Optimizing Feature Interaction Detection

Alessandro Fantechi[1,2], Stefania Gnesi[2], and Laura Semini[2,3(✉)]

[1] Dip. di Ing. Dell'Informazione, Università di Firenze, Firenze, Italy
[2] ISTI-CNR, Pisa, Italy
[3] Dipartimento di Informatica, Università di Pisa, Pisa, Italy
semini@di.unipi.it

Abstract. The feature interaction problem has been recognized as a general problem of software engineering. The problem appears when a combination of features interacts generating a conflict, exhibiting a behaviour that is unexpected for the features considered in isolation, possibly resulting in some critical safety violation. Verification of absence of critical feature interactions has been the subject of several studies. In this paper, we focus on functional interactions and we address the problem of the 3-way feature interactions, i.e. interactions that occur only when three features are all included in the system, but not when only two of them are. In this setting, we define a widely applicable definition framework, within which we show that a 3 (or greater)-way interaction is always caused by a 2-way interaction, i.e. that pairwise sampling is complete, hence reducing to quadratic the complexity of automatic detection of incorrect interaction.

1 Introduction

The specification of a complex software system may be simplified by decomposing the system into *features* that identify units of functionality. Feature-oriented software development of safety critical systems can simplify the configuration of large systems, as well as their verification and certification, by concentrating the verification efforts on single features, rather than on the whole system. But this happens only if a high degree of independence between features can be assumed, while frequently instead the *feature interaction* problem can be encountered, a problem which occurs when the concurrent composition of two (or more) features generates an unexpected behaviour.

The feature interaction problem has indeed been recognized as a general problem of software engineering in all those contexts where *features* are the basic functionality units that are composed to build up complex software systems [1,9,25,31], as also recently advocated in [2]. In particular, if a feature interaction affects critical systems, it may cause safety requirements violation; hence verification of the absence of feature interactions becomes a very important aspect of safety certification. The question is how many features are required to generate an interaction, two or more than two. In this paper, we concentrate on the so-called "3-way interaction". The problem was first discussed in the feature

L. Petrucci et al. (Eds.): FMICS-AVoCS 2017, LNCS 10471, pp. 201–216, 2017.
DOI: 10.1007/978-3-319-67113-0_13

interaction detection contest at [18], where the community suggested that there are two types of 3-way interaction: those reducible to an interaction between a pair of features and those where the interaction only exists if all three features are present. The latter were termed "true" 3-way interactions.

The very existence of such cases is still under discussion, in the sense that there does not seem to be a consensus on the definition of the problem itself. For example, in [17,18] the existence of 3-way interactions is negated but with no proof. On the other end, in [3] an example of 3-way interaction is reported. The question is quite important for the verification of safety critical systems that are built by feature-oriented development: if we can limit the definition of features to a framework where "true" 3-way feature interactions do not exist, then the problem of checking for feature interactions can be reduced to checking features pairwise, hence with reduced (quadratic) verification complexity. In order to give a contribution to the New Feature Interaction Challenge [2], this paper offers a framework of feature definition by condition-action rules and interleaved composition, and presents a definition of a feature interaction as when the execution of one feature disallows the execution of another or when the two possible results of the interleaved execution of two features are inconsistent with each other. In this framework we then prove that any 3-way interaction is due to a 2-way interaction. In regards to verification, this amounts to say that checking by *pairwise sampling* [20,21,26] the combinations of features is complete with respect to feature interaction detection.

The proposed framework is contrasted with cases reported in the literature of 3-way interactions and discusses why these are not considered true 3-way interactions according to our behavioural interpretation of composition and interaction.

In the following, we define a running example: the features of a metro train (Sect. 2), and the formalisation of features and feature composition (Sect. 3). We then prove that the interactions among three features can always be revealed by checking for the 2-way interactions, therefore reducing the complexity of the verification problem (Sect. 4). A section on related work concludes the paper.

2 Running Example

As an example, we consider a control system composed by the following features, each feature devoted to the actuation of a separate requirement over safety-related behaviour of a metro train, regarding the usage of emergency brakes and the opening of doors, in normal situations or when a smoke sensor detects a fire. The train can be travelling in a tunnel, in which case safety regulations require that the train cannot be stopped even in case of fire. On the other hand, doors, normally opened only at stations, cannot be opened, even in emergency situations, when the train is running. If not in a tunnel, the train can normally be running in the open air or at standstill in a station. We assume that there are smoke sensors and that whether the train is running in a tunnel or at standstill in a station or elsewhere is known to the system through proper positioning sensors.

SD Station & Doors:	If the train is at a station, the doors are opened
DS Danger in Station:	If the train is at a station and there is a danger in the station, doors are closed and the train leaves the station
EH Emergency Handle:	If the emergency handle is pulled, actuate the emergency brake
TB Tunnel & Brake:	If the train is in a tunnel, disable the emergency brake
FA Fire alarm:	Raise a fire alarm when smoke is sensed
FB Fire alarm & Brake:	If fire alarm is raised and the train is running, actuate the emergency brake
FE Fire alarm & Escape:	If fire alarm is raised, open the doors

It is easy to note that, due to some interactions between the above features, we may have interacting behaviours. It is also apparent that, in order to provide a safe global behaviour, some form of conflict resolution is needed, possibly prioritizing some features with respect to others.

Below, we discuss all the possible interactions between the metro features and address their detection.

For instance, *EH* interferes with *TB* since, if applied concurrently, i.e. when the emergency handle is pulled and the train is in a tunnel, their actions conflict.

A 3-way interaction refers to those cases in which the interaction is generated by the composition of three features. Apparently, in the metro example there is a 3-way interaction among *FA*, *FB*, and *TB*. Assume that smoke is sensed while the train is in a tunnel, *FA* and *FB* are applied in sequence:

$$smoke\_sensed \xrightarrow{FA} fire\_alarm\_raised \xrightarrow{FB} emergency\_brake$$

and interact with *TB* disabling the emergency brake. In the paper we will provide a constructive technique to detect these interactions with pairwise analysis: such a technique will detect that the interaction is between *FB* and *TB*.

3 Formalisation of Features and Feature Interaction

Feature interaction is due to a mutual interference resulting in an unexpected behaviour. The most common way to define a feature interaction is based on behaviours:

> "A feature interaction occurs when the behavior of one feature is affected by the presence of another feature" [1].
> "A feature interaction is some way in which a feature or features modify or influence another feature in defining overall system behavior" [32].

Similar definitions can be found e.g. in [11,23,27]. This mutual influence of features is often described in an action oriented way, by listing the pairs of conflicting actions, and then deriving possible interactions between features including these actions. In a complementary way, we consider a state-based approach [24] and look at the effect of the features on a shared state. Indeed, any time two features F and F' access a shared state, and at least one of the accesses updates it, there might be an interaction.

The main purpose of this paper is to prove that, in the considered framework, the behavioural interactions between three features are always due to the interaction between two of the three considered features, therefore reducing the complexity of the verification problem to look for pairwise interaction. To do this we need to perform an analysis of the functional behaviour of feature combinations.

Without loss of generality, we define a framework in which features are described as condition-action rules and systems behave as the parallel composition of features [23,30]. In this framework, inspired by the *action systems* [5], if the action part of a condition-action rule of a feature is executed, it changes the state of the system: the state of a system is seen as a set of predicates that hold on some global, shared variables. A feature is said to be *enabled* when its condition is satisfied by the current state of the system. The application of the feature can occur only when it is enabled, having the effect of changing the state of the system according to its action. The computation of the system is given by a sequence of feature applications. When two or more features are enabled, one is selected non-deterministically.

In this section, we define a formalisation for the computation state and give the semantics of features in terms of transition systems.

3.1 Semantics of a Feature

Definition 1. *Let $\mathcal{S}$ be a finite set of states. Given a set AP of atomic propositions, with p ranging in AP, a* **computation state** *$s \in \mathcal{S}$ is defined as a conjunction of literals:*

$$s ::= \bot | p | \sim p | s \wedge s$$

where $\bot$ is the empty state, in which nothing is said on any atomic proposition.

Example 1. *Examples of computation states are: $s_1 = doors\_open$, $s_2 = tunnel \wedge \sim doors\_open$, $s_3 = \sim tunnel \wedge doors\_open \wedge smoke\_sensed$.*

We include negative atoms for convenience. An alternative modeling would have been defining states as conjunctions of atomic propositions, in a closed world assumption.

We assume the set of actions to be in correspondence with the set of predicates, i.e. each action α has an effect on the truth value of a predicate p: α can make p true or make p false.

Definition 2. *A* **feature** *is defined by a pair: $F = \langle C, [A] \rangle$, where C is a boolean condition to be evaluated on the current state, and $[A]$ is an (atomic) sequence of actions on the state.*

Example 2. *The features of our running metro example can be specified as:*
$SD = \langle station, [open\_doors] \rangle$
$DS = \langle station \wedge danger, [close\_doors, leave\_station] \rangle$
$EH = \langle ehpulled, [activate\_emergency\_brake] \rangle$
$TB = \langle tunnel, [disable\_emergency\_brake] \rangle$
$FA = \langle smoke\_sensed, [raise\_fire\_alarm] \rangle$
$FB = \langle fire\_alarm\_raised \wedge running, [activate\_emergency\_brake] \rangle$
$FE = \langle fire\_alarm\_raised, [open\_doors] \rangle$

An action α transforms a state s in a state $\alpha(s)$.

Definition 3. *We say that action α* **writes** *p when α makes p true, i.e. $\alpha(\bot) = p$ and that α* **writes** *$\sim p$ when α falsifies p, i.e. $\alpha(\bot) = \sim p$.*

For instance, action *open_doors* writes *doors_open*, *disable_emergency_brake* writes $\sim$ *emergency_brake*, and *leave_station* writes $\sim$ *station*. We define now the effect of an action on a general state.

Definition 4. *Given a state s, and a literal p, we say that $s \models p$ when p occurs as a literal in the conjunction of literals representing s.*

Definition 5. *Let s be a state, p a predicate, and α an action writing p, we define the effect of the application of α in s, $\alpha(s)$ by cases:*

$$\alpha(s) = \begin{cases} s \wedge p \text{ if } s \not\models p \text{ and } s \not\models \sim p \\ s \qquad \text{if } s \models p \\ \hat{s} \wedge p \text{ if } s \models \sim p, \text{ i.e. } s \text{ can be written as } \hat{s} \wedge \sim p \end{cases}$$

We have a symmetric definition when α writes $\sim p$. For instance, let s be a state not telling whether the doors are open or not, then:

$$close\_doors(s) = s \wedge \sim doors\_open$$

Therefore, action *close_doors* is the identity when applied on a state with doors already closed, and changes the truth value of *doors_open* when initially true, i.e.:

$$close\_doors(s \wedge \sim doors\_open) = s \wedge \sim doors\_open$$
$$close\_doors(s \wedge doors\_open) = s \wedge \sim doors\_open$$

To define the semantics of a feature, we model the effect on a state of a sequence of actions as an atomic transition.

Definition 6. *Let A be a sequence of actions $\alpha_1, \ldots, \alpha_n$, we say $s \xrightarrow{A} s'$ when $s' = \alpha_n(\alpha_{n-1}(...\alpha_1(s)...))$.*

Definition 7. *We say that a feature* $F = \langle C, [A] \rangle$ *is* **enabled** *in a state* s *when* $s \models C$.

Definition 8. *The* **semantics of feature** $F = \langle C, [A] \rangle$ *is the set of all pairs of states* $(s, s') \in \mathcal{S} \times \mathcal{S}$ *such that* F *is enabled in* s *and* $s \xrightarrow{A} s'$.

In such a case, we also write $s \xrightarrow{F} s'$.

Example 3. *As an example, consider feature* FA, *and a state* s *satisfying* $smoke\_sensed$. *For simplicity we take* $s = smoke\_sensed$. *We have:*

$$smoke\_sensed \xrightarrow{FA} smoke\_sensed \wedge fire\_alarm\_raised$$

3.2 Composition of Features

Definition 9. *A* **software system** *is specified as the parallel composition of features:* $F_1 || \ldots || F_n$.

Example 4. *The metro system can be specified as :* $SD||DS||EH||TB||FA||FB||FE$.

The **semantics of a software system** composed of features is given as a labeled transition system $(\mathcal{S}, \mathcal{S}_0, \mathcal{F}, \rightarrow)$ where: $\mathcal{S}$ is the set of states, $\mathcal{S}_0 \subseteq \mathcal{S}$ is a set of initial states, $\mathcal{F}$ is the set of features, and $\rightarrow \subseteq \mathcal{S} \times \mathcal{F} \times \mathcal{S}$ is a transition relation, whose elements, written $s \xrightarrow{F} s'$, are given by all the pairs of states s, s' of the semantics of each feature F, according to Definition 8.

Definition 10. *According to an interleaving semantics of the parallel composition, the* **semantics of a software system** $F_1||F_2$, *with* $F_1 = \langle C_1, A_1 \rangle$ *and* $F_2 = \langle C_2, A_2 \rangle$, *is given by the labeled transition system* $(\mathcal{S}, \mathcal{S}_0, \mathcal{F}, \rightarrow)$, *generated by the alternative sequences of transitions possible from any initial state in* $\mathcal{S}_0$, *applying one of the features, followed by the other one.*

Hence, the application of features $F_1||F_2$ *to an initial state* s *generates the following transitions:*

- $s \xrightarrow{F_1} s' \xrightarrow{F_2} s''$ *if* $s \models C_1$ *and* $s' \models C_2$
- $s \xrightarrow{F_2} s' \xrightarrow{F_1} s''$ *if* $s \models C_2$ *and* $s' \models C_1$
- $s \xrightarrow{F_1} s'$ *if* $s \models C_1$ *and* $s' \not\models C_2$
- $s \xrightarrow{F_2} s'$ *if* $s \models C_2$ *and* $s' \not\models C_1$

We write:

$$s \xrightarrow{F_1||F_2} s'$$

as a shorthand for any sequence of transitions from s *to* s' *applying the features in the parallel composition.*

Definition 10 easily extends to the parallel composition of n features: $F_1||\ldots||F_n$:

$$s \xRightarrow{F_1||\ldots||F_n} s'$$

is the result of applying, in any possible ordering, the features $F_1, \ldots F_n$.

Example 5. *Let us consider $F_1||F_2$, where: $F_1 = \langle p, [\alpha_1]\rangle$, $F_2 = \langle p, [\alpha_2]\rangle$, α_1 writes r, and α_2 writes q. We have:*

$$p \xrightarrow{F_1} p\wedge r \xrightarrow{F_2} p\wedge r\wedge q \qquad \textit{and} \qquad p \xrightarrow{F_2} p\wedge q \xrightarrow{F_1} p\wedge r\wedge q$$

i.e. $p \xRightarrow{F_1||F_2} p\wedge r\wedge q$ and we have two traces from p to $p\wedge r\wedge q$.

It is not guaranteed that all traces of $F_1||F_2$ from s converge in a unique state: it can happen that $s \xRightarrow{F_1||F_2} s'$ and $s \xRightarrow{F_1||F_2} s''$ with $s' \neq s''$, as in the following example.

Example 6. *Now consider $F_1||F_3$ where F_1 is as above, $F_3 = \langle p, [\alpha_3]\rangle$, and α_3 writes $\sim r$. We have:*

$$p \xrightarrow{F_1} p\wedge r \xrightarrow{F_3} p\wedge \sim r \qquad \textit{and} \qquad p \xrightarrow{F_3} p\wedge \sim r \xrightarrow{F_1} p\wedge r$$

i.e. $p \xRightarrow{F_1||F_3} p \wedge r$ and $p \xRightarrow{F_1||F_3} p\wedge \sim r$

Example 6 introduces a feature interaction: feature F_1 interacts with F_3 since $F_1||F_3$ can lead to different states, depending on the order of feature application.

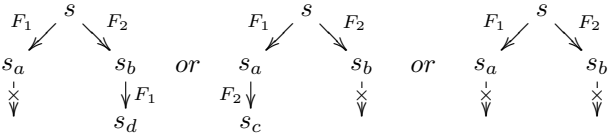
In Sect. 3.3, we formalise the concept of interaction.

3.3 Formalisation of Interaction

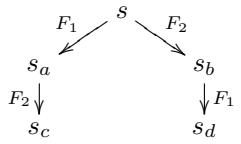
We now give a formal definition of interaction between pairs of features.

Definition 11. *There is an* **interaction** *in $F_1||F_2$, where $F_1 = \langle C_1, [A_1]\rangle$ and $F_2 = \langle C_2, [A_2]\rangle$, when, given a state s enabling both, i.e. such that $s \models C_1 \wedge C_2$, one of the following two situations occurs:*

1. $s \xrightarrow{F_1} s_a$, and $s_a \not\models C_2$, or $s \xrightarrow{F_2} s_b$ and $s_b \not\models C_1$, or both, i.e.:



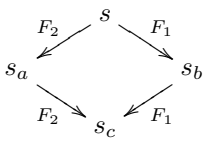
2. $s \xRightarrow{F_1||F_2} s_c$, $s \xRightarrow{F_1||F_2} s_d$ and s_c, s_d are inconsistent *(that is, $s_c \wedge s_d = false$)*



$with\ s_c \wedge s_d \Leftrightarrow false$

Example 7. *There is an interaction of the first kind in $SD||DS$. In fact, in case of danger, if DS is applied first, it takes to a state that satisfies the condition: $\sim$ station and where SD is no longer enabled.*
There is an interaction of the second kind in $EH||TB$ since they both can be applied if the condition ehpulled$\wedge$tunnel holds, and their executions go to inconsistent states.

Dually, according to Definition 11, the system $F_1||F_2$ is *interaction free* either if the features can never be applied in the same state ($C_1 \wedge C_2 \Leftrightarrow false$) or if the execution of any of them does not falsify the condition of the other and the order of execution is irrelevant, i.e. the diagram commutes:

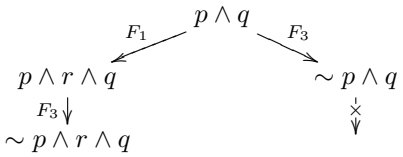


Definition 11 is extended straightforwardly to the parallel composition of **three features**: the features interact when there exists a state s enabling all of them, and the paths of the transition system rooted in s do not converge in a common final state. This happens, for instance, in $FA||FB||TB$, taking $s = smoke\_sensed \wedge tunnel$.

We also consider a second form of 3-way interaction arising in a situation where such a state cannot be built. This happens when a feature is enabled by p and another by $\sim p$, for some p, and a third feature writes $\sim p$ (p resp.). As the most general case, consider $F_1||F_2||F_3$, where:

$$F_1 = \langle p, [writes\ r]\rangle \qquad F_2 = \langle \sim p, [writes\ s]\rangle \qquad F_3 = \langle q, [writes \sim p]\rangle$$

In this case the three features are not enabled together in any state, but the triggering of F_3 enables F_2 as well, so that starting from a state in which F_1 and F_3 are enabled, we can derive the following diagram, rooted in $p \wedge q$, which does not converge to a unique state. Actually, this 3-way interaction is due to a pairwise interaction in the subsystem $F_1||F_3$:



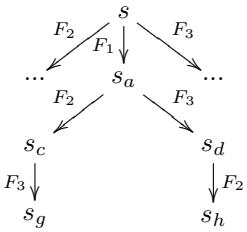
4 No True 3-Way Interaction

Now, we can address the main point of the paper and prove that, under our definition of feature interaction, any 3-way interaction is due to the interaction between two of the considered features. We first observe that there is a constructive way to find a (minimal) state enabling two (or more) features: it is sufficient to take the conjunction of their conditions.

Proposition 1. *Let F_1, F_2, and F_3 be a triple of interacting features, then there is an interaction between at least a pair of them.*

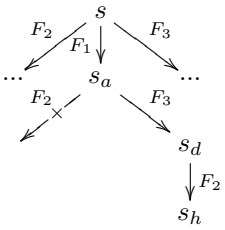
Proof. The 3-way interaction means that for some s satisfying the conditions of the three features we have one of the following cases:

Case 1. It is not possible to complete all the possible sequences of transitions for $s \xRightarrow{F_1||F_2||F_3} s'$, i.e. one of the six sequences



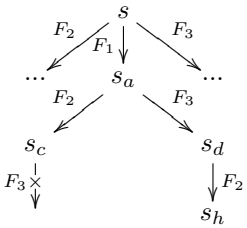
stops either after one step or two steps, because the conditions of the remaining feature(s) are not satisfied.

– *Case 1a.* (one step). Let a $s \xrightarrow{F_1} s_a$ with s_a not satisfying the condition of F_2 (resp. F_3),



then F_1 interacts with F_2 (resp.F_3).

– *Case 1b.* (two steps). Let the subtree rooted in s_a be incomplete, e.g.

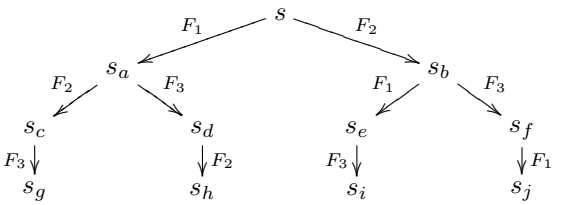


in this case, F_2 interacts with F_3.

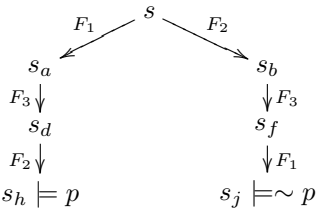
Case 2. In the semantics of $F_1||F_2||F_3$, there are (at least) two sequences of transitions rooted in a common state and reaching two states which are inconsistent, i.e., for some s:

$$s \xRightarrow{F_1||F_2||F_3} s' \quad \text{and} \quad s \xRightarrow{F_1||F_2||F_3} s''$$

and $s' \wedge s'' = false$. We build the following tree and reason by cases. The tree is partial since it is sufficient to find two sequences. We can assume that none of them has F_3 at the first step (the general result is obtained with a label switching).



- *Case 2a.* s_g, s_h are inconsistent (similar reasoning for s_i, s_j): in this case the subtree rooted in s_a leads to inconsistent states, hence there is an interaction in $F_2||F_3$.
- *Case 2b.* s_g, s_i are inconsistent. Both s_g and s_i are the result of the application of F_3. This means that already s_c, s_e were inconsistent, hence the interaction is in $F_1||F_2$.
- *Case 2c.* s_h, s_j are inconsistent. This entails that there is a predicate p true in s_h and false in s_j (or vice-versa). We restrict our attention to the interested tree fragment.



Assume p was false in s (similar reasoning with p true in s). Predicate p can be true in s_h only if there is a feature F_i writing p. As a consequence there is another feature F_j writing $\sim p$ (otherwise, since F_i is also in the right path, p would be true in s_j). We can say that $F_i \neq F_1$, since F_1 makes the last step before s_j. Similarly, we can say that $F_j \neq F_2$.

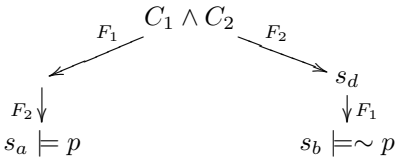
We are thus left with three cases:

(2.c.1) F_2 writes p and F_1 writes $\sim p$

(2.c.2) F_2 writes p and F_3 writes $\sim p$

(2.c.3) F_3 writes p and F_1 writes $\sim p$

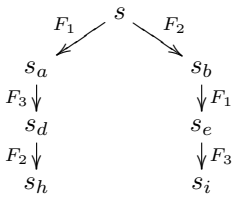
We only show the proof of the first one, since the other cases use the same reasoning. Let F_2 write p and F_1 write $\sim p$. We consider the tree rooted in the subset of s satisfying $C_1 \wedge C_2$ (we recall that, for the initial hypothesis of the proposition, s satisfies $C_1 \wedge C_2 \wedge C_3$). Either one path is blocked after one step, or the tree is the following:



In both cases there is an interaction between F_1 and F_2.

– *Case 2d.* s_h, s_i are inconsistent (Similar reasoning for s_g, s_j).

We redraw the interested fragment of the initial tree:



As discussed above, there must be:

– a predicate p with $s_h \models p$ and $s_i \models\sim p$;
– a feature writing p and a feature writing $\sim p$.

One of these features must be F_3, otherwise there is an inconsistency at the first step proving the interaction between F_1 and F_2.

State $s_i \models\sim p$ implies that F_3 writes $\sim p$.

If F_1 writes p we consider the tree rooted in $C_1 \wedge C_3$ and prove the interaction between F_1 and F_3. Otherwise, F_2 writes p, we derive the interaction between F_2 and F_3. □

5 Related Work

Li et al. [19] define a technique for the analysis of feature interactions where features are complex state machines and the paper defines how to abstract their behaviour in a compact model. Abstraction makes analysis of the composition of the models much simpler, though sound and complete, with respect to the analysis of the composition of the base machines. The approach for analysing the composition of the models is not far from the one proposed in this paper

for analysing the pairwise composition of features (in our setting features are already abstract).

Previous work on feature interactions addressed logical inconsistencies between features, due to conflicting actions, nondeterminism, deadlock, invariant violation, or unsatisfiability, as reported in [4]: that paper presents a method for measuring the degree to which features interact in feature-oriented software development, extending the notion of simulation between transition systems to a similarity measure and lifting it to compute a behavioural interaction score in featured transition systems.

In [29] a more general definition of feature interaction, in terms of a feature that is developed and verified to be correct in isolation but found to behave differently when combined with other features, has been presented showing how such behavioral interactions could be detected as a violation of a bisimulation [22].

In the action-based way to analyse features to detect interactions, pairs of actions are typically defined to be conflicting in the domain description by an expert. An interaction arises when, as a result of the features application, two actions are executed, which were defined as conflicting in the domain description. In this setting a true 3-way interaction is possible only if a triple of conflicting actions exists in the domain, and no combination of two of them does. This is an expert evaluation and we can rely on the results of feature interaction detection contest at FIW2000 [17,18] where no such an example was found.

The state-based approach addressed in this paper analyses interacting features (and their actions) looking at their effect on a shared state. In this line there is the abstract semantics given in [24], where the state is a set of resources and the operations on the shared state are abstracted to consider only their access mode, namely *read* or *write* to some resources. Then features interact when one of them accesses in write mode a resource accessed also by the other, in any mode. The results presented in Sect. 4 can be extended to this semantics, which is indeed not far from the one presented here.

An alternative formalisation of the same problem can be obtained using contextual Petri Nets with inhibitor arcs [6,7]. Indeed, the read-only arcs of the contextual nets permit to model a (positive) condition which is not overwritten, and the inhibitor arcs permit to model the negative conditions. A result similar to ours – reduction of all triples of conflicting transitions to the conflict of a pair of them – exists for regular Petri Nets, while, to the best of our knowledge, nothing has been proved yet for the enriched ones.

In case of an interaction, some conflict resolution strategy should be applied, and several resolution techniques are proposed in the literature [3,8,10,14,15,24].

5.1 Different Computational Models and Non-functional Interactions

In order to discuss feature interactions, we have assumed that features are computational bricks that are independently developed and can freely be composed to achieve the desired functionality, possibly in an incremental development. In

```
1  class Stack {
2
3    boolean push(Object o) {
4  #ifdef LOCKING
5      Lock lock = lock ();
6      if(lock == null) {
7  #ifdef LOGGING
8        log("lock failed for: " + o);
9  #endif
10       return false;
11     }
12 #endif
13 #ifdef UNDO
14       rememberValue();
15 #endif
16       elementData[size++] = o;
17       /*...*/
18     }
19
20 #ifdef LOGGING
21     void log(String msg) { /*...*/ }
22 #endif
```

```
23 #ifdef UNDO
24   boolean undo() {
25 #ifdef LOCKING
26     Lock lock = lock ();
27     if(lock == null) {
28 #ifdef LOGGING
29       log("undo-lock failed");
30 #endif
31       return false;
32     }
33 #endif
34     restoreValue();
35     /*...*/
36 #ifdef LOGGING
37     log("undone.");
38 #endif
39   }
40
41   void rememberValue(); { /*...*/ }
42   void restoreValue(); { /*...*/ }
43 #endif
44 }
```

Fig. 1. This example appears in Sect. 9.1.1 "Higher-order Iteractions" of [3],

this setting, feature interactions resemble the classic notion of *race condition* in multithreaded environments.

There are other ways to look at the feature interaction problem, either because features are used to choose between alternative control flows or because their composition is subject to non-functional constraints posed by available resources. In both these views irreducible 3-way (or n-way) interactions can be observed.

Features and Conditional Compilation. Within the *Software Product Line* discipline, features are considered as units of functionality that may be present or not in different products of the same family. According to our approach, this can be obtained by composing or not certain features. An alternative common way to achieve variability in product lines is by referring to *presence conditions*, which tell which parts of a software component have to be included if a certain feature is present.

In Apel et al. [3], presence conditions are configuration tags that drive conditional compilation in a Java-like program including all the features. The example reported in [3], which we reproduce in Fig. 1, is presented as a case of interaction that occurs only if all three features are selected, and does not occur if only two are. Indeed, the usage of conditional compilation directives makes line 29 executable only if all the three features UNDO, LOGGING and LOCKING are selected. If line 29 contains an error (e.g. a null pointer access), this error occurs only in products that include all the three features, and not in products that include only two of them.

This case cannot be considered a 3-ways interaction according to our definition, since it is not an interaction error possibly occurring at run-time, but rather an error that is present in the code anyway and is activated only if at the

level of feature selection the proper feature combination is selected: in this way it is not different from a similar error inside a single #ifdef, that is activated just by selecting a single feature. For the detection of such an error it is not necessary to recur to behavioral analysis, but for example a static analysis of the "150% model", that is, the code obtained by switching all the features on, can be able to detect it.

Notice that the use of presence conditions reported in [3] is not amenable to incremental development, as the code of each feature is intertwined with that of the other features. A situation of this kind may occur also when delta-oriented programming or modelling [28] is adopted, in which a new feature may be defined as a set of changes to an existing program, or model. In this case, if a nested delta contains an error (similarly to the null pointer access in line 29 in the example above) this could be activated only if more features are included; again, a similar error could be detected with proper static analysis techniques run on the deltas. Notice also that similarly to this example, examples of interactions triggered by the selection of n features but not triggered by the selection of $n-1$ features can be easily built for any n, as well as examples of interactions triggered only by some particular set of selections of features.

Although presence conditions are of common use in product line development, we tend to believe that in the case of incremental development of safety-critical systems, even when configuration of different products is needed, the entangled appearance of resulting code, as the one in Fig. 1, makes verification and certification of software more difficult. A development approach in which features are separately implemented and verified and then composed appears to be more suitable for this class of systems, and our results indicate that only pairwise verification of possible interactions is needed.

Non-functional Interactions. Even if features are properly composed so that they do not produce undesired functional feature interactions, non functional ones can occur when features have to compete for the usage of shared (physical) resources, other than shared variables. Typical cases are memory space and computation time. The usage of such resources typically sums up, and if a maximum usage threshold is globally reached, unexpected behaviour may occur. Hence, we could have the case in which two out of three features do not exhaust available memory, but all three of them do, or the case of a real-time system in which running only two out of three features satisfy real-time requirements, while running all three does not.

In general, exceeding a resource usage threshold may be triggered by the selection of n features but not triggered by the selection of $n-1$ features for any given n, or may be triggered only by some particular set of selections of features.

6 Conclusions

We have addressed the problem of 3-way functional feature interactions, by giving a widely applicable definition framework within which we show that such

cases can be always reduced to 2-way interactions, hence reducing the complexity of automatic verification of incorrect interactions. We believe that other definition frameworks based on feature composition concepts share the same property. However, we have also pointed out at different definitions of feature interactions which admit "true" 3-way interactions, either because they define features through presence conditions scattered in different software artifacts, or because non functional feature interactions are considered.

Acknowledgements. This work has been partially supported by the Tuscany Region project POR FESR 2014-2020 SISTER and the H2020 Shift2rail project ASTRail.

References

1. Feature Interactions: The Next Generation (Dagstuhl Seminar 14281), Dagstuhl Reports, vol. 4, n.7. pp. 1–24. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2014)
2. Apel, S.: The new feature interaction challenge. In: Proceedings of the Eleventh International Workshop on Variability Modelling of Software-intensive Systems, VAMOS 2017, p. 1. ACM, New York (2017)
3. Apel, S., Batory, D., Kästner, C., Saake, G.: Feature-Oriented Software Product Lines: Concepts and Implementation. Springer, Heidelberg (2013). doi:10.1007/978-3-642-37521-7
4. Atlee, J.M., Fahrenberg, U., Legay, A.: Measuring behaviour interactions between product-line features. In: Gnesi, S., Plat, N. (eds.) 3rd IEEE/ACM FME Workshop on Formal Methods in Software Engineering, FormaliSE 2015, Florence, Italy, May 18, 2015, pp. 20–25. IEEE Computer Society (2015)
5. Back, R.-J., Kurki-Suonio, R.: Distributed cooperation with action systems. ACM Trans. Program. Lang. Syst. **10**(4), 513–554 (1988)
6. Baldan, P., Busi, N., Corradini, A., Pinna, G.M.: Domain and event structure semantics for Petri Nets with read and inhibitor arcs. Theor. Comput. Sci. **323**(1–3), 129–189 (2004)
7. Baldan, P., Corradini, A., Montanari, U.: Contextual Petri Nets, asymmetric event structures, and processes. Inf. Comput. **171**(1), 1–49 (2001)
8. Boström, M., Engstedt, M.: Feature interaction detection and resolution in the Delphi framework. In: [13], pp. 157–172, October 1995
9. Bruns, G.: Foundations for features. In: Reiff-Marganiec, S., Ryan, M. (eds.) Feature Interactions in Telecommunications and Software Systems VIII, pp. 3–11. IOS Press, Amsterdam (2005)
10. Buhr, R.J.A., Amyot, D., Elammari, M., Quesnel, D., Gray, T., Mankovski, S.: Feature-interaction visualization and resolution in an agent environment. In: [16], pp. 135–149, September 1998
11. Calder, M., Kolberg, M., Magill, E.H., Reiff-Marganiec, S.: Feature interaction: a critical review and considered forecast. Comput. Netw. **41**, 115–141 (2001)
12. Calder, M., Magill, E. (eds.): Feature Interactions in Telecommunications and Software Systems VI. IOS Press, Amsterdam (2000)
13. Cheng, K.E., Ohta, T. (eds.): Feature Interactions in Telecommunications Systems III. IOS Press, Amsterdam (1995)

14. Danelutto, M., Kilpatrick, P., Montangero, C., Semini, L.: Model checking support for conflict resolution in multiple non-functional concern management. In: Alexander, M., D'Ambra, P., Belloum, A., Bosilca, G., Cannataro, M., Danelutto, M., Martino, B., Gerndt, M., Jeannot, E., Namyst, R., Roman, J., Scott, S.L., Traff, J.L., Vallée, G., Weidendorfer, J. (eds.) Euro-Par 2011. LNCS, vol. 7155, pp. 128–138. Springer, Heidelberg (2012). doi:10.1007/978-3-642-29737-3_16
15. Dunlop, N., Indulska, J., Raymond, K.: Methods for conflict resolution in policy-based management systems. In: Enterprise Distributed Object Computing Conference, pp. 15–26. IEEE Computer Society (2002)
16. Kimbler, K., Bouma, L.G. (eds.): Feature Interactions in Telecommunications and Software Systems V. IOS Press, Amsterdam (1998)
17. Kolberg, M., Magill, E.H., Marples, D., Reiff, S.: Results of the second feature interaction contest. In: [12], pp. 311–325 (2000)
18. Kolberg, M., Magill, E.H., Marples, D., Reiff, S.: Second feature interaction contest. In: [12], pp. 293–310, May 2000
19. Li, H., Krishnamurthi, S., Fisler, K.: Verifying cross-cutting features as open systems. SIGSOFT Softw. Eng. Notes **27**(6), 89–98 (2002)
20. Marijan, D., Gotlieb, A., Sen, S., Hervieu, A.: Practical pairwise testing for software product lines. In: Proceedings of the 17th International Software Product Line Conference, SPLC 2013, pp. 227–235. ACM, New York (2013)
21. Medeiros, F., Kästner, C., Ribeiro, M., Gheyi, R., Apel, S.: A comparison of 10 sampling algorithms for configurable systems. In: Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, pp. 643–654. ACM, New York (2016)
22. Milner, R.: Communication and Concurrency. International Series in Computer Science. Prentice Hall, New York (1989)
23. Montangero, C., Reiff-Marganiec, S., Semini, L.: Logic-based conflict detection for distributed policies. Fundamenta Informaticae **89**(4), 511–538 (2008)
24. Montangero, C., Semini, L.: Detection and resolution of feature interactions, the early light way. Int. J. Adv. Syst. Measurements **8**(34), 210–220 (2015)
25. Nhlabatsi, A., Laney, R., Nuseibeh, B.: Feature interaction: the security threat from within software systems. Prog. Inform. **5**, 75–89 (2008)
26. Oster, S., Markert, F., Ritter, P.: Automated incremental pairwise testing of software product lines. In: Bosch, J., Lee, J. (eds.) SPLC 2010. LNCS, vol. 6287, pp. 196–210. Springer, Heidelberg (2010). doi:10.1007/978-3-642-15579-6_14
27. Reiff-Marganiec, S., Turner, K.J.: Feature interaction in policies. Comput. Networks **45**(5), 569–584 (2004)
28. Schaefer, I., Bettini, L., Bono, V., Damiani, F., Tanzarella, N.: Delta-oriented programming of software product lines. In: Bosch, J., Lee, J. (eds.) SPLC 2010. LNCS, vol. 6287, pp. 77–91. Springer, Heidelberg (2010). doi:10.1007/978-3-642-15579-6_6
29. Shaker, P., Atlee, J.M.: Behaviour interactions among product-line features. In: Gnesi, S., Fantechi, A., Heymans, P., Rubin, J., Czarnecki, K., Dhungana, D. (eds.) 18th International Software Product Line Conference, SPLC 2014, Florence, Italy, September 15–19, 2014, pp. 242–246. ACM (2014)
30. Turner, K.J., Reiff-Marganiec, S., Blair, L., Pang, J., Gray, T., Perry, P., Ireland, J.: Policy support for call control. Comput. Stand. Inter. **28**(6), 635–649 (2006)
31. Various Editors. Series of International Conferences on Feature Interactions in Software and Communication Systems (ICFI). IOS Press (1994–2009)
32. Zave, P.: An experiment in feature engineering. In: Morgan, C., McIver, A. (eds.) Programming Methodology, pp. 353–377. Springer, New York (2003). doi:10.1007/978-0-387-21798-7_17