Analysis of Feature Configuration Workflows

Andreas Classen*, Arnaud Hubaux, and Patrick Heymans
PReCISE Research Centre, Faculty of Computer Science,
University of Namur, 5000 Namur, Belgium

E-mail: {acs,ahu,phe}@info.fundp.ac.be

Abstract

We recently introduced feature configuration workflows, *a formalism for modelling the complex configuration processes in software product line engineering. In earlier work we identified obstacles to efficient tool support for which we now outline the main concepts of a solution. These take the form of a set of analysis tasks that can be performed on feature configuration workflows.*

1 Introduction

In software product line engineering [6], feature diagrams (FDs) [4, 7] are used to model the variability of the product line (PL), i.e. its set of products. The *configuration process* is the process of gradually deciding which features of the FD should be included in a product and which should not [6, 1]. It is thus a bridge between the *domain engineering* and the *application engineering* processes.

A configuration process can be a complex and lengthy endeavour [2]. In order to provide a modelling and reasoning framework for this process, we built on our earlier work on formal semantics for FDs [7] and proposed feature configuration workflows (FCWs) [3], a formalism that combines the workflow language YAWL [8] with FDs. FCWs relax the requirement of a striclty sequential configuration process as originally imposed in multi-level staged configuration (MLSC) [1].

An FCW, such as the one shown in Figure 1, is a YAWL workflow where $\boxed{\text{tasks}}$ (such as $\boxed{t}$ or $\boxed{v}$) are annotated with FDs. The configuration process follows the workflow and an FD of the PL can only be configured when the task to which it is linked is executed. Generally, a task is assigned to a stakeholder, and the FD that is configured during the task captures her responsibility. The second kind of node in a workflow is the (condition) (e.g. ▶, ⓤ or ■) which designates a point in time. Linking an FD to a condition means that the FD has to be fully configured when the condition is reached—it is the FD's stop.

*FNRS Research Fellow.

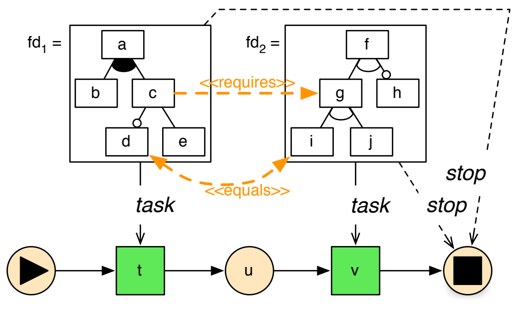

Figure 1. An example FCW.

The individual FDs are actually modules of an overall FD, and there are usually links between the features of different modules, such as the $\ll$*equals*$\gg$ link between features d and i in Figure 1. Since the task and the stop of an FD are not necessarily connected, this allows to have situations in which a configuration decision is postponed. The decision of including or excluding the feature d, for instance, does not have to be taken during task $\boxed{t}$, because the person responsible of task $\boxed{v}$ is also able to make this decision, her feature i being equal to d.

Although the principal implementation strategies for FCWs are known [3], there is one problem that needs to be solved before an efficient implementation can be provided. It is to make sure that a decision is only postponed if we are certain that it will be taken at a later time. The decision of whether or not to select feature b during task $\boxed{t}$, for instance, cannot be postponed, since there is no way this choice could be made afterwards, meaning that the task's stop condition ■ would not be satisfied.

2 Module types: towards a normal form

A first way of preventing unsatisfiable stop conditions is to prevent them when the model is being constructed. This way, they cannot occur when the workflow is executed. To this end, we propose the following classification of modules of an FCW (a *module* being an FD with a task and a stop).

An *invalid module* is a module for which, in at least one path of the workflow, the stop precedes the task. This would mean that the configuration of the module had to be finished before the person responsible for it can even get a chance to configure it. In the example of Figure 1, if the stop of fd_2 were ⓤ, then this would be an invalid module.

A *closed module* is a module where task and stop immediately follow each other (fd_2 in the example). This corresponds to the original definition of a module in MLSC: the configuration of an FD has to be done in 'one shot'.

All other modules, such as fd_1 in Figure 1, are called *open*. In their case, certain configuration decisions might be postponed. Having an open module, however, only makes sense if between its task and stop, there are tasks of other modules that can configure it through inter-module links. Modules for which this is indeed the case are called *justified*. Concretely, fd_1 is justified open because its feature d is equal to a feature of a later module.

Now consider a workflow with an invalid module. In fact, it might still have a valid execution: if it has no variability to begin with (e.g. an FD with only *and*-decompositions) or in case the module is fully configured through inter-module links by the modules that appear before its stop is reached. Furthermore, workflows with non-justified open modules also have valid executions: those where the configuration of each module finishes in its assigned task, ahead of the stop. Therefore, invalid modules as well as open modules that are not justified can be considered modelling errors, and can be corrected by turning the FCW into a *normal form*. The normal form of an FCW is an FCW with the same executions but with closed and justified open modules only. An editor for FCWs would have to offer ways to check whether an FCW is in normal form.

These concepts can all be formally defined for the FCW syntax and can automatically be checked either by an analysis of the underlying workflow (which is, in fact, a Petri net [8]) or through approximation by syntactical checks. Armed with such verifications, a workflow can be transformed into normal form before being executed, and the execution engine can be fed with precalculated information.

3 Runtime prevention

Even with an FCW in normal form, a configuration tool has to perform additional analyses in order to prevent the process from deadlocking. Indeed, if a module is justified open, this means that certain decisions can be postponed, but the configuration tool still has to determine which ones. If a stakeholder was allowed to proceed without having made a decision she had to, once the stop of her module is reached, the process would deadlock and would have to be rolled back to the point where the decision was forgotten. Put more generally, the problem is to determine in each task which are the choices to be made *at least*.

This can be done by analysing the boolean formula encoding of the FD d, Γ_d, in order to determine for each feature whether it is *propositionally defined* [5] in terms of the subsequent features. Concretely, if a feature f is defined by an expression over f_a, f_b and f_c, then if f_a, f_b and f_c are features that belong to modules that occur on all subsequent paths, the choice of f can be left open. In addition, if f_a, f_b and f_c only appear on *some* subsequent paths, but a mechanism to force workflow execution into taking these paths can be used, the choice of f can also be postponed. This process is recursive, since f_b might itself be defined in terms of features in subsequent modules, and runs until the choices of f_a, f_b and f_c are fixed, inducing the choice for f. Definability can be checked with a single UNSAT [5].

4 Conclusion

We recalled FCW, a notation previously defined in [3], and discussed how deadlocks might arise during execution of an FCW—something that has to be avoided in practice. The contribution of this paper is a proposal of how this can be achieved, namely by (i) analysing the FCW statically, requiring it to be in normal form, and (ii) performing checks at runtime for decisions that are about to be postponed. An implementation of this is underway.

Acknowledgements. Sponsored by the Interuniversity Attraction Poles Programme of the Belgian State, Belgian Science Policy (MoVES project) and the FNRS.

References

[1] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Staged configuration through specialization and multi-level configuration of feature models. *Software Process: Improvement and Practice*, 10(2):143–169, 2005.

[2] S. Deelstra, M. Sinnema, and J. Bosch. Product derivation in software product families: a case study. *J. Syst. Softw.*, 74(2):173–194, 2005.

[3] A. Hubaux, A. Classen, and P. Heymans. Formal modelling of feature configuration workflows (to appear). In *Proceedings of SPLC'09*, 2009.

[4] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, SEI, Carnegie Mellon University, November 1990.

[5] J. Lang and P. Marquis. On propositional definability. *Artificial Intelligence*, 172(8-9):991–1017, 2008.

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

[7] 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, September 2006.

[8] W. van der Aalst and A. ter Hofstede. Yawl: yet another workflow language. *Information Systems*, 30(4):245–275, 2005.