe. Although it becomes possible to reason about the SPL as a whole, the main difficulty lies in the fact that all possible behaviours of the SPL need to be described upfront, and in a single MTS.

In the context of model management, Sabetzadeh *et al.* [18] present an approach for checking the consistency of related models. Consistency checking of a number of distributed models is achieved by first merging them, and then checking the merged model for consistency. They overcome the need to check models in pairs. However, the number of different combinations is still exponential.

Post *et al.* propose a technique called 'lifting', which consists in incorporating the information about allowed combinations (the *variability*) into the verifiable model itself [16]. They use this approach to verify the feature combinations that are possible with the configuration options of the Linux kernel. If transposed to behavioural models, the disadvantage is the same as for [6, 5], since there needs to be one model for all configurations.

3 Towards a combined approach

The approaches introduced in the previous section all have their respective advantages, but also significant drawbacks. Upon closer inspection, however, one can see that the drawbacks of one approach are almost all mitigated by the other approaches. It thus seems that a combined approach could do away with most of those drawbacks.

Indeed, Fischbein *et al.*'s limited modelling scaleability (one big model for the whole SPL) can be overcome by combining the approaches by Larsen *et al.* and Sabetzadeh *et al.*. The behaviour of each feature is specified individually with an LTS, and to check safe composition, the individual LTSs are merged to obtain the system behaviour. Note that each approach addresses a different problem, makes different assumptions and uses its own formalism. Hence, combining them as such might not make much sense. However, we argue that *combining their underlying ideas* is possible and represents a promising idea to resolve our problem. For the merge, we identified four possible strategies depending on two factors.

The first is the *nature of the merge operation*. One alternative is that the merge produces an LTS of the system behaviour that can be model-checked against properties expressing safe composition (the requirements of the features), similar to [18]. Another alternative is that it produces an LTS, which by construction satisfies such properties, similar to [7]. The first option sticks more closely to the idea of model-checking, because it is analytic in nature; it allows to identify an issue, but does not solve it. The second option does not explicitly identify any issues, it avoids them by construction.

The second is the *scope of the merge operation*. The *simple merge*, illustrated in Figure 1(a), consists in merging the behavioural descriptions for a single configuration (set of features), which can be done with the method of Nejati *et al.* [13], for instance. Since the simple merge would need to be executed for every possible combination of features, it still has the disadvantage that there can be an exponential number of such combinations. The *full merge*, illustrated in Figure 1(b), tries to overcome this limitation. Following the ideas of [16], it takes the variability information into account and 'lifts' it into the behavioural model. The resulting model thus describes the behaviour of the whole SPL rather than the behaviour of a single feature combination. More formally, when projected to the actions performed by the features, the traces of the resulting LTS or MTS should be the union of the traces of each feature combination's LTS. Basically, the complexity is moved to the analysis task.

Of the four possible strategies, we decided to investigate one that is analytical and constructs a full merge. Analytical, since a drawback of the "issue-avoiding" approach is that its solution (i.e. the merge operation) is restricted to the model. The model, however, is only an abstraction of the real system, and it is not clear whether the solution found for the model also works for the real system, nor how it can be transposed. Also, an issue-avoiding approach will hide issues that really need to be solved, rather than avoided.

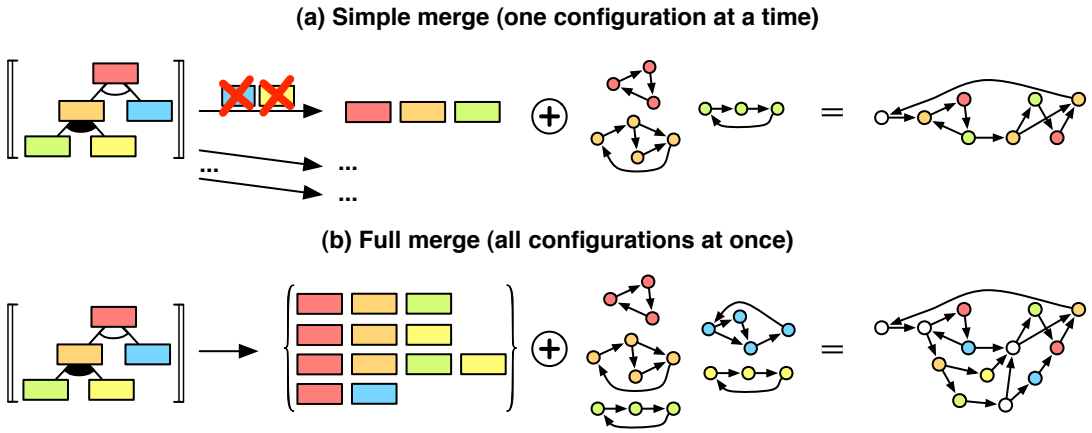

Figure 1. Different scopes of the merge operation.

Full merge, because we believe that such an approach will be more efficient, a thesis that will have to be verified once sufficient progress is made. Indeed, assuming the cost (time and space complexity) of both merge operations to be the same, we conjecture that checking a property on the full merge model is less expensive than calculating the simple merge an exponential number of times and checking the property on each simple merge model.

4 First results and open issues

The elaboration of the approach can be broken down into three (closely related) steps: (1) identify a suitable representation for the feature behaviour, (2) define how to merge the behaviour descriptions of different features, (3) encode the variability information as part of this merge operation.

We assume that the behaviour is specified with an automata-like language, such as the aforementioned LTS or MTS. Since the discussion is still led at a conceptual level, we cannot already settle on a formalism. The feature behaviour should be defined in a way that makes it possible to model each feature individually, since this is essential for the approach to be scaleable. It should also be possible to specify features whose behaviour is not necessarily a refinement of their parent's behaviour (such as, for instance, crosscutting features). At the same time, however, the way in which the feature behaviour is expressed should allow it to be easily merged and "variability-encoded". We discovered that the last two goals are hard to achieve at the same time, as explained in the following paragraphs. The alternatives we identified are the following.

A. Features as refinements of their parents. A feature is an automaton with two designated states (*start* and *end*), and when merged into its parent, expands one of its states (in the sense of hierarchical statecharts). The advantage of this approach is that the merging and variability-encoding tasks become relatively simple, because the impact of the feature is localised. Also, as a feature is self-contained, each can be individually reasoned about. It is, however, rather restrictive in that it does not allow to model crosscutting features, or features that run in parallel to other features.

B. Freely-merged diagrams. A feature is an automaton or part of an automaton, where each state is assigned a textual ID. When merged, the IDs of states are matched against the existing behavioural description, new states are added and then the transitions reconnected. The connected parts of the resulting model are interpreted as individual automata and parallelised. The advantage of this option is its high expressiveness. The disadvantage is that features are not self-contained (since the IDs are universal), and cannot be reasoned about individually. Furthermore, the variability-encoding becomes harder, since the implications of a feature addition are not local anymore.

C. Features as transformations. A feature is a syntactic graph transformation. This approach has the additional advantage of being able to change or delete states and transitions. The disadvantages are the same as for (B).

The merging approach itself was partly defined in the previous three paragraphs. For options (B) and (C), however, a question remains; namely whether the merge is calculated *bottom-up* (sub-features are first merged into their parent and then into the base), *top-down* (first the parent is merged into the base, then its sub-features into the result), or *flat* (features are merged in an arbitrary order). The advantage of a bottom-up approach is that it limits the application scope of a feature since a feature can only affect its parent. However, crosscutting features can only be accommodated in a flat approach, where they are merged last.

Finally, the most crucial part is the encoding of the variability information. It is vital that the full merge is not of

exponential size itself, i.e. the encoding of the variability information should not be explicit, that is, it should not be a mere juxtaposition of the simple merge models. Our preliminary results indicate that one way of achieving this, is to precede the transformation result of a feature f by a special transition pre_f to a new state, that indicates that the feature is about to be executed. From this state, there are two outgoing transitions, one do_f that executes the feature and one $skip_f$ that skips its execution. The allowed feature combinations can then be encoded by observer processes over these three transitions.

In conclusion, the most promising approach appears to be one that combines bottom-up and flat (for crosscutting features only) algorithms based on graph transformations. We believe that the aforementioned disadvantages of features as transformations can be mitigated by the use of transformation patterns (refinement, for instance, would be one pattern), guidelines and appropriate tool support.

5 Conclusion

We presented the problem of safe composition in a general form that current literature fails to recognise: given a set of sub-problems obtained by decomposing a complex problem, and a set of intended combinations of solutions to the sub-problems, are all of these combinations valid with respect to a specific objective? After an examination of current approaches to safe composition, it becomes clear that none of them addresses the problem as a whole, but that most approaches are built on complementary ideas. We thus proposed a combined approach that is free from the initial drawbacks, and that attempts to address the more general safe composition problem. Preliminary results are (1) a survey and characterisation of the merge operation, that is at the heart of our approach, as well as (2) an analysis of ways to express feature behaviour. From there, we are able to precisely describe the shape that the merge operation should have. In addition, this paper demonstrates that *safe composition* is a root problem for a number of existing approaches. Although there is a group of loosely connected researchers working on safe composition, it is our hope that they become a cohesive community in the near future.

Acknowledgements

We thank the anonymous referees for their helpful comments. This work was partially funded by the EPSRC, the Interuniversity Attraction Poles Programme, Belgian State, Belgian Science Policy (as part of the MoVES project), the BNB and the FNRS. We are also grateful for the support of our colleagues at the Open University and at the University of Namur, in particular Germain Saval.

References

[1] G. Booch. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, 1993.

[2] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec. Feature interaction: a critical review and considered forecast. *Computer Networks*, 41(1):115–141, 2003.

[3] A. Classen, P. Heymans, and P.-Y. Schobbens. What's in a feature: A requirements engineering perspective. In *FASE'08, Held as Part of ETAPS'08*, volume 4961 of *LNCS*, pages 16–30. Springer, 2008.

[4] K. Czarnecki and K. Pietroszek. Verifying feature-based model templates against well-formedness ocl constraints. In *GPCE '06*, pages 211–220. ACM Press, 2006.

[5] A. Fantechi and S. Gnesi. Formal modeling for product families engineering. In *SPLC 2008*. IEEE CS, 2008.

[6] D. Fischbein, S. Uchitel, and V. Braberman. A foundation for behavioural conformance in software product line architectures. In *ROSATEA '06, ISSTA 2006 workshop*, pages 39–48. ACM Press, 2006.

[7] J. D. Hay and J. M. Atlee. Composing features and resolving interactions. In *SIGSOFT'00/FSE-8*, pages 110–119. ACM Press, 2000.

[8] M. Jackson. Some complexities in computer-based systems and their implications for system development. *CompEuro'90*, pages 344–351, 1990.

[9] E. Katz and S. Katz. Incremental analysis of interference among aspects. In *Workshop on Foundations of Aspect-Oriented Languages 2008 at AOSD 08*, 2008.

[10] S. Krishnamurthi, K. Fisler, and M. Greenberg. Verifying aspect advice modularly. In *SIGSOFT'04/FSE-12*, pages 137–146. ACM Press, 2004.

[11] C. Kästner and S. Apel. Type-checking software product lines - a formal approach. In *ASE'08*. IEEE CS, 2008.

[12] K. G. Larsen, U. Nyman, and A. Wasowski. Modal i/o automata for interface and product line theories. In *ESOP*, pages 64–79, 2007.

[13] S. Nejati, M. Sabetzadeh, M. Chechik, S. Easterbrook, and P. Zave. Matching and merging of statecharts specifications. In *ICSE'07*, pages 54–64. IEEE CS, 2007.

[14] S. Nejati, M. Sabetzadeh, M. Chechik, S. Uchitel, and P. Zave. Towards compositional synthesis of evolving systems. In *FSE 2008*. ACM Press, 2008.

[15] K. Pohl, G. Bockle, and F. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, July 2005.

[16] H. Post and C. Sinz. Configuration lifting: Verification meets software configuration. In *ASE'08*. IEEE CS, 2008.

[17] A. M. Roldán, E. Pimentel, and A. Brogi. Safe composition of linda-based components. *Electr. Notes Theor. Comput. Sci.*, 82(6), 2003.

[18] M. Sabetzadeh, S. Nejati, S. Liaskos, S. Easterbrook, and M. Chechik. Consistency checking of conceptual models via model merging. In *RE'07*. IEEE CS, 2007.

[19] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps. Feature Diagrams: A Survey and A Formal Semantics. In *RE'06*, pages 139–148. IEEE CS, 2006.

[20] S. Thaker, D. S. Batory, D. Kitchin, and W. Cook. Safe composition of product lines. In *GPCE 2007*, pages 95–104. ACM Press, 2007.