Measuring Behaviour Interactions between Product-Line Features

Joanne M. Atlee, Uli Fahrenberg, Axel Legay

Measuring Behaviour Interactions between Product-Line Features

Joanne M. Atlee
University of Waterloo
jmatlee@uwaterloo.ca

Uli Fahrenberg
Inria Rennes
ulrich.fahrenberg@inria.fr

Axel Legay
Inria Rennes
axel.legay@inria.fr

***Abstract*—We suggest a method for measuring the degree to which features interact in feature-oriented software development. We argue that our method is practically feasible, easily extendable and useful from a developer's point of view.**

I. Introduction

The behaviour of a software system is often described in terms of its features, where each *feature* is a unit of functionality that adds value to the system. *Feature-oriented software development (FOSD)* is a software-development strategy that is based on feature decomposition and modularity. Features can be separate modules that are developed in isolation, allowing for parallel, incremental, or multi-vendor development of features. Feature orientation is particularly important in *software product lines*, where a family of related products is managed and evolved in terms of its features: a product line comprises a collection of mandatory and optional features, and individual products are derived by selecting among and integrating features from this feature set. A product line can be expressed as a single model, in which feature-specific behaviour is conditional on the presence of the feature in a product.

The downside of FOSD is that, although features are conceptualized, developed, managed, and evolved as separate concerns, they are not truly separate. They can interact with each other, for example by trying to control the same variables or external phenomena, by issuing events that trigger the other feature, or by affecting conditions that suppress the other feature. In general, there is a *behaviour interaction* whenever the behaviour of a feature deviates from its specification due to the presence of other features.

A number of researchers have investigated the automatic detection of feature interactions ([1], [3], [4], [7], [10], [13], etc.). The primary result of such analyses is effectively a *boolean* determination of whether a combination of features interact. The formulation of the result may be different for different tools (e.g., returning the *set* of features that interact with a given feature f; or returning the combinations of features, from a given feature set, that interact). Some techniques may also report a witness execution trace that manifests a detected interaction, to help the developer understand exactly how the features interact – as a first step towards addressing the interaction. But the essence of the analyses is to report simply the presence or absence of interactions.

We are interested in exploring how to measure the *degree* to which features interact. Features may have multiple interactions, where each interaction instance represents work for the developer: specifically, the interaction must be analyzed to determine if it is a problem; if so, then a patch must be designed, implemented, and tested. Thus, a measure of the number of ways in which features in a product line can interact would tell the developer more about the amount of effort needed to integrate features – or suggest that certain combinations of features should be prohibited – than a simple interaction-existence check provides.

We first provide an overview of our models of features, products, and product lines, and how to use bisimulation to detect the presence of feature interactions [15]. We then explore some ideas for computing richer measures that better reflect the degree to which features interact. We also consider how these measurements can be performed efficiently over a model of the product line, by computing metrics for each feature simultaneously and taking advantage of the commonalities among products.

II. Features, Product Lines, and Interactions

A software system is modelled as a *transition system (TS)* [2], which, for simplicity, we consider to be a set of states, and a set of transitions between states that are triggered by actions.

Definition 1 *A* transition system *(TS)* $\mathcal{S} = (S, \Sigma, I, T)$ *consists of a set of states* S*, a set of initial states* $I \subseteq S$*, a set of actions* Σ*, and a set of transitions* $T \subseteq S \times \Sigma \times S$*. We write* $s \xrightarrow{a} s'$ *to indicate that* $(s, a, s') \in T$*.*

We follow [5] and consider a *feature* to be an optional unit of behaviour that is modelled as transitions that are conditional on the presence of the feature. Let N be a set of features. $\mathbb{B}(N)$ denotes the set of Boolean expressions over N.

Definition 2 *A* featured transition system *(FTS)* $\mathcal{F} = (S, \Sigma, I, T, \gamma)$ *consists of a TS* (S, Σ, I, T) *and a mapping* $\gamma : T \to \mathbb{B}(N)$*. For* $s, s' \in S$ *in a FTS and* $p \subseteq N$*, we write* $s \xrightarrow{a}_p s'$ *if* $s \xrightarrow{a} s'$ *and* $p \models \gamma(s, a, s')$*.*

Example 1 *Figure 1 displays a FTS model of an ATM which we will use as running example. The base feature* B*, whose*

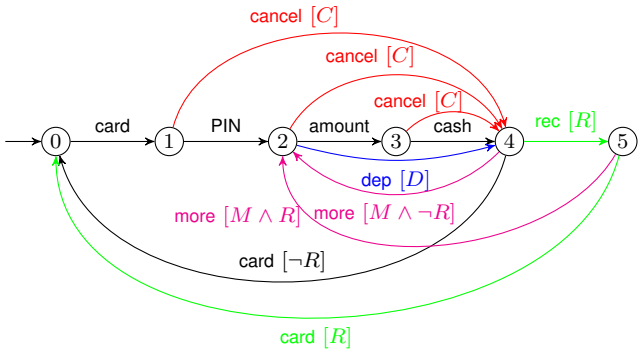

Fig. 1. FTS $\mathcal{F}$ modelling an ATM

presence we have not indicated for sake of brevity, consists of a cycle of card *insertion,* PIN *entrance,* amount *specification,* cash *retrieval, and* card *retrieval. Optional features allow for* cancel*lation, cash* dep*osit,* more *than one transaction, and obtaining a* rec*eipt. This last "R" feature is interesting, as it not only adds new behaviour to the ATM but also disables other behaviour.*

In order to identify a feature interaction, we look for a discrepancy between a feature's behaviour in isolation versus its behaviour in the presence of other features. To do this, we need to be able to refer to a feature's behaviour within a larger product. We use projection over an FTS [5]:

Definition 3 *The* projection *over an FTS* $\mathcal{F} = (S, \Sigma, I, T, \gamma)$ *with respect to a feature expression* $\phi \in \mathbb{B}(N)$ *is the FTS* $\pi^{\phi}(\mathcal{F}) = (S, \Sigma, I, T', \gamma')$*, given by* $\gamma'(t) = \gamma(t) \wedge \phi$ *and* $T' = \{t \in T \mid [\![\gamma(t) \wedge \phi]\!] \neq \emptyset\}$.

If $[\![\phi]\!] = \{p\}$ contains but a single product, we can forget about γ' in the projection (as we will have $[\![\gamma'(t)]\!] = p$ for all $t \in T'$), hence single-product projections, which we also will denote $\pi^p(\mathcal{F})$, can be seen as plain TS.

As shown in [15], discrepancy in behaviours can be detected using bisimulation [12]. Formally, a *behaviour interaction* is a violation of bisimilarity between the behaviours of a feature f in isolation and the behaviours of f when integrated with other (interacting) features. Violation of bisimilarity encompasses a number of specific types of interactions (e.g., conflicting actions, introduced nondeterminism, shared-trigger interactions [11], missed-trigger interactions [11]), thereby enabling a single analysis to detect a wide variety of interactions.

Definition 4 *Given an FTS* $\mathcal{F}$*, a product* $p \subseteq N$*, and a feature* $f \in N$*, we say that* f *has a* behaviour interaction *with* p *if* $\pi^p(\mathcal{F})$ *and* $\pi^p(\pi^{p \wedge f}(\mathcal{F}))$ *are not bisimilar.*

Example 2 *We want to know whether the feature* R *in the FTS of Fig. 1 has an interaction with the base ATM. The projections are depicted in Fig. 2; note how the green transitions are projected away in* $\pi^B(\pi^{B \wedge R}(\mathcal{F}))$*. We hence check whether*

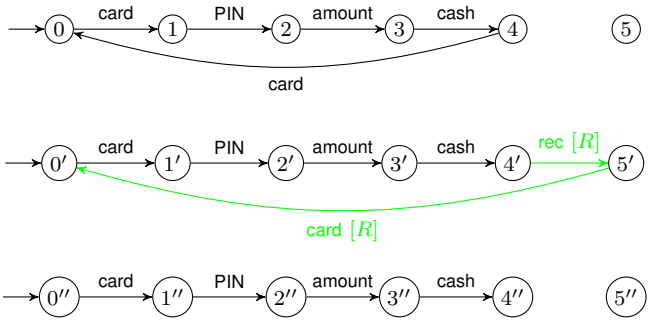

Fig. 2. Projections $\pi^B(\mathcal{F})$, $\pi^{B \wedge R}(\mathcal{F})$ and $\pi^B(\pi^{B \wedge R}(\mathcal{F}))$

$\pi^B(\mathcal{F})$ *and* $\pi^B(\pi^{B \wedge R}(\mathcal{F}))$ *are bisimilar, which of course is not the case, as state* $4''$ *misses the* card *transition of state* 4.

In [6] it is shown that FTS admit a notion of bisimilarity at FTS level, called featured bisimilarity:

Definition 5 *The FTS* $\mathcal{F} = (S, \Sigma, I, T, \gamma)$ *and* $\mathcal{F}' = (S', \Sigma, I', T', \gamma')$ *are* featured bisimilar *with respect to a product* $p \subseteq N$ *if there exists a mapping* $R : S \times S' \to (2^N \to \{\mathbf{ff}, \mathbf{tt}\})$ *such that*

- $\forall i \in I : \exists i' \in I' : R(i, i')(p)$,
- $\forall i' \in I' : \exists i \in I : R(i, i')(p)$,
- *if* $R(s, s')(p)$*, then* $\forall s \xrightarrow{a}_p t : \exists s' \xrightarrow{a}_p t' : R(t, t')(p)$,
- *if* $R(s, s')(p)$*, then* $\forall s' \xrightarrow{a}_p t' : \exists s \xrightarrow{a}_p t : R(t, t')(p)$.

The following lemma, also from [6], then allows to use featured bisimilarity to at once compute all products p with which f has a behaviour interaction. It is shown in [6] that this is about 30 times faster than to compute all single bisimilarities.

Lemma 1 *FTS* $\mathcal{F}$*,* $\mathcal{F}'$ *are featured bisimilar with respect to* $p \subseteq N$ *iff the TS* $\pi^p(\mathcal{F})$ *and* $\pi^p(\mathcal{F}')$ *are bisimilar.*

III. Behavioural Distances

We wish to generalize bisimilarity of TS to a notion which not only tells us *whether or not* two TS are bisimilar, but *how close* they are to being bisimilar. To this end, Algorithm 1 computes the number of unique behaviours, i.e., the number of behaviours which are present in only one of the two TS.

The intuition is that the algorithm tries to match transitions in one TS as good as possible in the other. Hence the function $dist(s, s')$ tries to match every transition $s \xrightarrow{a} t$ in S with a transition $s' \xrightarrow{a} t'$ in S'. If no such exists, a missing behaviour is detected and 1 is added to the score; if there are transitions $s' \xrightarrow{a} t'$, then distance is recursively computed for the pair t, t' with the best match. In the second half of the algorithm, a symmetric match from s' to s is computed. Once a pair s, s' of states has been checked for behaviour mismatches in this way, it is added to a *Passed* list of states which need not be checked again; hence the algorithm finishes after at most $|S| \cdot |S'|$ iterations.

The behavioural distance faithfully extends bisimilarity:

Algorithm 1 Calculates behavioural distance $d(\mathcal{S}, \mathcal{S}')$ between TS $\mathcal{S} = (S, \Sigma, I, T)$ and $\mathcal{S}' = (S', \Sigma, I', T')$

global *Passed* $\leftarrow \emptyset$
return $\max\left\{ \max_{i \in I} \min_{i' \in I'} \mathit{dist}(i, i'), \max_{i' \in I'} \min_{i \in I} \mathit{dist}(i, i') \right\}$

function $\mathit{dist}(s, s')$
    Add (s, s') to *Passed*
    $d \leftarrow \mathit{dista}(s, s') + \mathit{dista}(s', s)$
    return d

function $\mathit{dista}(s, s')$
    local $m \leftarrow \infty$, $d \leftarrow 0$
    for all $s \xrightarrow{a} t$ **do**
        if $s' \not\xrightarrow{a}$ **then** $d \leftarrow d + 1$
        else
            for all $s' \xrightarrow{a} t'$ **do**
                if $(t, t') \notin$ *Passed* **then**
                $m \leftarrow \min(m, \mathit{dist}(t, t'))$
            $d \leftarrow d + m$
    return d

Theorem 1 *TS $\mathcal{S}$, $\mathcal{S}'$ are bisimilar iff $d(\mathcal{S}, \mathcal{S}') = 0$.*

Proof sketch: If $\mathcal{S} = (S, \Sigma, I, T)$ and $\mathcal{S}' = (S', \Sigma, I', T')$ are bisimilar, then there is a bisimulation relation $R \subseteq S \times S$. It can easily be shown that algorithm 1 will follow this relation when computing the distance, so that every time $\mathit{dist}(s, s')$ is called, $(s, s') \in R$. But then all transitions $s \xrightarrow{a} t$ have a match $s' \xrightarrow{a} t'$, and vice versa, with $(t, t') \in R$, so that $\mathit{dist}(s, s') = 0$. For the other direction of the proof, one easily sees that $R \subseteq S \times S'$ defined by $R = \{(s, s') \mid \mathit{dist}(s, s') = 0\}$ is a bisimulation.

IV. Measuring Feature Interactions

We can now use our behavioural distance to measure feature interactions. The following definition of a behaviour interaction score generalizes Definition 4 and allows us to count, algorithmically, the number of behaviour interactions between a feature and a product.

Definition 6 *Given an FTS $\mathcal{F}$, a product $p \subseteq N$, and a feature $f \in N$, the* behaviour interaction score *of f with respect to p is $d(\pi^p(\mathcal{F}), \pi^p(\pi^{p \wedge f}(\mathcal{F}))$.*

Note that by Theorem 1, the behaviour interaction score is 0 iff there is no behaviour interaction.

Example 3 *We have already seen that the feature R has a behaviour interaction with the base ATM. To see how many of these are present, we compute $d(\pi^B(\mathcal{F}), \pi^B(\pi^{B \wedge R}(\mathcal{F})))$. We have $d(0, 0'') = d(1, 1'') = d(2, 2'') = d(3, 3'') = d(4, 4'') = 1$. This fits with the intuition that there is precisely one behaviour missing in $\pi^B(\pi^{B \wedge R}(\mathcal{F}))$ compared to $\pi^B(\mathcal{F})$, i.e., the feature R has one behaviour interaction with the base ATM.*

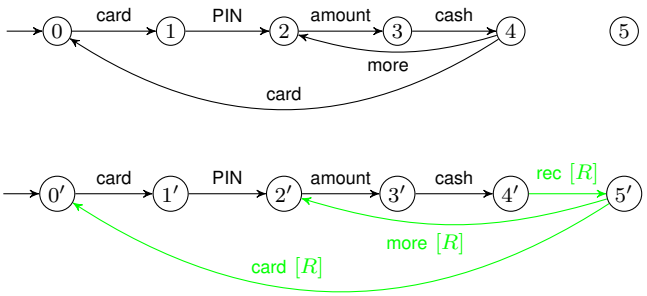

Fig. 3. Projections $\pi^{B \wedge M}(\mathcal{F})$ and $\pi^{B \wedge M \wedge R}(\mathcal{F})$

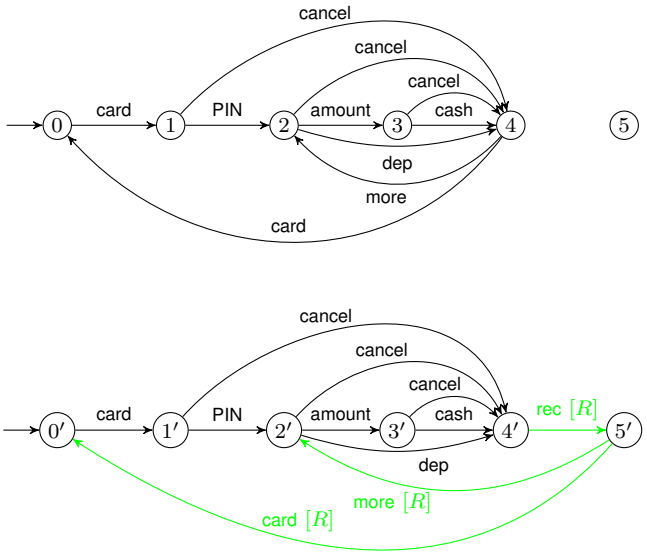

Fig. 4. Projections $\pi^{B \wedge M \wedge C \wedge D}(\mathcal{F})$ and $\pi^{B \wedge M \wedge C \wedge D \wedge R}(\mathcal{F})$

We also want to know how many interactions R has with $B \wedge M$, i.e. $d(\pi^{B \wedge M}(\mathcal{F}), \pi^{B \wedge M}(\pi^{B \wedge M \wedge R}(\mathcal{F})))$. The projections are depicted in Fig. 3. We have $d(0, 0'') = d(1, 1'') = d(2, 2'') = d(3, 3'') = d(4, 4'') = 2$, as expected: $\pi^{B \wedge M}(\pi^{B \wedge M \wedge R}(\mathcal{F}))$ misses two behaviours compared to $\pi^{B \wedge M}(\mathcal{F})$, i.e., the feature R has two behaviour interactions with the product $B \wedge M$.

Lastly, we count the number of interactions between the all-feature ATMs with and without R, see Fig. 4: $d(0, 0'') = d(1, 1'') = d(2, 2'') + d(4, 4'') = d(3, 3'') + d(4, 4'') = d(4, 4'') = 2$.

We want to lift our behaviour interaction score to FTS level, so that we can compute at once the score of f for all possible products. This generalization is provided by algorithm 2. Intuitively, like algorithm 1, algorithm 2 tries to match transitions in one FTS as good as possible in the other, but for all products p. Hence the *Passed* list is now a function of products, and at every iteration, the algorithm loops through all products. Note

Algorithm 2 Calculates featured behavioural distance $fd(\mathcal{F}, \mathcal{F}')$ between FTS $\mathcal{F} = (S, \Sigma, I, T, \gamma)$ and $\mathcal{F}' = (S', \Sigma, I', T', \gamma')$

global *Passed* : $2^N \to$ Set
for all $p \subseteq N$ **do** *Passed*$(p) \leftarrow \emptyset$
return $\max\{\max_{i\in I}\min_{i'\in I'} fdist(i,i'), \max_{i'\in I'}\min_{i\in I} fdist(i,i')\}$

function *fdist*(s, s')
    for all $p \subseteq N$ **do**
        Add (s, s') to *Passed*(p)
        $d \leftarrow fdista(s, s') + fdista(s', s)$
    return d

function *fdista*(p, s, s')
    local $m \leftarrow \infty$, $d \leftarrow 0$
    for all $s \xrightarrow{a}_p t$ **do**
        if $s' \not\xrightarrow{a}_p$ **then** $d \leftarrow d + 1$
        else
            for all $s' \xrightarrow{a}_p t'$ **do**
                if $(t, t') \notin$ *Passed*(p) **then**
                $m \leftarrow \min(m, fdist(t, t'))$
            $d \leftarrow d + m$
    return d

that the algorithm returns a function *fd* which maps products to behavioural distances.

Work in [6] shows that the similar algorithm for computing featured bisimilarity can be implemented efficiently; we believe that similar ideas apply to our algorithm for featured behavioural distance.

Theorem 2 *For all FTS $\mathcal{F}$, $\mathcal{F}'$ and all products $p \subseteq N$, $d(\pi^p(\mathcal{F}), \pi^p(\mathcal{F}')) = fd(\mathcal{F}, \mathcal{F}')(p)$.*

Proof sketch: Similar to the proof of Theorem 11 in [6]: The computations in algorithm 1 can be integrated into a loop over all $p \subseteq N$, which then can be split like in algorithm 2.

V. Vision

We have shown that it is possible to measure the degree to which features interact in feature-oriented software development. Using a simple but realistic example of a featured transition system, we have seen that the measure we have defined concurs with the intuition.

The measure we have introduced here is but one example of a so-called *branching distance* between transition systems [8], [9], and many other such distances may be defined. Precisely which of them are useful in FOSD remains to be seen.

Measuring the degree to which features interact in FOSD will be a useful addition to the developer's tool box, allowing her to determine how much integration of a feature affects the overall product line. Our algorithm can easily be extended to also show precisely *where* the features interact, hence giving visual feed-back to the developer where there may be problems in the model.

Using other and more realistic examples, also expressed with a richer modelling language, we intend to further gauge the usefulness of our behaviour interaction score and other possible measures for feature interaction. For this, it will be useful to devise an efficient implementation of our algorithms.

We note that the algorithms presented in this paper are of a conceptual rather than a practical nature. [6] have shown that similar algorithms for determining whether or not there are feature interactions can be implemented efficiently and give experimental evidence on realistic examples. More precisely, they implement their equivalent of our algorithm 2 using iteration instead of recursion and feature expressions instead of sets of products. Both ideas can be transferred to our setting.

Using a richer modelling language such as e.g. FORML [14], [16], one can differentiate between *intended* and *unintended* behaviour interactions, c.f. [15]. This is useful from a practical point of view and can be integrated into our approach by extending the syntax to be able to express such intentions.

To conclude, we argue that measurement of behavioural interactions in software product lines is an important part of feature-oriented software development, and that the methods we envision for doing so are both practically feasible and would be a useful addition to the developer's tool box.

References

[1] S. Apel, W. Scholz, C. Lengauer, and C. Kästner. Detecting dependences and interactions in feature-oriented design. In *ISSRE*, 2010.
[2] C. Baier and J. Katoen. *Principles of model checking*. MIT Press, 2008.
[3] 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.
[4] M. Calder and A. Miller. Feature interaction detection by pairwise analysis of LTL properties - A case study. *Formal Methods in System Design*, 28(3):213–261, 2006.
[5] A. Classen, M. Cordy, P. Schobbens, P. Heymans, A. Legay, and J. Raskin. Featured transition systems. *IEEE Trans. Software Eng.*, 39(8):1069–1089, 2013.
[6] M. Cordy, A. Classen, G. Perrouin, P. Schobbens, P. Heymans, and A. Legay. Simulation-based abstractions for software product-line model checking. In *ICSE*, 2012.
[7] A. L. J. Dominguez. Feature interaction detection in the automotive domain. In *ASE*, 2008.
[8] U. Fahrenberg and A. Legay. General quantitative specification theories with modal transition systems. *Acta Inf.*, 51(5):261–295, 2014.
[9] U. Fahrenberg and A. Legay. The quantitative linear-time–branching-time spectrum. *Theor. Comput. Sci.*, 538:54–69, 2014.
[10] P. K. Jayaraman, J. Whittle, A. M. Elkhodary, and H. Gomaa. Model composition in product lines and feature interaction detection using critical pair analysis. In *MoDELS*, 2007.
[11] M. Kolberg, E. H. Magill, D. Marples, and S. Tsang. Feature interactions in services for internet personal appliances. In *ICC*, 2002.
[12] R. Milner. *Communication and concurrency*. PHI Series in computer science. Prentice Hall, 1989.
[13] W. Scholz, T. Thm, S. Apel, and C. Lengauer. Automatic detection of feature interactions using the Java modeling language: an experience report. In *SPLC Workshops*, 2011.
[14] P. Shaker. *A Feature-Oriented Modelling Language and a Feature-Interaction Taxonomy for Product-Line Requirements*. PhD thesis, University of Waterloo, Waterloo, Ontario, Canada, 2013.
[15] P. Shaker and J. M. Atlee. Behaviour interactions among product-line features. In *SPLC*, 2014.
[16] P. Shaker, J. M. Atlee, and S. Wang. A feature-oriented requirements modelling language. In *RE*, 2012.