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. To this end, we extend the notion of simulation between transition systems to a similarity measure and lift it to compute a behaviour interaction score in featured transition systems. We then develop an algorithm which can compute the degree of feature interactions in a featured transition system in an efficient manner.**

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 interfere with each other, for example by trying to control the same variables, by issuing events that trigger other features, or by imposing conditions that suppress other features. Most of the early work on feature interactions focused on interactions that manifest themselves as logical inconsistencies, such as conflicting actions, nondeterminism, deadlock, invariant violation, or unsatisfiability [3], [11], [13], [15], [17]. More recently, [20] presented a more general definition of feature interaction – in terms of a feature that is developed and verified to be correct in isolation but is found to behave differently when combined with other features – and showed how such *behaviour interactions* could be detected as a violation of bisimulation [16].

In general, approaches to detecting feature interactions automatically ([1], [4], [5], [8], [12], [18], etc.) are effectively a *boolean* determination of whether a combination of features interact. The formulation of the result may be different for different tools: for example, an analysis may report the *set* of features that interact with a given feature f; or report the combinations of features, from a given feature set, that interact. Some techniques may report also 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. There may be multiple interactions among a set of features, where each interaction instance represents work for the developer. Specifically, each 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 than a simple interaction-existence check provides.

In practice, one must distinguish between *intended* and *unintended* behaviour interactions. An intended behaviour interaction happens when a feature is *designed* to disable certain aspects of other products, whereas an unintended interaction happens as a side effect, or as emergent behaviour, when including a new feature. The interest should, of course, lie in detecting the second, unintended type of feature interactions. For simplicity however, we only model general feature interactions, whether intended or unintended, in this work. A differentiation of intentions can be achieved by employing a richer modeling language than what we use here, but this would clutter the main points of this paper and can in any case easily be achieved by extending the syntax.

We first provide an overview of our models of features, products, and product lines, and how to use simulation to detect the presence of behaviour interactions among features [20]. 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.

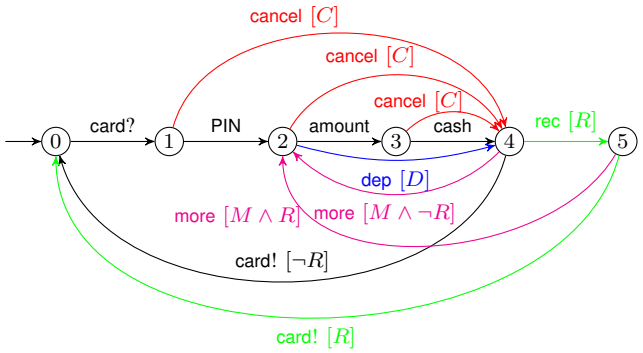

Fig. 1. FTS $\mathcal{F}$ modelling an ATM

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* $(s, a, s') \in T$*.*

A *software product line (SPL)* is a family of related software systems that share a common set of mandatory features and that differ in their selection of optional features. Let N be the set of all the optional features in all of the software systems in an SPL. A specific subset of features $p \in N$ specifies a particular software system (called a *product variant* or simply *product*) in the SPL. The full SPL is then a set of products, or a set of sets of features $px \in \mathcal{P}(N)$.

We follow [6] and consider a *feature* to be an atomic unit of behaviour that is modelled as a set of transitions. An SPL is modelled as a *featured transition system (FTS)*, in which the transitions of optional features are conditional on the presence of the feature. $\mathbb{B}(N)$ denotes the set of Boolean expressions over N. These expressions, called *feature expressions*, are used to annotate feature-specific transitions in an FTS. We write $p \models \phi$, for a product p and a feature expression ϕ, if p satisfies the feature expression ϕ. For $\phi \in \mathbb{B}(N)$ a feature expression, let $[\![\phi]\!] = \{p \subseteq N \mid p \models \phi\}$ denote the set of all products which satisfy ϕ.

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 an 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 an FTS model of an automated teller machine (ATM), which we will use as running example. It has a mandatory feature* B *that consists of a cycle of* card *insertion,* PIN *entrance,* amount *specification,* cash *retrieval, and* card *retrieval. Optional features* C, D, M*, and* R *add additional behaviours for, respectfully,* cancel*ling a transaction,* dep*ositing cash,* more *than one transaction, and obtaining a* rec*eipt. This last feature* R *is interesting, as it not only adds new behaviour to the ATM but also disables other behaviour.*

A *feature interaction* is a discrepancy between a feature's behaviour in isolation versus its behaviour in the presence of other features. Note that we take a fine-grained, *branching* view on behaviours here, hence missing behaviours will manifest themselves by missing *transitions*. It can be shown that missing branching behaviour implies missing trace-based behaviour (see also [10]), but the opposite will not always be the case.

To detect such feature interactions, we need to be able to refer to individual products within an FTS, and to a feature's behaviour within a product. We use projection over an FTS [6] to refer to specific product(s) in the FTS:

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* $T' = \{t \in T \mid [\![\gamma(t) \wedge \phi]\!] \neq \emptyset\}$*.*

Hence the projection $\pi^{\phi}(\mathcal{F})$ contains precisely those transitions, when restricted to the feature expression ϕ, can still be enabled for some products (*i.e.*, $[\![\gamma(t) \wedge \phi]\!] \neq \emptyset$). 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, a single-product projection can be seen as a plain TS. We will denote such projections as $\pi^{p}(\mathcal{F})$.

As shown in [20], a discrepancy in behaviours can be detected using bisimulation [16]. 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 [14], missed-trigger interactions [14]), 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* $\pi^{p}(\pi^{p \oplus f}(\mathcal{F}))$ *are not bisimilar.*

Here $p \oplus f$ denotes the product which contains all features of p and, additionally, the feature f. Note again that bisimilarity detects *all* feature interactions, whether intended or unintended; see [20] for a way to extend the syntax of feature specifications to be able to differentiate between these two.

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* $\pi^{B}(\pi^{B \oplus R}(\mathcal{F}))$*. We hence check whether*

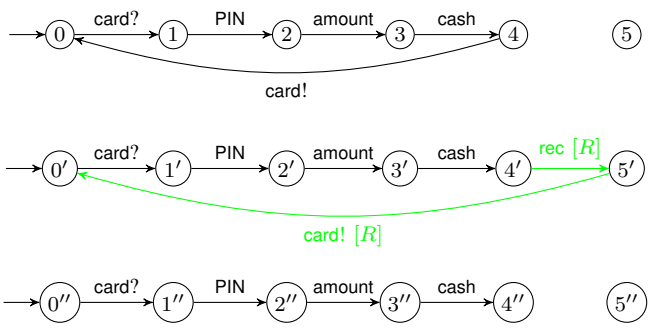

Fig. 2. Projections $\pi^B(\mathcal{F})$, $\pi^{B\oplus R}(\mathcal{F})$ and $\pi^B(\pi^{B\oplus R}(\mathcal{F}))$

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

We can immediately note a simplification in our definition of behaviour interaction above: we are comparing $\pi^p(\mathcal{F})$ with $\pi^p(\pi^{p\oplus f}(\mathcal{F}))$, but the latter TS will generally have *less* behaviour than the former, as we are projecting away the behaviour in p which is disabled by f. Hence we have the following.

Lemma 1 *A feature f has a behaviour interaction with a product p iff $\pi^p(\mathcal{F})$ is not simulated by $\pi^p(\pi^{p\oplus f}(\mathcal{F}))$.*

In [7] it is shown that FTS admit a notion of simulation at FTS level, called featured simulation:

Definition 5 ([7]) *A* featured simulation *between FTS $\mathcal{F} = (S, \Sigma, I, T, \gamma)$ and $\mathcal{F}' = (S', \Sigma, I', T', \gamma')$ is a mapping $R : S \times S' \to \mathbb{B}(N)$ such that for all $s \in S$, $s' \in S'$,*

$$R(s,s') = \bigwedge_{s \xrightarrow{a} t} \Big(\gamma(s \xrightarrow{a} t) \Rightarrow \bigvee_{s' \xrightarrow{a} t'} (\gamma'(s' \xrightarrow{a} t') \wedge R(t,t')\Big).$$

Lemma 2 ([7]) *The projection $\pi^p(\mathcal{F})$ is simulated by $\pi^p(\mathcal{F}')$ iff there exists a featured simulation R between $\mathcal{F}$ and $\mathcal{F}'$ such that $p \models \bigwedge_{i\in I} \bigvee_{i'\in I'} R(i,i')$.*

It is shown in [7] that this notion of featured simulation is useful if one wants to compute *all* products p for which $\pi^p(\mathcal{F})$ is simulated by $\pi^p(\mathcal{F}')$: According to their experiments, computing the biggest featured simulation $\bar{R}$ and then $\bigwedge_{i\in I} \bigvee_{i'\in I'} \bar{R}(i,i')$ is about 30 times faster than to check simulation for every single product.

We modify featured simulation to achieve a notion of *behaviour interaction simulation* which lifts Definition 4 to FTS level:

Definition 6 *For an FTS $\mathcal{F} = (S, \Sigma, I, T, \gamma)$ and a feature f, a* behaviour interaction simulation *with respect to f is a mapping $R : S \times S \to \mathbb{B}(N)$ such that for all $s, s' \in S$,*

$$R(s,s') = \bigwedge_{s \xrightarrow{a} t} \Big(\gamma(s \xrightarrow{a} t) \Rightarrow \bigvee_{s' \xrightarrow{a} t'} (\gamma(s' \xrightarrow{a} t') \wedge f \wedge R(t,t')\Big).$$

Similarly to Lemma 2, we now have

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

```
var Passed ← ∅
return max_{i∈I} min_{i'∈I'} dist(i, i')

function dist(s, s')
    Add (s, s') to Passed
    var m ← ∞, d ← 0
    for all s -a-> t do
        if s' -a-/-> then d ← d + 1
        else
            for all s' -a-> t' do
                if (t, t') ∉ Passed then
                    m ← min(m, dist(t, t'))
                else m ← 0
                end if
            end for
            d ← d + m
        end if
    end for
    return d
end function
```

Theorem 1 *The feature f has no behaviour interaction with p iff there exists a behaviour interaction simulation R with respect to f such that $p \models \bigwedge_{i\in I} \bigvee_{i'\in I} R(i,i')$.*

Proof: Similar to the proof of [7, Thm. 11]. ■

Hence, given a feature f, we can use Definition 6 to simultaneously compute *all* products with which f has no behaviour interaction.

III. Behavioural Distances

We wish to generalize simulation of TS to a notion which not only tells us *whether or not* there is a simulation between two TS, but (in the negative case) *how many failures* there are which prevent simulation. To this end, Algorithm 1 computes a *behavioural distance* $d(\mathcal{S}, \mathcal{S}')$ between two TS $\mathcal{S}$ and $\mathcal{S}'$ which counts the number of unique behaviours, *i.e.*, the number of behaviours which are present in $\mathcal{S}$ but not in $\mathcal{S}'$.

The intuition is that the algorithm tries to match transitions in the first TS as good as possible in the second. 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. 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.

Our behavioural distance faithfully extends the notion of simulation:

Theorem 2 *There is a simulation between TS $\mathcal{S}$ and $\mathcal{S}'$ iff $d(\mathcal{S}, \mathcal{S}') = 0$.*

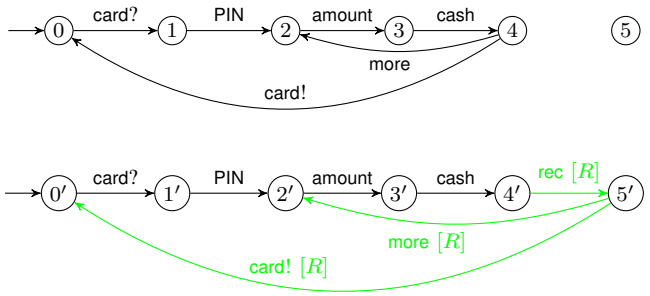

Fig. 3. Projections $\pi^{B\oplus M}(\mathcal{F})$ and $\pi^{B\oplus M\oplus R}(\mathcal{F})$

Proof sketch: If there is a simulation relation $R \subseteq S \times S$, then it can easily be shown that the algorithm will follow this relation when computing the distance, so that every time $dist(s, s')$ is called, $(s, s') \in R$. But then all transitions $s \xrightarrow{a} t$ have a match $s' \xrightarrow{a} t'$ with $(t, t') \in R$, so that $dist(s, s') = 0$. For the other direction of the proof, one easily sees that $R \subseteq S \times S'$ defined by $R = \{(s, s') \mid dist(s, s') = 0\}$ is a simulation. ■

Note that our distance is not a metric, but rather an *asymmetric pseudometric* or a *hemimetric*: It clearly holds that $d(\mathcal{S}, \mathcal{S}) = 0$ for any TS $\mathcal{S}$, and also the triangle inequality $d(\mathcal{S}, \mathcal{S}') + d(\mathcal{S}', \mathcal{S}'') \geq d(\mathcal{S}, \mathcal{S}'')$ can easily be shown (see also [10]). It is not generally true that $d(\mathcal{S}, \mathcal{S}') = 0$ implies $\mathcal{S} = \mathcal{S}'$ (identity of indiscernibles; but see Theorem 2), neither that $d(\mathcal{S}, \mathcal{S}') = d(\mathcal{S}', \mathcal{S})$ (symmetry). The lack of symmetry is natural, as we are measuring simulation rather than *bi*simulation, and by Theorem 2, $d(\mathcal{S}, \mathcal{S}') = 0$ iff $\mathcal{S}$ is simulated by $\mathcal{S}'$, hence the *kernel* of our distance is simulation.

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

Note that by Theorem 2, the behaviour interaction score is 0 iff there are no behaviour interactions.

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 $d(\pi^B(\mathcal{F}), \pi^B(\pi^{B\oplus R}(\mathcal{F})))$. 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 $\pi^B(\pi^{B\oplus R}(\mathcal{F}))$ compared to $\pi^B(\mathcal{F})$, i.e., the feature R has one behaviour interaction with the base ATM.*

We also want to know how many interactions R has with $B \oplus M$, i.e. $d(\pi^{B\oplus M}(\mathcal{F}), \pi^{B\oplus M}(\pi^{B\oplus M\oplus R}(\mathcal{F})))$. 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: $\pi^{B\oplus M}(\pi^{B\oplus M\oplus R}(\mathcal{F}))$ misses two behaviours compared to $\pi^{B\oplus M}(\mathcal{F})$, i.e., the feature R has two behaviour interactions with the product $B \oplus M$.

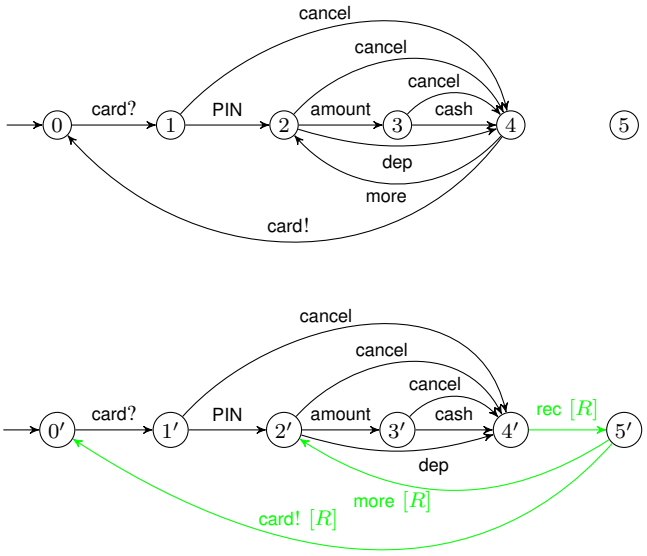

Fig. 4. Projections $\pi^{B\oplus M\oplus C\oplus D}(\mathcal{F})$ and $\pi^{B\oplus M\oplus C\oplus D\oplus R}(\mathcal{F})$

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

IV. Measuring Behaviour Interactions

A typical situation in FOSD is that one wants to see how a new feature f interacts with *all* the different products in a product line. In our setting, this means we should wish to compute the behaviour interaction score of f with respect to all products. As the number of products $p \subseteq N$ may be exponential in the number of features, it seems futile to approach this problem using Definition 7 and Algorithm 1.

Generalizing what we did at the end of Section II for the boolean check for behaviour interactions, Algorithm 2 computes, in one go, the behaviour interaction score of a given feature f with all products. Here the update function *upd* is defined as follows:

$$upd(n, s, s', k) \leftarrow \bigvee_{\substack{T_1 \subseteq \{s \xrightarrow{a} t\} \\ |\{s \xrightarrow{a} t\} \setminus T_1| = k}} \bigwedge_{s \xrightarrow{a} t \in T_1} \Big(\gamma(s \xrightarrow{a} t) \Rightarrow \bigvee_{s' \xrightarrow{a} t'} \big(\gamma(s' \xrightarrow{a} t') \wedge f \wedge (Passed(t, t') \vee fdist(n - k, t, t', \gamma(s \xrightarrow{a} t) \wedge \gamma(s' \xrightarrow{a} t'))) \big) \Big)$$

The algorithm computes a function $D : \{0, \ldots, \max\} \to \mathbb{B}(N)$, where $\max = |T|$ is the number of transitions in

Algorithm 2 Calculates behaviour interaction score $D_f(\mathcal{F})$ of f in FTS $\mathcal{F} = (S, \Sigma, I, T, \gamma)$

1: var $\max \leftarrow |T|$
2: var $D : \{0, \dots, \max\} \to \mathbb{B}(N)$
3: var *Passed* $: S \times S \to \mathbb{B}(N)$
4: **for** $n \leftarrow 0$ to $\max$ **do**
5: **for all** $s, s' \in S$ **do**
6: *Passed*$(s, s') \leftarrow$ **ff**
7: **end for**
8: $D(n) = \bigwedge_{i \in I} \bigvee_{i' \in I'}$ *fdist*$(n, i, i', \mathbf{tt})$
9: **end for**
10: **return** D

11: **function** *fdist*(n, s, s', ϕ)
12: var $d : \mathbb{B}(N)$
13: *Passed*$(s, s') \leftarrow$ *Passed*$(s, s') \vee \phi$
14: $d \leftarrow$ **ff**
15: **for** $k \leftarrow 0$ to n **do**
16: $d \leftarrow d \vee$ *upd*(n, s, s', k)
17: **end for**
18: **return** d
19: **end function**

$\mathcal{F}$, with the intention that for any product p, $p \models D(n)$ iff $d(\pi^p(\mathcal{F}), \pi^p(\pi^{p \oplus f}(\mathcal{F})) \leq n$ (see Theorem 3 below). Intuitively, the recursion to compute D works as follows.

The function *fdist*(n, s, s', ϕ) computes the feature expression covering precisely those products p for which the distance between s and s' with respect to p is at most n. This means we need to try to match all but k p-enabled transitions from s with transitions from s', where $k \leq n$, and then from any pair of states (t, t') reached in the matching, the p-distance can be at most $n - k$.

The matching itself is computed in the function *upd*. We can choose any subset S_1 of transitions from s which includes all but k of these transitions, and then for each of these transitions-to-be-matched, any product p which is enabled along the transition (*i.e.*, for which $p \models \gamma(s \to t)$) also needs to be enabled along a disjunction of transitions from s', and at the target states, p needs to be enabled for a distance of at most $n - k$.

Note how the *upd* function is a generalization to the right-hand side of the equation in Definition 6: in Definition 6, we need to match *all* transitions out of s with transitions from s', and the target states need to be related. Here, we match all but k transitions from s, and the target states need to be $(n-k)$-related. (Hence *fdist*$(n, s, s', 0)$ looks precisely like the expression in Definition 6; this is closely related to the notion of relation families for branching distances in [9].)

The algorithm again uses a *Passed* list to keep track of pairs of states which we have already seen, but now *Passed* $: S \times S' \to \mathbb{B}(N)$ returns a feature expression for each pair of states, with the intention that for any product p, we have seen a pair (s, s') iff $p \models$ *Passed*(s, s'). At each call of *fdist*(n, s, s', ϕ), *Passed*(s, s') is updated with the feature expression ϕ which has led us to (s, s'), *i.e.*, with all the products for which the transitions we took to (s, s') were enabled.

Example 4 *We compute the behaviour interaction score of R with the FTS in our example of Fig. 1. To compute $D(0)$, the algorithm calls fdist$(0, 0, 0', \mathbf{tt})$ (the FTS has unique initial state 0, hence this is what the expression in line 8 gets resolved to), which sets Passed$(0, 0')$ to* **tt** *and calls upd$(0, 0, 0', 0)$.*

Because $n = 0$, all transitions from $(0, 0')$ have to be matched, and because $\mathcal{F}$ is deterministic, matches are unique, hence the disjunctions in upd$(0, 0, 0', 0)$ disappear, and upd$(0, 0, 0', 0) = \big(B \Rightarrow (B \wedge R \wedge$ fdist$(0, 1, 1', B))\big)$.

Hence the algorithm now calls fdist$(0, 1, 1', B)$, which sets Passed$(1, 1')$ to B and, similarly to the above, sets out to compute upd$(0, 1, 1', 0) = \big(B \Rightarrow (B \wedge R \wedge$ fdist$(0, 2, 2', B))\big)$.

After two more steps, the algorithm has also set Passed$(2, 2') =$ Passed$(3, 3') =$ Passed$(4, 4') = B$ and called fdist$(0, 4, 4', B)$. Now upd$(0, 4, 4', 0)$ wants to match the more*-transition from 4 to 2, leading to the expression $(B \wedge M \wedge \neg R) \Rightarrow (B \wedge M \wedge \neg R \wedge R \wedge \dots)$, which gets pruned to $\neg B \vee \neg M \vee R$ because the right-hand side of the implication is false. Similarly, when matching the* card*-transition, the expression gets pruned to $\neg B \vee R$. Lastly, we need to match the* rec*-transition, leading to a call of fdist$(0, 5, 5', B)$ which returns* **tt**. *The return expression of upd$(0, 4, 4', 0)$ is hence $(\neg B \vee \neg M \vee R) \wedge (\neg B \vee R) = \neg B \vee R$.*

The expression returned by fdist$(0, 4, 4', B)$ is thus $\neg B \vee R$. Intuitively, this tells us that R only has zero behaviour interactions if the base feature is not present or R already is in the product. As we know from Example 3 that R has a behaviour interaction with the base feature, this is the expected result.

In the rest of the computation, the expression $\neg B \vee R$ now gets back-propagated to the pair $(0, 0')$ of initial states; due to the Passed-values, no more function calls are initiated: upd$(0, 3, 3', 0)$ also needs to match the cancel*-transition and returns $(B \Rightarrow (B \wedge R \wedge (\neg B \vee R))) \wedge ((B \wedge C) \Rightarrow (B \wedge C \wedge R \wedge (B \vee$ fdist$(0, 4, 4', B \wedge C)))) = (\neg B \vee R) \wedge (\neg B \vee \neg C \vee \neg R) = \neg B \vee R$. Hence fdist$(0, 3, 3', B) = \neg B \vee R$, and similarly fdist$(0, 2, 2', B) = \neg B \vee R$, fdist$(0, 1, 1', B) = \neg B \vee R$ and finally $D(0) = \neg B \vee R$. Again, Example 3 tells us that this is the expected result.*

Now for the computation of $D(1)$. The expression in line 8 resolves to call fdist$(1, 0, 0', \mathbf{tt})$, which initiates recursive calls to fdist$(1, 1, 1', B)$, fdist$(1, 2, 2', B)$, fdist$(1, 3, 3', B)$, and finally fdist$(1, 4, 4', B)$ like above. All these are for $k = 0$ in the loop in line 15; for $k = 1$, fdist$(1, 0, 0', \mathbf{tt})$ just returns **tt** *as no transitions need to be matched, and in the other state pairs, this ends up in a computation of fdist$(0, 4, 4', B)$ which will not matter in the end.*

In the computation of fdist$(1, 4, 4', B)$, the case $k = 0$ contributes $\neg B \vee R$ like above, which is expected: in this case, all *transitions from $(4, 4')$ need to be matched, and the resulting state pairs either have distance 0 or have been passed already, hence this is precisely the same computation as for fdist$(0, 4, 4', B)$ above.*

For the case $k = 1$, the computations in upd$(1, 4, 4', 1)$, to match the three transitions labeled more, card *and* rec, *respectively, return $E_{\mathsf{more}} = \neg B \vee \neg M \vee R$, $E_{\mathsf{card}} = \neg B \vee R$ and $E_{\mathsf{rec}} = \mathbf{tt}$ as above, but now we do not compute their conjunction. As one of the transitions can remain unmatched, we instead compute $(E_{\mathsf{more}} \wedge E_{\mathsf{card}}) \vee (E_{\mathsf{more}} \wedge E_{\mathsf{rec}}) \vee (E_{\mathsf{card}} \wedge E_{\mathsf{rec}})$, which returns $\neg B \vee \neg M \vee R$.*

The expression returned by fdist$(1, 4, 4', B)$ is thus upd$(1, 4, 4', 0) \vee$ upd$(1, 4, 4', 1) = \neg B \vee \neg M \vee R$. Intuitively, this tells us that R has at most one behaviour interaction if the base feature is not present, R already is in the product, or the M feature is not present. Hence any product without M has at most one behaviour interaction with R (and by $D(0) = \neg B \vee R$, precisely one), which conforms to the results of Example 3. In the rest of the computation, the expression $\neg B \vee \neg M \vee R$ gets back-propagated to the initial pair $(0, 0')$ like above.

To compute $D(2)$, the algorithm calls fdist$(2, 0, 0', \mathbf{tt})$, which again results in recursive calls to fdist$(2, 1, 1', B)$, fdist$(2, 2, 2', B)$, fdist$(2, 3, 3', B)$ and fdist$(2, 4, 4', B)$. In this case however, upd$(2, 4, 4', 2)$ only needs to match one of the three outgoing transitions of state 4, and as the rec*-transition can be matched perfectly, upd$(2, 4, 4', 2) = \mathbf{tt}$. Hence fdist$(2, 4, 4', B)$ returns $\mathbf{tt}$, which gets back-propagated to give $D(2) = \mathbf{tt}$. Hence R has at most two behaviour interactions with any product, and we can stop the calculation at this point.*

Theorem 3 *The feature f has at most n behaviour interactions with p iff $p \models D_f(\mathcal{F})(n)$.*

Proof sketch: As an invariant of the algorithm, one can show that in each pair of states (s, s') in which a number k of p-enabled transitions from s cannot be matched by p-enabled transitions from s', it holds for all m that $p \models$ *fdist*(m, s, s') implies $p \models \bigwedge_{s \xrightarrow{a}_p t} \bigvee_{s' \xrightarrow{a}_p t'}$ *fdist*$(m - k, t, t')$. Hence $p \models D_f(\mathcal{F})(n)$ implies that there is a computation of the algorithm which notes at most n behaviour interactions and ends in a state pair (s, s') in which $p \models$ *fdist*$(0, s, s')$, *i.e.*, from which there are no more behaviour interactions. Conversely, if f has at most n behaviour interactions with p, then there is a computation of the algorithm which notes at most n behaviour interactions and ends in a state pair (s, s') in which $p \models$ *fdist*$(0, s, s')$, thus $p \models D_f(\mathcal{F})(n)$. ■

We notice in the example that $D(0) \Rightarrow D(1) \Rightarrow D(2)$. In general, $D(k) \Rightarrow D(k + 1)$ will always hold, as any product which has behaviour interaction score at most k with f also has score at most $k + 1$. Hence the expression $D(k)$ can be re-used in the computation of $D(k+1)$, making the algorithm more efficient. Similarly, we notice that many recursive calls are avoided because subformulas evaluate to **tt** or **ff**; this should also generally be the case. Altogether we believe, and [7] seems to support this, that an efficient implementation of algorithm 2 will be able to calculate the feature interaction score of f with respect to all products much faster than a naïve approach using projections and algorithm 1.

V. Conclusion

We have defined a technique to measure the degree to which features within a software product line interact with each other. Informally, feature interactions within a product $p \oplus f$ manifest themselves as differences between how the sub-product p behaves versus how p behaves when extended to include feature f. We use simulation to detect these differences as simulation violations between p and the projection of p's behaviours within the product $p \oplus f$. In this paper, we propose counting the number of simulation violations, as a means of measuring the number of distinct interactions between a feature f and the features in a product p. To be complete, one would want to know the number of interactions that a feature has with *any* of the SPL's products. Thus, this paper also provides an algorithm that, with a single analysis of an SPL and a given feature f, reports for each product p in the SPL the number of interactions between the features of p and f.

This metric is useful as an indicator of the relative amount of work needed to integrate a feature into the products of a product line. Each feature interaction needs to be analyzed (to determine if it is a problem), each undesired interaction needs to be fixed, and each fixed interaction needs to be tested. Thus, larger behavioural distances between f and the products of an SPL indicate greater amounts of rework needed to analyze and fix interactions and thereby properly integrate feature f into the product line.

For future work, we intend to explore how our technique could be extended to distinguish between *intended* interactions (such as the simulation violations introduced between a feature R and the base feature B that it extended) and *unintended* interactions, which occur between distinct features that extend the same base feature. Using a richer modelling language such as for example FORML [19], [21], one can differentiate between these types of behaviour interactions, see [20]. 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.

We also plan to implement our agorithm and extend it so that it also shows precisely *where* features interact, hence giving visual feed-back to the developer where there may be problems in the model. We have already mentioned a number of possible optimizations of the basic algorithm; we intend to implement these and see whether the algorithm scales to real-life product line case studies.

Our proposed behavioural distance is but one example of a so-called *branching distance* between transition systems [9], [10], and many other such distances may be defined. Precisely which of them are useful in product-line analyses remains to be seen.

References

[1] S. Apel, W. Scholz, C. Lengauer, and C. Kästner, "Detecting dependences and interactions in feature-oriented design," in *ISSRE*, 2010, pp. 161–170. [Online]. Available: http://dx.doi.org/10.1109/ISSRE.2010.11

[2] C. Baier and J. Katoen, *Principles of model checking*. MIT Press, 2008.

[3] J. Blom, B. Jonsson, and I. Kempe, "Using temporal logic for modular specificaition os telephone services," in *Proc. of Feature Interaction Workshop*, 1994, pp. 197–214.

[4] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec, "Feature interaction: a critical review and considered forecast," *Computer Networks*, vol. 41, no. 1, pp. 115–141, 2003. [Online]. Available: http://dx.doi.org/10.1016/S1389-1286(02)00352-3

[5] M. Calder and A. Miller, "Feature interaction detection by pairwise analysis of LTL properties - A case study," *Formal Methods in System Design*, vol. 28, no. 3, pp. 213–261, 2006. [Online]. Available: http://dx.doi.org/10.1007/s10703-006-0002-5

[6] A. Classen, M. Cordy, P. Schobbens, P. Heymans, A. Legay, and J. Raskin, "Featured transition systems," *IEEE Trans. Software Eng.*, vol. 39, no. 8, pp. 1069–1089, 2013. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/TSE.2012.86

[7] 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, pp. 672–682. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2012.6227150

[8] A. L. J. Dominguez, "Feature interaction detection in the automotive domain." in *ASE*, 2008, pp. 521–524.

[9] U. Fahrenberg and A. Legay, "General quantitative specification theories with modal transition systems," *Acta Inf.*, vol. 51, no. 5, pp. 261–295, 2014. [Online]. Available: http://dx.doi.org/10.1007/s00236-014-0196-8

[10] ——, "The quantitative linear-time–branching-time spectrum," *Theor. Comput. Sci.*, vol. 538, pp. 54–69, 2014. [Online]. Available: http://dx.doi.org/10.1016/j.tcs.2013.07.030

[11] P. Gibson, "Towards a feature interaction algebra," in *Proc. of Feature Interactions in Telecommunications and Software Systems V*, 1998, pp. 217–231.

[12] 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, pp. 151–165. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-75209-7_11

[13] A. Khoumsi, "Detection and resolution of interactions between services of telephone networks," in *Proc. of Feature Interaction Workshop IV*, 1997, pp. 78–82.

[14] M. Kolberg, E. H. Magill, D. Marples, and S. Tsang, "Feature interactions in services for internet personal appliances," in *ICC*, 2002, pp. 2613–2618.

[15] T. LaPorta, D. Lee, Y.-J. Lin, and M. Yannakakis, "Protocol feature interactions," in *FORTE*, 1998, pp. 59–74.

[16] R. Milner, *Communication and concurrency*, ser. PHI Series in computer science. Prentice Hall, 1989.

[17] M. Rinard, A. Salciam, and S. Bugrara, "A classification system and analysis for aspect-oriented programs," in *Proc. of Foundations of Software Engineering (FSE)*, 2004, pp. 147–158.

[18] W. Scholz, T. Thüm, S. Apel, and C. Lengauer, "Automatic detection of feature interactions using the Java modeling language: an experience report." in *SPLC Workshops*, 2011, p. 7.

[19] P. Shaker, "A feature-oriented modelling language and a feature-interaction taxonomy for product-line requirements," Ph.D. dissertation, University of Waterloo, Waterloo, Ontario, Canada, 2013.

[20] P. Shaker and J. M. Atlee, "Behaviour interactions among product-line features," in *SPLC*, S. Gnesi, A. Fantechi, P. Heymans, J. Rubin, and K. Czarnecki, Eds., 2014, pp. 242–246. [Online]. Available: http://doi.acm.org/10.1145/2648511.2648538

[21] P. Shaker, J. M. Atlee, and S. Wang, "A feature-oriented requirements modelling language," in *RE*, M. P. E. Heimdahl and P. Sawyer, Eds., 2012, pp. 151–160. [Online]. Available: http://dx.doi.org/10.1109/RE.2012.6345799