Electronic Communications of the EASST
Volume 47 (2012)

Proceedings of the 11th International Workshop on Graph Transformation and Visual Modeling Techniques (GTVMT 2012)

Optimizing Model-Based Software Product Line Testing with Graph Transformations

Anthony Anjorin, Sebastian Oster, Ivan Zorcic and Andy Schürr

14 pages

Guest Editors: Andrew Fish, Leen Lambers
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Optimizing Model-Based Software Product Line Testing with Graph Transformations

Anthony Anjorin*, Sebastian Oster, Ivan Zorcic and Andy Schürr

anjorin, oster, zorcic, schuerr@es.tu-darmstadt.de
Real-Time Systems Lab,
Technische Universität Darmstadt, Germany

Abstract:
Software Product Lines (SPLs) are increasing in relevance and importance as various domains strive to cope with the challenges of supporting a high degree of variability in modern software systems. Especially the systematic testing of SPLs is non-trivial as a high degree of variability implies a vast number of possible products. As testing every valid product individually quickly becomes infeasible, heuristics are often used to choose a representative subset of products to be tested. MoSo-PoLiTe (Model-Based Software Product Line Testing) is a framework for SPL testing that combines and applies combinatorial (in particular pairwise) and model-based testing to SPL feature models.

In this paper, we (1) present MoSo-PoLiTe as a novel case study for graph transformations in general and Story Driven Modelling (SDM) in particular, (2) show why we consider SDMs to be ideal for rapid prototyping optimization strategies in this context, and (3) evaluate our implemented optimizations and quantify the realized improvements for MoSo-PoLiTe.

Keywords: Software Product Line, Model-Driven Testing, Graph Transformations, Story-Driven Modelling, Constraint Satisfaction Problem

1 Introduction

A Software Product Line (SPL) architecture provides a systematic means of producing different applications from a common architecture family, reusing common *features* (units of functionality) in the process [PBL05]. The SPL paradigm, already applied successfully in various domains, promises increased software quality, reduced development and maintenance costs, and a decreased time-to-market [CN01].

Developing a feasible strategy for testing SPLs, which aim at reusing the same software components in very different combinations and contexts, poses quite a challenge [McG01]. Classical approaches can only be applied for testing individual products and this becomes quickly infeasible with respect to time and cost even for a moderate degree of variability [Eng10].

An established strategy is to determine a representative subset of products which are tested in lieu of the complete product line. Determining an *optimal* subset of products with respect

* Supported by the 'Excellence Initiative' of the German Federal and State Governments and the Graduate School of Computational Engineering at TU Darmstadt.

to a chosen test metric is, however, NP-hard as it can be mapped to the minimum cardinality hitting set problem [Sch07]. Many approaches, thus, use some heuristics to guide the choice [Sch07, POS+11, KBK11].

MoSo-PoLiTe (Model-Based Software Product Line Testing) [OMR10] is a framework for SPL testing that combines and applies combinatorial testing and model-based testing to SPL feature models [KCH+90]. Experience from various industrial cooperations shows that especially *pairwise* testing results in a reasonable subset of products, which can be feasibly tested. Based on the results of applying MoSo-PoLiTe for real-world use-cases, the basic assumption that most errors can be found by testing all valid pairwise combinations of features appears to hold in practice [POS+11]. In the MoSo-PoLiTe approach, a feature model that describes the variability in an SPL as a tree of interrelated features [KCH+90], is converted to a Constraint Satisfaction Problem (CSP) via a series of transformation rules that flatten the feature tree appropriately. This flattening transformation has been proven to be semantic preserving [Ost11]. The task of determining a subset of product configurations, which covers all valid pairwise combinations of features, is thereby reduced to solving the CSP using well known approaches such as *forward checking* with some extensions. Details concerning how the chosen subset of products to be tested can be mapped to concrete test cases using a test model are discussed in [OZML11].

The flattening transformation that converts the feature model to a CSP is by no means unique and can be varied and optimized for a concrete CSP solver. In this paper, we concentrate on the optimization of this flattening transformation using graph transformations. An optimization strategy is, for example, to create redundant constraints that do not change the semantics of the CSP but lead to a reduction of the search space for valid combinations. This basically results in a trade-off of memory (for redundant constraints and annotations in the flattened tree) for efficiency (reducing the search space and preventing *backtracking*).

Our contribution is to (1) present a novel application of graph transformations in general and Story Driven Modelling (SDM) in particular in the domain of SPL testing, (2) show that SDMs are well suited for rapid prototyping the transformation and trying out various optimization strategies, (3) measure and evaluate our implemented optimization strategies for the MoSo-PoLiTe SPL testing framework.

The paper is structured as follows: In Sec. 2 we introduce a concrete running example and define the necessary concepts used in the rest of the paper. Section 3 discusses the transformation rules that flatten a feature model to a CSP and explains how the chosen CSP solver works. In Sec. 4 various optimization strategies are presented and we show how these ideas can be translated almost 1-to-1 in concise graph transformation rules. Section 5 presents our optimization results, Sec. 6 discusses related approaches, and Sec. 7 concludes the paper.

2 Fundamentals

A feature f represents a system property that is relevant to some stake holder [CHE05]. Given the set of all features $F = \{f_1, f_2, \ldots, f_n\}$, a Product Configuration $PC \in \mathscr{P}(F)$ is a combination of features that constitute the corresponding product. A Feature Model $FM(F) \subseteq \mathscr{P}(F)$ restricts feature combinations to valid product configurations $PC \in FM(F)$ thus defining the variability in an SPL [KCH+90]. We employ a FODA-like FM [KCH+90] with *mandatory*, *optional*, *or*,

and *alternative* features as well as binary *excludes* and *requires* cross-tree dependencies. Our running example is a sample SPL from the automotive domain, a Body Comfort System (BCS) [LOGS11]. Figure 1 depicts the feature model for the BCS SPL (left) and a valid product configuration (right) in concrete syntax.

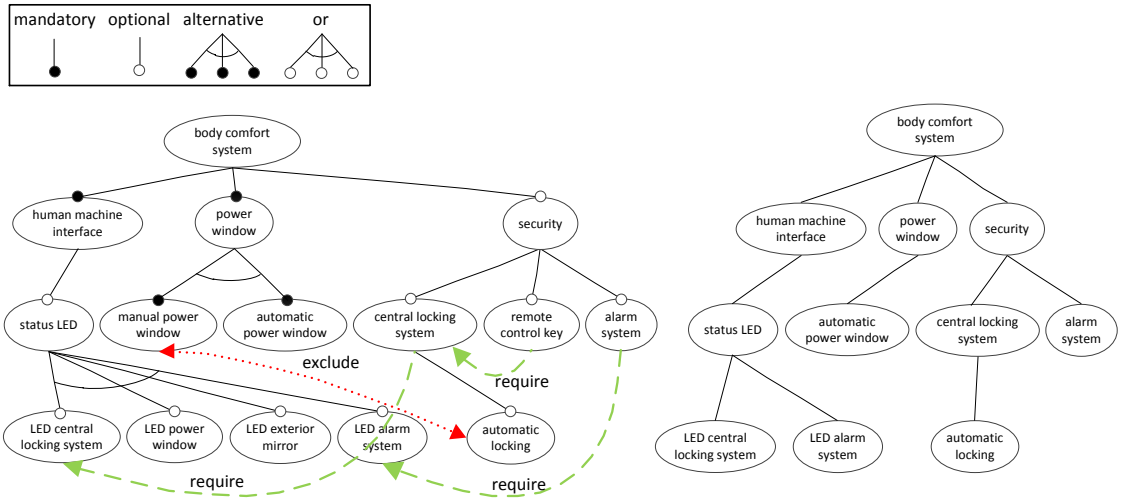

Figure 1: A feature model and a valid product configuration in concrete syntax.

A *body comfort system* consists of the mandatory features *human machine interface* and *power window* and an optional feature *security*. The feature model also defines cross-tree dependencies, e.g, the *remote control key* feature requires a *central locking system* and a *manual power window* excludes the *automatic locking* feature. A feature model can be transformed to a set of conditions in propositional logic over the features [CW07]. A valid product configuration is a set of features that fulfills all conditions, or, in terms of the concrete syntax used in Fig. 1, a valid subtree that satisfies all cross-tree dependencies.

As our graph transformation language *Story Driven Modelling* (SDM) [F+00] operates on typed graphs (models) in *abstract syntax*, we need to represent feature models as graphs typed according to a type graph (*metamodel*). In the following and in the rest of the paper, we shall use the terms *model* and *metamodel* instead of *typed graph* and *type graph*, respectively. Our tool *eMoflon*[1] supports Ecore/EMF [ALPS11] and Fig. 2 depicts our *metamodel* for feature models which is of course neither unique nor optimal, but has proven to be suitable for our needs.

Testing every valid product configuration of the BCS SPL individually is already quite challenging as this would involve testing 152 products. For real-world SPLs that are typically larger[2], individual product configuration testing is no longer feasible. The solution provided by the MoSo-PoLiTe framework is to determine a representative subset of valid product configurations $PC_{UT} \subseteq FM(F)$[3] according to a combinatorial criterion on $FM(F)$ [OMR10]. For our running

[1] www.moflon.org
[2] Up to 270 features according to www.splot-research.org
[3] UT stands for "Under Test"

example we could, for example, require that PC_{UT} comprise all *valid pairwise* combinations of features. For a small subset of the BCS feature model consisting of the three features security, central locking system and remote control key, all valid pairwise combinations would be: (security, central locking system), (security, ¬central locking system), (¬security, ¬central locking system), (security, remote control key), (security, ¬remote control key), (¬security, ¬remote control key), (central locking system, remote control key), (central locking system, ¬remote control key) and (¬central locking system, ¬remote control key), where ¬f means the feature f is not part of the product. All invalid pairwise combinations would be: (¬security, central locking system), (¬security, remote control key) and (¬central locking system, remote control key).

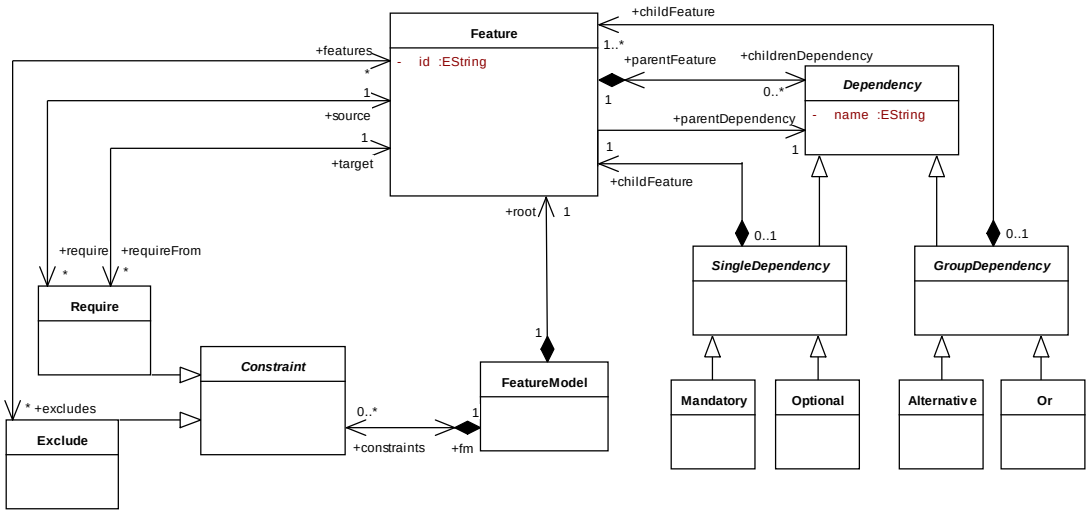

Figure 2: A metamodel for feature models

Using graph transformations to implement the necessary steps is advantageous as feature models typically have numerous cross-tree dependencies and are, thus, complex graph-like structures. We argue and show with our case-study that graph transformations provide a declarative, high-level means of manipulating such structures and increase not only productivity (via rapid prototyping) but also readability and hence maintainability of the system.

3 MoSo-PoLiTe

The MoSo-PoLiTe framework for SPL testing combines combinatorial testing with model-based testing methods and applies this to SPL feature models [OMR10]. In the following, we concentrate on the subtask of determining a representative subset of valid product configurations according to a combinatorial criterion (e.g. pairwise). For a complete overview of MoSo-PoLiTe, especially of the aspects related to model-based testing, we refer to [OMR10, OZML11]. Figure 3 gives a schematic representation of the most important steps involved in the subset selection process.

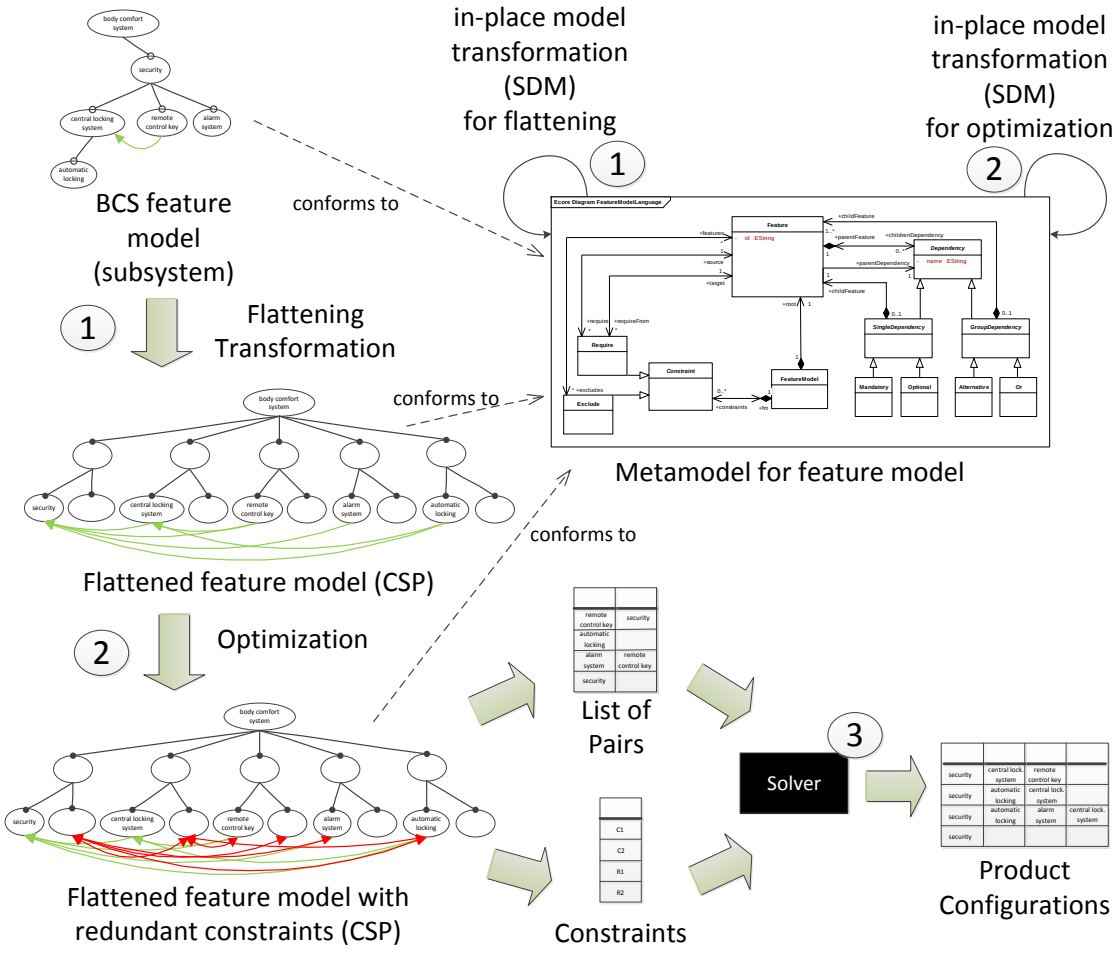

Figure 3: The MoSo-PoLiTe Framework

(1) Flattening Transformation: The BCS feature model is transformed in a first step to a Constraint Satisfaction Problem ($CSP = (P, V, \mathscr{C})$), which consists of a set P of parameters, a set V of values, and a set $\mathscr{C}$ of constraints. Constraints $c \in \mathscr{C}$ are of the form $< p, R >$ where $p = (p_1, p_2, \ldots, p_n)$ is an n-tuple of parameters and R is an n-ary relation on V. A valid product configuration PC is defined by a mapping $v : P \rightarrow V$ that fulfills all constraints (i.e., for $c =< (p_1, p_2, \ldots, p_n), R >$, $(v(p_1), v(p_2), \ldots, v(p_n)) \in R$ holds).

As SDMs work *in-place*[4], they are especially suitable for endogenous[5] model transformations and we exploit this by interpreting a flattened form of the feature model as a CSP. Via a series of transformations rules, a feature model can be flattened (Fig. 3::1[6]) until it consists of a root and two layers: the first layer represents the parameters of the CSP, while the leaves are the possible

[4] The input model is transformed destructively into the output model.
[5] Transformation between instances of the same metamodel.
[6] We use Fig. n::m to refer to the annotation m in Figure n.

values of the corresponding parameters. The constraints of the CSP correspond to the cross-tree dependencies of the flattened feature model. Resulting parameters and values are shown for a small subset of the BCS feature model in Figure 4.

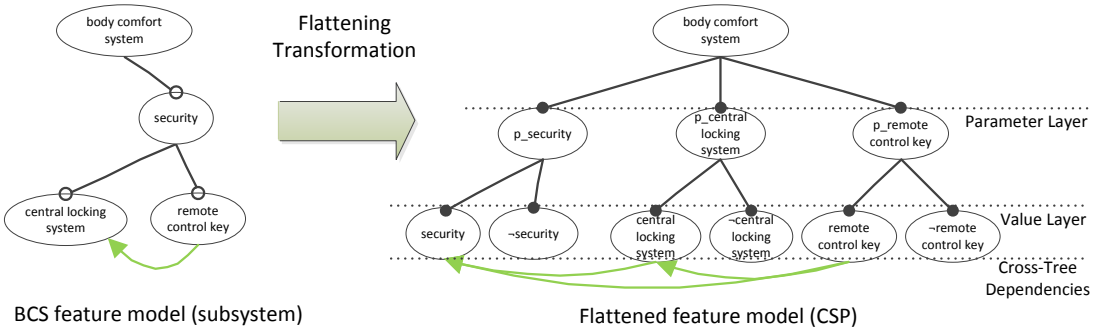

Figure 4: Parameter and Value Layer for a BCS-Subset

Figure 5 depicts an excerpt of one of the 16 rules used for flattening. In the SDM notation, the left-hand side and right-hand side of a transformation rule $r = (L, R)$ are merged together in a single specification. When the rule is applied to a match in a model, the elements in $R \setminus L$ and $L \setminus R$ are created (denoted by the stereotype `create`) and deleted (`destroy`), respectively. All other elements ($L \cap R$) are retained and do not have a stereotype[7]. SDMs also have an imperative part for specifying basic control flow, i.e., the order in which rules should be applied. This is however synonymous to an action language with the usual imperative concepts (if/else, forEach) and we shall focus more on the declarative model transformation rules.

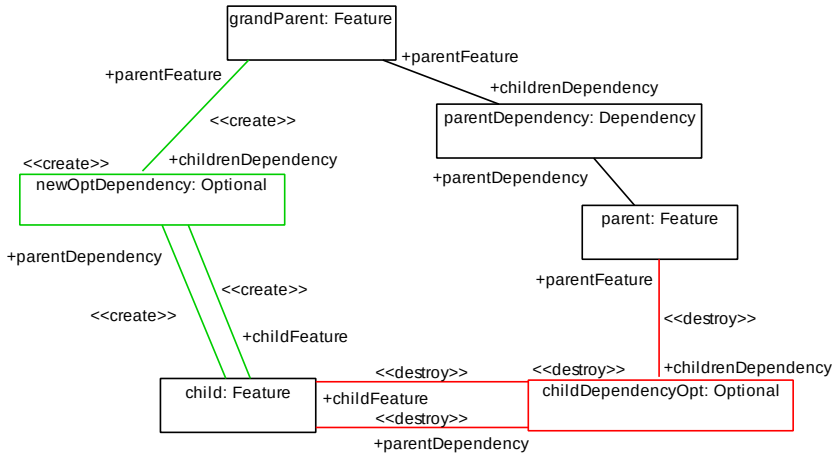

Figure 5: A transformation rule used for flattening the feature model.

[7] This is also depicted using colours: black (retain), green (create), red (delete).

In our opinion, the SDM specification consisting of flattening rules such as Fig. 5 is quite clear and concise, and is ideal for implementing the required transformation. The rule depicted in Fig. 5 matches a feature `parent` that has an optional child-feature `child` and flattens this part of the feature tree by deleting the `parent-child` connection and *pulling up* `child` by creating an optional dependency to `grandParent`, the parent feature of `parent`.

We refer to [Ost11] for a complete list of rules and the proof that the flattening transformation is indeed semantic preserving (i.e., the equivalent of the feature model in first order logic is retained).

(2) Optimization: In an optimization step (Fig. 3::2), the flattened feature model can be *annotated* with extra information to improve the performance of the CSP solver. In our case, optimization rules also implemented with SDMs (discussed in detail in Sec. 4) basically add redundant constraints to the CSP that help the solver to avoid getting caught in a deadlock (partial configuration that cannot be completed to become a valid configuration) and having to backtrack.

(3) Solver: In a final step, the optimized CSP is *solved*, i.e., product configurations that fulfill all constraints are determined until all valid pairwise combinations of features have been covered. For the scope of this paper, the solver is treated as a well-tested black-box that should not be changed (Fig. 3::3).

The CSP solver applies the following strategy: (i) a list of all pairwise combinations of features are derived from the flattened feature model, (ii) an initial pair is chosen according to a greedy component[8] and the solver attempts to extend it to a product configuration that fulfills all constraints in the CSP, backtracking if necessary. If this fails then the initial pair was invalid and is removed from the list of pairs to be covered. If this was successful, all other pairs that are contained in the determined product configuration are marked as covered, (iii) this is repeated until all pairs are covered.

The basic backtracking approach in step (ii) is complemented with a *forward check*[9] that reduces the search space whenever a new parameter value is chosen by removing all parameter values that contradict the current choice. As we shall present in the following section, this already quite effective forward checking technique can be further augmented by adding redundant constraints (cross-tree dependencies) to the CSP.

4 Optimization Strategies

The basic idea of our optimization strategies is to annotate the flattened feature tree (the CSP) by adding redundant constraints. The goal of this optimization is to (1) help filter out pairs of features that are invalid, and (2) further improve the forward check of the solver so that wrong decisions and consequent backtracking can be avoided. The following subsections present three optimization strategies, implemented as graph transformations, with short proofs that the semantic preservation of the entire flattening transformation is retained:

(1) Transitive Closure for Requirements: The first and most intuitive rule involves building the transitive closure of all requirement constraints. As depicted in Fig. 6, a chain of two requirements induces a direct transitive requirement, which is created if it doesn't exist already. These

[8] Based on weights determined by the frequency of pairs.

[9] We use the term *forward check* to indicate a look-ahead of one step.

redundant requirement constraints simplify other rules, which can now assume the transitive closure, i.e., $\forall val1, val2, val3 : (val1 \text{ requires } val2) \wedge (val2 \text{ requires } val3) \Rightarrow (val1 \text{ requires } val3)$.

To show semantic preservation, the corresponding subtree with the extra transitive requirement can be transformed into a set of expressions in first-order logic, which can be reformulated until it is equivalent to the set of expressions of the subtree *without* the extra requirement. This is trivial for transitive require constraints.

(2) Derived Excludes: The second optimization rule concerns deriving extra exclude constraints. As depicted in Fig. 7, if a certain value val_q of a parameter q requires a value val_{p1} of a further parameter p, an exclude constraint between val_q and another value val_{p2} of p can be derived if $p \neq q \wedge val_q \neq val_{p1} \neq val_{p2}$ and the exclude constraint does not already exist, i.e. the parameter p can only be assigned one of its possible values, and val_q restrains the choice to val_{p1} excluding all other options for p.

Please note that our transformation engine implicitly enforces *injective* matches, i.e., no two object variables in the pattern can be mapped to the same model element. This means we do not have to explicitly demand $p \neq q \wedge val_q \neq val_{p1} \neq val_{p2}$ in the pattern. Also note how NACs (Negative Application Conditions) are depicted as *negative* elements that are cancelled such as `noRequire` in Fig. 6 and `noExclude` in Fig. 7.

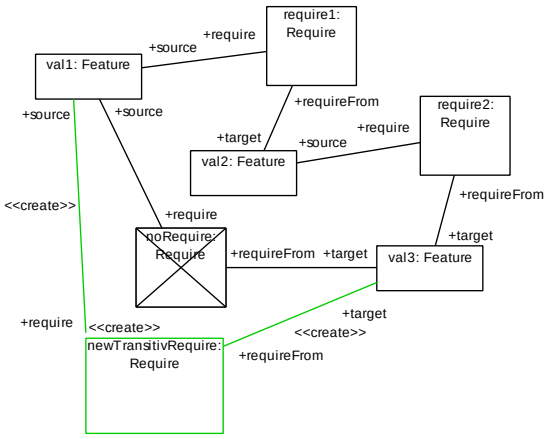

Figure 6: Transitive requires

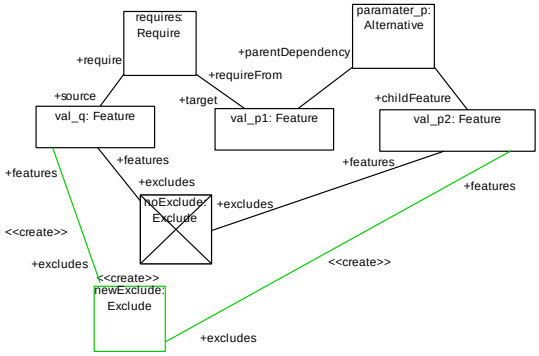

Figure 7: Derived excludes

As in the case of building the transitive closure for requirements, deriving excludes according to Fig. 7 is also semantic preserving ($\oplus$ represents an exclusive or):

$$(val_q \ \ requires \ \ val_{p1}) \wedge (val_q \ \ excludes \ \ val_{p2}) \wedge (val_{p1} \oplus val_{p2}) \quad (1)$$

$$\Leftrightarrow (val_q \Rightarrow val_{p1}) \wedge (\neg val_q \vee \neg val_{p2}) \wedge ((val_{p1} \wedge \neg val_{p2}) \vee (\neg val_{p1} \wedge val_{p2})) \quad (2)$$

$$\Leftrightarrow (\neg val_q \vee val_{p1}) \wedge (\neg val_q \vee \neg val_{p2}) \wedge (val_{p1} \vee val_{p2}) \wedge (\neg val_{p1} \vee \neg val_{p2}) \quad (3)$$

$$\Leftrightarrow (\neg val_q \wedge \neg val_{p1} \wedge val_{p2}) \vee (\neg val_q \wedge val_{p1} \wedge \neg val_{p2}) \vee (val_{p1} \wedge \neg val_{p2}) \quad (4)$$

$$\Leftrightarrow (\neg val_q \vee val_{p1}) \ \wedge \ (\neg val_{p1} \vee \neg val_{p2}) \wedge (val_{p1} \vee val_{p2}) \quad (5)$$

$$\Leftrightarrow (val_q \ \ requires \ \ val_{p1}) \ \wedge \ (val_{p1} \oplus val_{p2}) \quad (6)$$

(3) Propagated Excludes: A further rule depicted in Fig. 8 also adds redundant excludes. If a value val_1 requires another value val_2 which excludes a further value val_3, then val_1 also excludes val_3. This rule is also semantic preserving as shown in the following:

$$(val_1 \ \ requires \ \ val_2) \wedge (val_2 \ \ excludes \ \ val_3) \wedge (val_1 \ \ excludes \ \ val_3) \quad (7)$$
$$\Leftrightarrow (\neg val_1 \vee val_2) \wedge (\neg val_2 \vee \neg val_3) \wedge (\neg val_1 \vee \neg val_3) \quad (8)$$
$$\Leftrightarrow (\neg val_1 \wedge \neg val_2) \vee (\neg val_1 \wedge \neg val_3) \vee (val_2 \wedge \neg val_3) \quad (9)$$
$$\Leftrightarrow (\neg val_1 \vee val_2) \wedge (\neg val_2 \vee \neg val_3) \quad (10)$$
$$\Leftrightarrow (val_1 \ \ requires \ \ val_2) \wedge (val_2 \ \ excludes \ \ val_3) \quad (11)$$

After deriving the initial CSP via the flattening transformation, the rules (1), (2) and (3) are applied repeatedly in sequence until the total number of constraints of the CSP no longer increases.

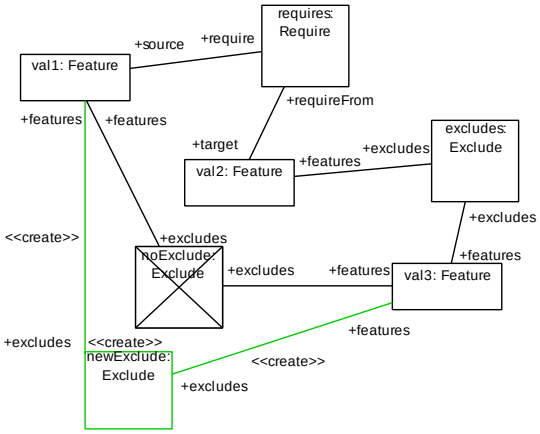

Figure 8: Transformation for propagating exclude constraints.

5 Evaluation

To evaluate our optimization, a series of measurements were performed on a standard PC with a 2.93 GHz Intel Core2 Duo CPU and 4 GB RAM. The underlying operating system and virtual machine was Windows XP Professional SP3 and Java 1.7, respectively, and the amount of time the CPU spent performing actions for the tests, i.e., *user time*, was measured. For all tests, a feature model generator was implemented that can generate random feature models of a specified size and with a given number of cross-tree dependencies. Figure 9 presents our results[10] in 4 plots, which are explained in detail in the following.

Each test was repeated 10 times and measurements were averaged to compensate for fluctuations and improve accuracy. Furthermore, 10 different sets of randomly generated cross-tree

[10] A workspace with our implementation and all tests can be downloaded from www.moflon.org.

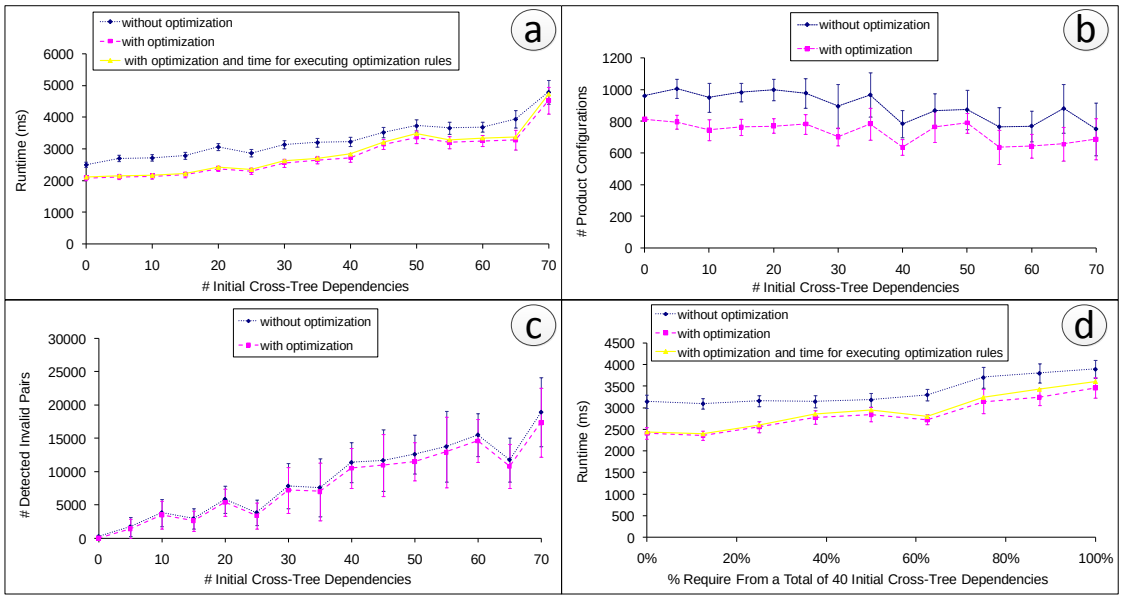

Figure 9: Effects of optimization strategies.

dependencies were used. All results are depicted with a 95% confidence interval in each data point showing that concrete values may vary, but our results are not dependent on a specific choice of constraints.

Although we were able to verify our results for feature models consisting of up to about 700 features with 10% initial cross-tree dependencies, we chose to use a feature model with 319 features to present our evaluation. This ensures that results can be compared to real feature models available from www.splot-research.org, which range in size from about 9 to 290. With feature models consisting of thousands of features our optimization becomes a bottleneck in some cases and an appropriate strategy to restrain the rules as required is left to future work.

Plot (a): The runtime (y-axis) of the whole process is shown for a varying number of initial cross-tree dependencies (x-axis) ranging from 0 to 70. The performance of the process without our optimization (blue dotted line) is compared with the performance on the optimized CSP (magenta dashed line) and with the total time (yellow solid line), which includes the time for executing the optimization rules as well. Please note that 0 initial cross-tree dependencies means that the feature model was used without *initial* cross-tree dependencies. As the flattening transformation, however, always introduces extra dependencies to preserve semantics, the resulting CSP, even for 0 initial cross-tree dependencies, still contains a considerable number of constraints derived from the tree structure of the feature model and from the flattening process. This explains why there is a fairly constant 15% reduction in runtime across the x-axis.

As expected, the runtime for all cases increases steadily with the number of initial cross-tree dependencies. These results show that our optimizations indeed improve performance and that the actual time required to execute the optimization rules is negligible compared to the improvement.

Plot (b): For the same configuration, the sets of generated product configurations are plotted (y-axis) for the same x-axis as in Plot (a). Our optimization clearly leads to a 15–20% reduction of the generated sets. This result is even more important than a reduction in runtime, as this is the actual goal of the whole process.

Plot (c): The reason for the reduction in size of the generated sets of product configurations lies in the improved detection of invalid pairs *before* the CSP is solved. We obtain best results by filtering as many invalid pairs as possible using *simple* heuristics, which become more effective with the redundant constraints introduced by the optimization. The number of invalid pairs that could *not* be filtered and are later detected by the solver is plotted for the same x-axis as in Plot (a) and Plot (b) showing a 10% reduction. Less invalid pairs improves the greedy component[11] of the CSP solver as the used weights become more accurate, leading to a reduction in size of the set of product configurations (Plot (b)).

Plot (d): Using the same feature model but with a constant number of 40 initial cross-tree dependencies, runtime (y-axis) is plotted for a varying ratio of *require* constraints to *exclude* constraints (x-axis). As two of our three optimization rules involve exclude constraints, the reduction in runtime is higher for a larger number of exclude constraints (0% initial cross-tree require dependencies) and reduces slightly as more exclude are replaced with require constraints (100% initial cross-tree require dependencies). This is confirmed by our measurement results.

6 Related Work

We classify related work into two groups: alternative approaches to SPL testing in general, and approaches that are similar to ours, but differ in how feature models are transformed to a CSP.

Alternative approaches to SPL testing: Approaches to SPL testing can be further categorized into three subgroups: *contra-SPL philosophy*, *reuse techniques*, and *subset heuristics*. Contra-SPL philosophy approaches ignore the inherent reuse in SPLs and are only appropriate for small SPLs [Sch07]. Reuse techniques aim to reduce the testing effort for SPLs by reusing test artifacts (e.g., test cases and data) across products of an SPL. This is achieved by incrementally testing products via regression testing techniques or by appropriately adapting domain tests during product testing. Incremental SPL testing, as introduced in [Eng10], faces the challenges of identifying a *suitable* product to start the process with and identifying *what* needs to be re-tested. Approaches that adapt domain tests often employ model-based testing techniques for SPLs as models can provide suitable mechanisms to describe variable test-relevant behaviour. We refer to [OWES10] for an extensive comparison of model-based testing techniques. Approaches that use subset heuristics aim at generating a representative subset of products according to a coverage criterium such as combinatorial testing [OMR10, POS+11], or requirements coverage [Sch07]. Although SAT solvers can also be used to solve the basic problem of identifying a subset of products that satisfies a set of combinatorial constraints on features [POS+11], feature models mainly employ *binary* constraints over parameters with *non-binary* domains for which CSP solvers seem to be a natural choice [Ben04]. We have furthermore compared our concrete implementation with a SAT approach and our solution proved to be superior in terms of efficiency and the size of determined subsets [POS+11].

[11] This determines which initial pair of features is used to start every new product configuration.

Alternative feature model to CSP transformation: The authors of [WDS09] perform a cartesian flattening to transform feature models into a *knappsack* problem which is solved and used to generate representative subsets of products of the SPL. A major difference to our flattening approach is that they choose to prohibit an exponential explosion of all possible feature combinations via the transformation but lose semantic equivalence in the process. As an example, cardinality/or groups are translated into XOR/alternative groups of a *bounded* size [WDS09]. We prefer to retain semantic equivalence between the original and the flattened feature model and control the number of combinations via a coverage criterium that can range from pairwise to n-wise as required. The flattening transformation is, in general, not unique and we also differ from [WDS09] in the treatment of *alternative* groups beneath an *alternative* parent as depicted in Fig. 10 for an abstract example taken from [WDS09].

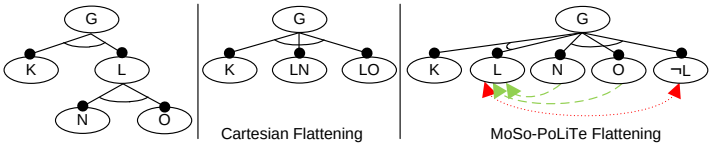

Figure 10: Flattening approaches for an *alternative* parent with *alternative* children [POS+11]

In the cartesian flattening approach, the features **N** and **O** are merged with its parent feature **L**. If, however, another feature **X** requires **L**, this binary constraint must be translated into **X** requires (**L,N** *xor* **L,O**) which is a *non-binary* constraint. For this reason we use a different flattening strategy as depicted in Fig. 10 that results in only binary constraints, which are handled better by our subset extraction algorithm.

7 Conclusion

In this paper we presented a novel case study for graph transformations in the domain of SPL testing. We have successfully employed SDMs not only for the involved flattening transformation that transforms a feature model into a CSP, but also for an optimization process that adds redundant constraints to improve the performance of our unchanged CSP solver. Our results show that the optimization leads to a reduction in runtime and to an increase in quality (reduction in size) of the generated sets of product configurations for pairwise testing. This novel case study for the graph transformation community showcases the advantages of using graph transformations including improved readability and maintainability of the transformation and optimization rules, and the possibility of rapid prototyping further optimization rules.

As future work we plan to re-engineer the CSP solver using graph transformations. This should not only improve the maintainability of the complete system but should also support the investigation of further optimization rules that involve the current product configuration and must be employed *during* the CSP resolution process by the solver. This would effectively extend the current forward check to a larger look-ahead. It is, however, unclear if this will further improve performance. A performance comparison with the current hand-written Java implementation is also quite interesting and should indicate the tradeoff in efficiency, if any, of using graph transformations.

Last but not least, we shall continue scalability measurements on larger feature models. The cost of performing the optimization rules increases for large feature models (thousands of features) and leads to a bottleneck as a very large number of constraints is obtained in some cases. Depending on the number of features, an appropriate handling must be implemented to decide how to restrain the optimization rules.

Bibliography

[ALPS11] A. Anjorin, M. Lauder, S. Patzina, A. Schürr. eMoflon: Leveraging EMF and Professional CASE Tools. In *3. Workshop Methodische Entwicklung von Modellierungswerkzeugen (MEMWe2011)*. LNI. GI, Bonn, 2011.

[Ben04] H. Bennaceur. A Comparison between SAT and CSP Techniques. *Constraints* 9(2):123–138, 2004.

[CHE05] K. Czarnecki, S. Helsen, U. Eisenecker. Staged Configuration Through Specialization and Multilevel Configuration of Feature Models. *Software Process: Improvement and Practice* 10(2):143–169, 2005.

[CN01] P. Clements, L. Northrop. *Software product lines: practices and patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[CW07] K. Czarnecki, A. Wasowski. Feature Diagrams and Logics: There and Back Again. In *Proc. of the 11th Int. Conf. on Software Product Lines*. Pp. 23–34. IEEE Computer Society, 2007.

[Eng10] E. Engström. Regression Test Selection and Product Line System Testing. In *Proc. of the Third Int. Conf. on Software Testing, Verification and Validation*. Pp. 512–515. IEEE Computer Society, Washington, DC, USA, 2010.

[F+00] T. Fischer et al. Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. In *TAGT '98 Selected Papers*. LNCS 1764, pp. 296–309. Springer, 2000.

[KBK11] C. H. P. Kim, D. S. Batory, S. Khurshid. Reducing Combinatorics in Testing Product Lines. In *Proc. of the 10th Int. Conf. on Aspect-Oriented Software Development*. Pp. 57–68. ACM, 2011.

[KCH+90] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Carnegie-Mellon University Software Engineering Institute, 1990.

[LOGS11] M. Lochau, S. Oster, U. Goltz, A. Schürr. Model-based Pairwise Testing for Feature Interaction Coverage in Software Product Line Engineering. *SQJ - Special issue on Quality Engineering for SPLs*, 2011.

[McG01] J. D. McGregor. Testing a Software Product Line. Technical report CMU/SEI-2001-TR-022, 2001.

[OMR10] S. Oster, F. Markert, P. Ritter. Automated Incremental Pairwise Testing of Software Product Lines. In *Proc. of the 14th Int. Software Product Line Conf.* Pp. 196–210. 2010.

[Ost11] S. Oster. A Semantic Preserving Feature Model to CSP Transformation. Technical report 11, Technische Universität Braunschweig, 2011.

[OWES10] S. Oster, A. Wübbeke, G. Engels, A. Schürr. Model-Based Software Product Lines Testing Survey. In Zander et al. (eds.), *Model-based Testing for Embedded Systems*. CRC Press/Taylor&Francis, 2010. to appear.

[OZML11] S. Oster, I. Zorcic, F. Markert, M. Lochau. MoSo-PoLiTe - Tool Support for Pairwise and Model-Based Software Product Line Testing. In Czarnecki and Eisenecker (eds.), *5th Int. Workshop on Variability Modelling of Software-Intensive Systems*. Pp. 79–82. ACM Press, New York, 2011.

[PBL05] K. Pohl, G. Böckle, F. J. v. d. Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., USA, 2005.

[POS+11] G. Perrouin, S. Oster, S. Sen, J. Klein, B. Baudry, Y. Traon. Pairwise Testing for Software Product Lines: Comparison of Two Approaches. *SQJ - Special issue on Quality Engineering for SPLs*, 2011.

[Sch07] K. Scheidemann. Verifying Families of System Configurations. *PhD Thesis*, TU Munich 2007.

[WDS09] J. White, B. Dougherty, D. C. Schmidt. Selecting Highly Optimal Architectural Feature Sets with Filtered Cartesian Flattening. *Journal of Systems and Software* 82(8):1268–1284, 2009.