Detection and Resolution of Feature Interactions, the Early Light Way

Carlo Montangero
Dipartimento di Informatica
Università di Pisa, Pisa, Italy
Email: monta@di.unipi.it

Laura Semini
Dipartimento di Informatica
Università di Pisa, Pisa, Italy
Email: semini@di.unipi.it

Abstract—**The feature interaction problem has been recognized as a general problem of software engineering, whenever one wants to reap the advantages of incremental development. In this context, a feature is a unit of change to be integrated in a new version of the system under development, and the problem is that new features may interact with others in unexpected ways. We introduce a common abstract model, to be built during early requirement analysis in a feature oriented development. The model is common, since all the features share it, and is an abstraction of the behavioural model retaining only what is needed to characterize the features with respect to their possible interactions. The basic constituents are the abstract resources that the features access in their operations, the access mode (read or write), and the reason of each access. Given the model, the interactions between the features are automatically detected, and the goal oriented characterization of the features provides the developers with valuable suggestions on how to qualify them as synergies or conflicts (good and bad interactions), and on how to resolve conflicts. We provide evidence of the feasibility of the approach with an extended example from the Smart Home domain. The main contribution is a lightweight state-based technique to support the developers in the early detection and resolution of the conflicts between features.**

Keywords–Feature interactions; State-based interaction detection; Conflict resolution.

I. Introduction

This paper extends the approach to the early detection and resolution of feature interactions we introduced in SOFTENG'15 [1]. The feature interaction problem has been recognized as a general problem of software engineering [2] [3] [4] [5], whenever an incremental development approach is taken. In this broader context, the term *feature*, originally used to identify a call processing capability in telecommunications systems, identifies a unit of change to be integrated in a new version of the system under development. The advantages of such an approach lay in the possibility of frequent deliveries and parallel development, in the *agile* spirit. The feature based development is now becoming more and more popular in new important software domains, like automotive and domotics. So, it is worthwhile to take a new look at the main problem with feature based development: a newly added feature may interact with the others in unexpected, most often undesirable, ways. Indeed, the combination of features may result in new behaviours, in general: the behaviours of the combined features may differ from those of the two features in isolation. This is not a negative fact, per se, since a new behaviour may be good, from an opportunistic point of view; however, most often the interaction is disruptive, as some requirements are no longer fulfilled. For instance, consider the following requirements, from the Smart Home domain:

Intruder alarm (IA)	Send an alarm when the main door is unlocked.
Main door opening (MDO)	Allow the occupants to unlock the main door by an interior switch.
Danger prevention (DP)	Unlock the main door when smoke is sensed.

Assuming a feature per requirement, it is easily seen that combining *Intruder alarm* and *Danger prevention* leads to an interaction, since the latter changes the state so that the former raises an alarm. However, an alarm in case of a fire is likely to be seen as a desirable side effect, so that we can live with such an interaction. Also, the combination of the first two features leads to an interaction: an alarm is raised, whenever the occupants decide to open the main door from inside. However, this is likely to be seen as an undesirable behaviour, since the occupants want to leave home quietly.

In general, the process of resolving conflicts in feature driven development has the same cyclic nature: look for interactions in the current specification, identify the conflicts, resolve them updating the specification, cycle until satisfaction.

Many techniques have been proposed to automate (parts of) this process. The search for interactions by manual inspection, as we did above, is obviously unfeasible in practice, due to the number of requirements in current practice. It is also the step with the greatest opportunity for automation. The other steps need human intervention since, at the current state of the art, they cannot be automatized. However, as discussed in Section XII, what is still lacking, in our opinion, is the ability to detect the interactions, identify the conflicts and resolve them by working on a simple model, as it may be available at the beginning of requirements analysis, before any major effort in the development of requirements.

We introduce a technique to support the detection and resolution of feature interactions in the early phases of requirements analysis. The approach is based on a common abstract model of the state of the system, which i) is simple enough to induce a definition of interaction which can be checked by a simple algorithm, and ii) can be modified, together with the feature specification, taking care only of few, essential facets of the system.

The model is *abstract*, since it is an abstraction of the behavioural model retaining only what is needed to characterize each feature with respect to the possible interactions:

the constituents of the model are *resources*, that is, pieces of the state of the system that the features access during their operations. To keep the model, and the analysis, simple, the operations on the resources are abstracted to consider only their access mode, namely *read* or *write*. This way, however, we do not loose in generality since the essential cause of an interaction is a pair of conflicting accesses to a shared resource. In this respect we were inspired by notion of conflict between build tasks introduced by the CBN *software build* model [6].

The work required to build the abstract model can be amortized in two ways. The shared state models can be defined in a reusable and generic manner so that, for a given domain, they can be exploited in many different development efforts, as it happens in Software Product Lines; moreover, the model can be taken as a skeleton to be fleshed out with details as requirements analysis proceeds.

The main concepts and ideas of the approach have been first introduced in [1]. Here, we formalize the proposed detection technique and extend it to deal with *indirect* interactions, i.e., those that depend on the relations among the resources. We also add a feature model to state relations between features such as priorities and mutual exclusions.

The next section summarizes the approach. Subsequent sections describe it in detail: Sections III, IV, and V deal with the definition of the abstract model. Sections VI and VII illustrate the automatic process to derive the interactions from the abstract model. Section VIII discusses synergies and conflicts, and Section IX illustrates some resolution techniques. Section X discusses briefly the complexity of the analysis. Section XI assesses the correctness and completeness of the approach, and Section XII discusses related work. Finally, we draw some conclusions and discuss future work.

In the paper, we use the Smart Home domain described in [7] as a running example. The features are intended to automate the control of a house, managing the home entertainments, providing surveillance and access control, regulating heating, air conditioning, lighting, etc.

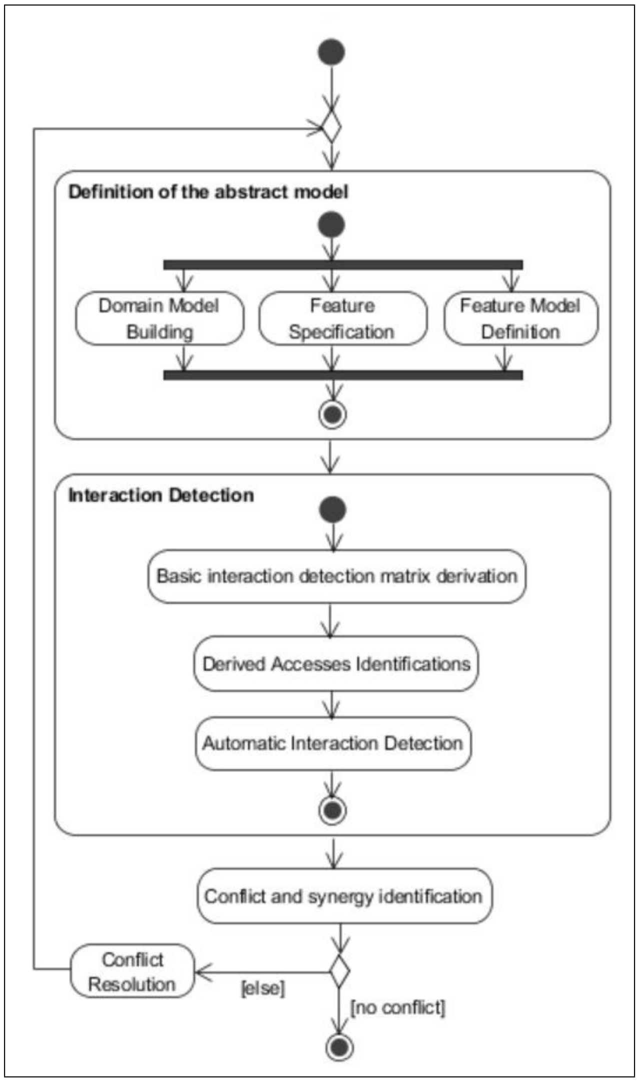
Definition of the abstract model
Domain Model Building
Feature Specification
Feature Model Definition
Interaction Detection
Basic interaction detection matrix derivation
Derived Accesses Identifications
Automatic Interaction Detection
Conflict and synergy identification
Conflict Resolution
[else]
[no conflict]

Figure 1. Activities of the lightweight approach.

II. Summary of the Approach

The lightweight approach to the detection and resolution of feature interaction requires the following activities:

1) Definition of the abstract model. This is obtained by the cooperation of three activities:
    - Domain model building, in terms of the resources accessed by the features.
    - Feature specification. Each feature is described in terms of: its goal; its accesses (r/w) to the resources in the domain model; the goal of each access.
    - Feature model definition, to state mutual exclusion and priority relations among the features.
2) Interaction detection is based on the construction and analysis of an *interaction detection matrix*, which is automatically built in two steps from the abstract model.
    - A basic interaction detection matrix is first derived, where the (direct) accesses mentioned in the features specification are considered.
    - The matrix is filled with indirect resource accesses, which take care of the relations between resources captured in the domain model.
    - Automatic interaction detection: the complete matrix is analyzed to single out possible interactions.
3) Conflict and synergy identification:
    - The interactions identified in the previous step are classified as conflicts or synergies: only conflicts will need to be dealt with in the resolution step.
4) Conflict resolution, that modify the abstract model using various strategies:
    - Restriction
    - Priority
    - Integration
    - Refinement

Figure 1 models the whole process. Note that, from the point of view of the development process, there is no constraint on how the abstract model is built: in other words, domain model building, feature specification, and feature model definition can be performed in sequence, as well as arm in arm as suggested in Figure 1. All the other activities are each dependent on the outcomes of the previous one in the list. We describe in detail each of them in the next sections.

III. Domain model building

The description of the domain is an integral part of the abstract model. Its purpose is to provide a definition of the accessible resources, i.e., of the shared state that the features access and modify, detailed enough to allow describing the features precisely. There are no special requirements on the notation to express the model. In this paper, we use UML2.0 class diagrams for their wide acceptance.

Given the Smart Home example, so far limited to the features in the Introduction, Figure 2 shows the class diagram modelling the domain. The shared state is made up of the states of the all the resources, which may structured, like Main Door, which owns a Lock.

The structure shown is not final, as new resources can be added by the analyst if he needs them, not only to introduce new features, but also to resolve conflicts, as it happens, for instance, with refinement (Section IX-4).

IV. Feature specification

We model a feature defining: its goal; the resources in the domain model it accesses (r/w); the reason for each access. The feature goal and the resource access reason are used during conflict identification (vs synergies) and resolution.

We introduce a template (Table I), which lists the feature name, its goal, and the involved resources, grouped in two sets (read or written) together with the reason for reading or writing each resource. To make references short, we provide an acronym to each feature. Each access to a resource is identified by its goal.

The three features introduced in the previous section are represented in Table II following the template.

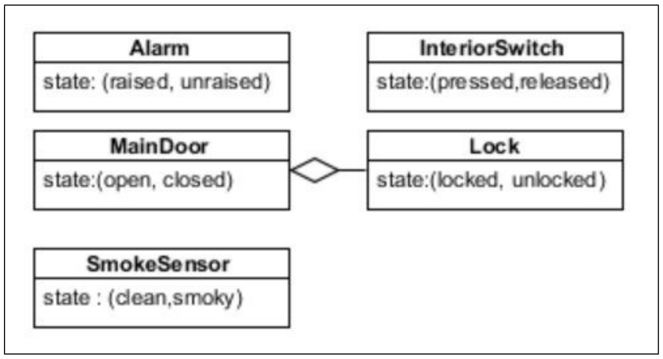

Figure 2. Smart Home Domain.

TABLE I. Feature specification template.

⟨**name**⟩ ⟨**acronym**⟩	**read**	**write**
⟨feature goal⟩	⟨resource⟩ $\hookrightarrow$ ⟨access reason⟩	⟨resource⟩ $\hookrightarrow$ ⟨access reason⟩

V. Feature Model Definition

A *Feature Model* is a compact representation of the constraints among the features that can be present in a system [8]. In our approach, the feature model records mutual exclusions and priorities between features: in the detection phase this structure is used to disregard the pairs that might interact but will not, since incompatibilities have already been solved by the introduced relations.

Definition (Feature Model) Let $\mathcal{F}$ be the set of features in the abstract model. A Feature Model is a pair

$$\mathcal{FM} = \langle \mathcal{P}, \mathcal{X} \rangle$$

where:

$\mathcal{P} \subseteq \mathcal{F} \times \mathcal{F}$ and $(F, F') \in \mathcal{P}$ when F has priority on F'

$\mathcal{X} \subseteq 2^{\mathcal{F}}$ and $X \in \mathcal{X}$ when the features in X are mutually exclusive.

There is no constraint among IA, DP, and MDO in the initial model, i.e., initially $\mathcal{FM} = \langle \emptyset, \emptyset \rangle$. We will fill it when adding new features, and after the resolution stage.

To focus on the essence of the approach, in the next section we deal with automatic interaction detection on the matrix including only the basic interactions and delay indirect interaction identification to the subsequent one.

VI. Automatic Interaction Detection

Our definition of feature interaction is based on the access mode (read or write) to the resources that make up the shared state of the system. The features access the resources in read mode to assess the state of the system, and in write mode to update it.

Any time two features F and F' access a resource, and at least one of the accesses updates it, there is an interaction: if F updates a resource which is read by F', the new value can change the behaviour of F', hence there is an interaction;

TABLE II. Feature specification: IA, MDO, DP.

Intruder Alarm (IA)	**read**	**write**
To raise an alarm when the main door is unlocked.	main door lock $\hookrightarrow$ To know when to raise an alarm	alarm $\hookrightarrow$ To raise the alarm

Main door opening (MDO)	**read**	**write**
To manually unlock the door.	InteriorSwitch $\hookrightarrow$ To receive the command	Lock $\hookrightarrow$ To unlock

Danger prevention (DP)	**read**	**write**
To automatically unlock the door in case of danger	SmokeSensor $\hookrightarrow$ To know when there is an alert	Lock $\hookrightarrow$ To unlock

TABLE III. INTERACTION DETECTION MATRIX FOR IA, MDO, DP. HERE AND LATER MDOOR STAYS FOR MAINDOOR.

$\mathcal{M}$	Lock	MDoor	Alarm	Interior Switch	Smoke Sensor
IA	r		w		
MDO	w			r	
DP	w				r

if F anf F' both modify the resource, the final value of the resource depends on the feature application order, and there is an interaction too. On the contrary, there is no interaction when F and F' both read a shared resource, since they do not interfere (still, F and F' can interfere if they access, and modify, another resource).

Definition (Interaction)

There is an interaction whenever two features are composed in the same system, and one of them accesses in write mode a resource accessed also by the other, in any mode.

Let us reconsider the features defined in the Introduction and the discussion in the previous section that led to detect some interactions. We can rephrase it in term of resource accesses. For instance, consider the main door lock: accessing it in read mode allows knowing its current state, that is, if the door is locked or unlocked; accessing it in write mode allows locking or unlocking the door. Both IA and MDO access the door lock, in read and write mode, respectively. By definition, we have an interaction. Similarly, also IA and DP interact, since they access the same resource in the same way.

To automate the interaction detection, an *interaction detection matrix* ($\mathcal{M}$) is built, with a row per feature and a column per resource. This is a sparse matrix where each entry is a set that contains information only if the feature in the row accesses the resource in the column, and is empty otherwise:

$$m \in \mathcal{M}_{F,R} \quad iff \quad F \text{ accesses } R \text{ in mode } m$$

As an example, Table III shows the interaction detection matrix for IA, MDO, and DP.

In the interaction detection matrix, it is possible to identify all the pairs of interacting features.

Statement F and F' interact on resource R if and only if

- $w \in \mathcal{M}_{F,R}$ and $\mathcal{M}_{F',R}$ is not empty.
- $(F, F') \notin \mathcal{FM}$, i.e., formally:
  - $(F, F') \notin \mathcal{P}$
  - $(F', F) \notin \mathcal{P}$
  - $\nexists X \in \mathcal{X}$ with $\{F, F'\} \subseteq X$

In other words, any pair of non empty entries in the same column with at least a *w* denotes an interaction of the features in the selected rows, provided the feature model does not prohibit their coexistence in the same system. In the example, we have that all pairs of features, (IA, MDO), (IA, DP), and (MDO, DP) interact on resource Lock, and $\mathcal{FM} = \langle \emptyset, \emptyset \rangle$.

Not all of these interactions are bad ones (conflicts): these have to be identiefied with a subsequent analysis (see Section VIII).

In this view, it is possible that a feature interacts with itself. An example is the following.

Silence at night (SAN)	At night, turn off the alarm after three minutes since it started beeping.

The feature specification is in Table IV and matrix is in Table V. The matrix shows an interaction, but it is evident that this is a desired behaviour, that is, a synergy. In other cases, the analysis may discover that the feature is ill-defined and has to be rewritten.

VII. INDIRECT INTERACTIONS IDENTIFICATION

There are other kinds of interactions, that we call *indirect* since they are due to accesses to different, but related, resources. Indeed, the domain model is not made of independent resources: they may be related in such a way that the access to one may entail an access to the other. We consider the following relations that cause *derived* accesses, inducing a state change in a resource as a consequence of a state change in another (related) one:

affects	when two resources are associated in such a way that a change in one affects the other;
composition	when a resource is a part of another one. This relation, due to its importance in structuring the domain, needs to be considered explicitely but can be reduced to instances of the previous one;
subclass/superclass	when a resource belongs to a sub/superclass of another one.

Then, we define how to complete the interaction detection matrix to take into account also these indirect interactions.

A. *Affects*

Consider the following example dealing with air conditioning (AC):

Natural AC (NAC)	If the air temperature in the room is above 27 degrees and the temperature

TABLE IV. FEATURE SPECIFICATION: SAN.

Silence at night (SAN)	**read**	**write**
To turn off the alarm at night	Alarm ↪ To know when it starts beeping	Alarm ↪ To turn it off

TABLE V. INTERACTION DETECTION MATRIX FOR SAN.

$\mathcal{M}$	Lock	MDoor	Alarm	Interior Switch	Smoke Sensor
SAN			r w		

TABLE VI. FEATURE SPECIFICATION: NAC AND ACS.

Natural AC (NAC)	**read**	**write**
To naturally change air.	Room Temp Sensor $\hookrightarrow$ To know if room has to be cooled Outside Temp Sens $\hookrightarrow$ To know if outside it is cold enought	Window $\hookrightarrow$ To open

AC switch-on (ACS))	**read**	**write**
To cool the room with AC.	Room Temp Sensor $\hookrightarrow$ To know if room has to be cooled	AirCond $\hookrightarrow$ To switch-on

TABLE VII. INTERACTION DETECTION MATRIX FOR IA AND DP2.

$\mathcal{M}$	Lock	MDoor	Alarm	Interior Switch	Smoke Sensor
IA	r		w		
DP2		w			r

outside is below 25, open the windows.

AC switch-on (ACS) If the air temperature in the room is above 27 degrees switch-on the air conditioner.

This is specified in Table VI.

The point here is that in both case there is an effect on the room air. Indeed, a change of state of the windows or the air conditioner affects the air in the room. We want this indirect interaction to be captured as a derived access.

Note that NAC and ACS are applied under the same condition. However, this read coincidence is not relevant for the interaction detection, since no interaction is caused by two read accesses to a resource.

B. Composition

Consider the main door, which is composed of a lock (Figure 3): a write access to the door may result in a write on the lock too, hence we add an *affects* relation between the two resources. For instance, consider a different version of DP, where, rather than simply unlock the door, the home automation system opens it, to facilitate escape and air change:

Danger prevention 2 (DP2) Open the main door when smoke is sensed.

Possibly, DP2 interferes with IA, since to open the door it may be needed to unlock it. However, with the basic matrix of the previous section, this interaction cannot be detected: the features access different resources, and no column in the matrix has more than an element (see Table VII).

In general, changing a resource may change also its parts and cause an interaction with the features accessing one of the

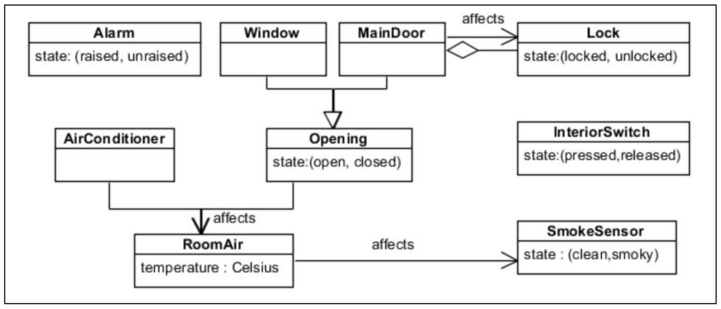

Figure 3. Smart Home Domain, extended.

parts. On the other side, often, the specifiers forget to mention these derived accesses (e.g., by stating explicitly: *Unlock the main door and* open it when smoke is sensed), and interactions are hardly identified. The domain structure can help in coming up with all interactions automatically.

C. Subclass or Superclass

Let us continue on the same example, with a third version of danger prevention:

Danger prevention 3 (DP3) Open all openings when smoke is sensed.

To cope with this new feature, the domain model needs to include also a new resource, *Opening*, superclass of Door and Window (Figure 3).

Here again, there is an interaction with IA, since a door is an opening (and the door is composed of a lock).

With inheritance we can have a derived access in both directions: a write on a Door may interfere with a feature accessing resource Opening, and hence we derive the write access from Door to Opening. Viceversa, a feature that specifies a change for Opening applies to both Door and Window.

However, there is no derived access between siblings: we must not derive an access to Window from an access to Door or vice-versa.

D. Extending the interaction detection matrix

From now on, to consider the extension just given, we interpret the definition of interaction given in Section VI to include derived accesses. The interaction detection matrix is completed accordingly.

We define a triple of write mode, resource, and relation

$$w^{relation}_{resource}$$

as entry of matrix $\mathcal{M}$, telling that `resource` is indirectly accessed in write mode, through `relation`.

We only derive the write accesses, and not the read accesses, to avoid filling the matrix with redundant information. Indeed, assume a write access on A and a read access on B, with A and B related with one of the aforementioned relations: once we derive a write on B, we can detect the interaction, and there is no need to derive a read on A.

TABLE VIII. EXTENDED INTERACTION DETECTION MATRIX FOR IA AND DP2.

$\mathcal{M}$	Lock	MDoor	Alarm	Interior Switch	Smoke Sensor
IA	r		w		
DP2	w^{comp}_{MDoor}	w			r

Definition ($\mathcal{M}_{F,R}$ extended) Matrix $\mathcal{M}$ is recursively built according to the following rule

$$\mathcal{M}_{F,R} \ni \begin{cases} m & \text{iff} \quad F \text{ accesses } R \text{ in mode } m \\ w^{aff}_{R'} & \text{iff} \quad (w \text{ or } w^{rel}_{res}) \in \mathcal{M}_{F,R'} \text{ and } R' \text{ affects } R \\ w^{sub}_{R'} & \text{iff} \quad (w \text{ or } w^{rel}_{res}) \in \mathcal{M}_{F,R'}, R \text{ is a subclass of } R', \text{ and } rel \neq sup \\ w^{sup}_{R'} & \text{iff} \quad (w \text{ or } w^{rel}_{res}) \in \mathcal{M}_{F,R'}, R \text{ is a superclass of } R', \text{ and } rel \neq sub \end{cases}$$

Specifically, we derive a write access on a resource R if there is a write on R' and the *affects* relation in the domain model tells that a change to R' can lead to a change to R. The derivation is unidirectional, respecting to the direction of the relation.

With inheritance, we derive accesses in both directions. However, derivation paths do not go up and down to avoid deriving an access to a window from an access to the main door (we require $rel \neq sup$). The constraint $rel \neq sub$ applies in the case of multiple inheritance.

Example An example of extended interaction detection matrix is in Table VIII, where w^{aff}_{MDoor} in $\mathcal{M}_{DP2,Lock}$ is added since in the model The Main Door affects the Lock. This triple permits to detect the indirect interaction between IA and DP2.

Example A larger example is Table XI. DP, DP2, and DP3 are alternative versions of danger prevention. They are mutually exclusive, since we want a system to include at most one of them:

$\mathcal{FM} = \langle \emptyset, \{\{\text{DP, DP2, DP3}\}\}\rangle$

This constraint simplifies the analysis since we can discard some pairs of features from the interference analysis. Namely, (DP,DP2), (DP, DP3) and (DP2,DP3).

Note also that the feature model permits to use a unique matrix accommodating various versions of the system instead of using a matrix per each version.

Remark The construction process of $\mathcal{M}$ is finite since a fix-point can always be reached. This is because: the domain model is finite; the matrix elements are sets (and not multisets).

TABLE IX. INTERACTING ACCESS TO LOCK.

–Interaction detected on Lock–			
Feature	Feature Goal	Mode	Access Reason
IA	To raise an alarm when the main door is unlocked.	r	To know if it has been unlocked
MDO	To manually unlock the door.	w	To unlock

TABLE X. INTERACTING DERIVED ACCESS TO LOCK.

–Interaction detected on Lock–			
Feature	Feature Goal	Mode	Access Reason
IA	To raise an alarm when the main door is unlocked.	r	To know if it has been unlocked
DP3	To open all openings in case of smoke.	w^{aff}_{MDoor}	$\Leftarrow$ ($w^{sub}_{Opening}$, MDoor) $\Leftarrow$ (w, Opening) $\Leftarrow$ To open.

VIII. CONFLICT AND SYNERGY IDENTIFICATION

For each detected interaction a summarizing table is built, with the information on the goals of the interacting features and on the reasons for the interacting accesses.

As an example, Table IX captures the interaction (IA, MDO) on the main door lock. Such a table will help the expert in the classification of the interaction and its resolution. At this point the expert can state whether the interaction is a synergy or a conflict, as clearly in this case, since we do not want the alarm to be sent when the opening is authorized.

Similar tables are built for the other pairs of interacting features. The expert can recognize that there is a synergy between *Intruder Alarm* and *Danger Prevention*, since sending the alarm is useful when some danger sensor is triggered. Also, the interaction between *Main Door Opening* and *Danger Prevention* is a synergy. Indeed, the two features pursue the same goal, that is to open the door.

In the case of a derived access, the summarizing table reconstructs the chain of the derived accesses, and then gives the reason for the base one, as in Table X.

IX. CONFLICT RESOLUTION

Once an interaction is recognized as a conflict in the analysis phase, we can take some actions to resolve it. In order to discuss possible resolution actions, we need to extend the working example. In addition to IA, MDO, and DP, we also consider a few more features, namely:

Air change (AC)	At 10:00 a.m. open the windows, at 10:30 a.m. close the windows.
Close window with rain (CW)	Close the windows when the rain sensor is triggered.
Video surveillance (VS)	Surveillance cameras are watched remotely via wifi.
Wifi switch-off (WSO)	Switch off the wifi at night.

The extended domain model is in Figure 4, and the specification of the new features is in Table XII.

TABLE XI. COMPLETE INTERACTION DETECTION MATRIX

$\mathcal{M}$	Lock	MDoor	Alarm	Interior Switch	Smoke Sensor	Window	Opening	AirCond	RoomAir	External Temp Sensor
IA	r		w							
MDO	w			r						
DP	w				r					
DP2	w^{aff}_{MDoor}	w			r		w^{sup}_{MDoor}			
DP3	w^{aff}_{MDoor}	$w^{sub}_{Opening}$			r		w			
NAC	w^{aff}_{MDoor}	$w^{sub}_{Opening}$				$w^{sub}_{Opening}$	w		r $w^{aff}_{Opening}$	r
ACS					$w^{aff}_{RoomAir}$			w	r $w^{aff}_{AirCond}$	

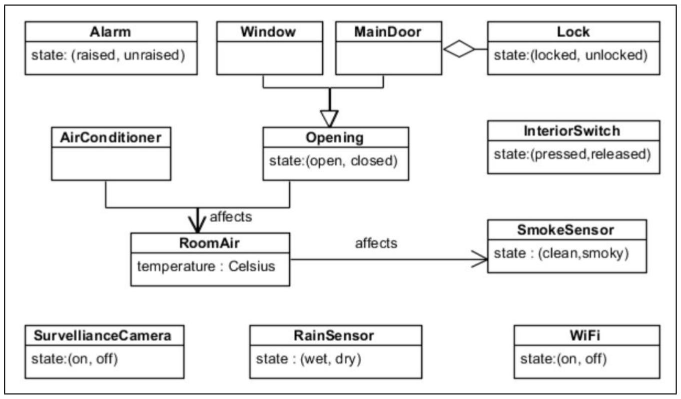

Figure 4. Smart Home Domain, the complete picture.

Various routes to resolution have been proposed in the literature (see [9] [10] [11] for interesting surveys):

1) Restriction: Avoid tout-court that the conflicting features are ever applied in the same system. This is the resolution strategy to be taken when the two features have incompatible goals. In other cases, it is an option the expert can choose. In the running example, we could prevent *Video surveillance (VS)* and *Wifi switch-off (WSO)* from being applied in the same house. We obtain restriction adding a mutual exclusion between the pair of conflicting features in the feature model.

2) Priority between the features: A weaker form of restriction is to guarantee that conflicting features are never applied at the same time. This behaviour can be obtained by defining priorities. Then, in the case two features are both enabled, only the one with higher priority is executed. In our example, priority can be likely used between *Air Change (AC)* and *Close window with rain (CW)*. Both features *write* on the resource *window*. In the case of rain at 10:00 a.m., we want the windows to be closed. The application of this strategy leads to adding a priority pair to the feature model.

3) Integration: According to this resolution strategy, the two interacting features are combined in a new one whose goal encompasses the goals of the two original ones.

VS and WSO can be integrated in a unique feature to switch off the wifi at night, and switch it on if an intruder is sensed, so that surveillance cameras can be watched from a remote machine.

TABLE XII. MORE SMART HOME FEATURES

Air Change (AC)	**read**	**write**
To ventilate the house		Window $\hookrightarrow$ To open/close

Close window with rain (CW)	**read**	**write**
To close the windows in case of rain	RainSensor $\hookrightarrow$ To know when to close	Window $\hookrightarrow$ To close

Video surveillance (VS)	**read**	**write**
To remotely control the house	VideoCamera $\hookrightarrow$ To read the recorded data Wifi $\hookrightarrow$ To access the camera	

Wifi switch-off (WSO)	**read**	**write**
To switch off the wifi when not used		Wifi $\hookrightarrow$ To switch-off

4) Refinement: In any approach based on a shared state, we can apply another resolution strategy, considering if it is possible to add a new resource and make the two conflicting accesses insist on two distinct resources. Since two features conflict only because they access, directly or indirectly the same resource, this refinement solves the problem, by definition. Think again of the conflict between *Intruder Alarm* and *Main door opening*. We might specify a new IA feature excluding the case where the door was unlocked using the interior switch. In some sense, we distinguish between the electrical and mechanical commands to the lock.

It is obvious that, after each resolution step, the features are to be checked again to detect if the changes have solved the conflicts without introducing new ones.

X. Implementation notes

The interaction detection matrix has two properties that are useful for the implementation of the analysis, i.e., to reduce the amount of time and space needed to search the pairs of interacting features: i) the matrix is sparse, and ii) the elaboration of each column (resource) is independent of the others, since it is only necessary to analyze pairs of cells in the same column. According to well known techniques, the matrix can be stored as a list of pairs $\langle resource, listOfAccesses \rangle$, where the second element represents the (sparse) column related to $resource$. Here, $listOfAccesses$ is the list of the non-null matrix entries, each represented as a pair $\langle feature, setOfAccessModes \rangle$.

The average cost of the analysis is then $\mathbf{O}(a^2 \times r)$, where a is the average number of accesses to the resources per feature, and r the number of resources. Note that the structure of the problem is such that it can be profitably attacked by parallel map-reduce, in case of very large matrixes, as it may be the case in real-life projects.

Note that, when creating this structure from the feature specification, there is no need to order the elements in (the lists representing) the columns, due to the independent elaboration of the columns. So, new items can be attached to the front of the list of the accessed resource (linear cost with r), and the matrix can be built in $\mathbf{O}(a \times r)$ in time (and space).

The need to sort the lists of accesses by feature arises only when it is requested to show the whole matrix to the engineers: by memoing the state (ordered or not) of each column, the cost can be made proportional to the number of updates to the matrix.

XI. Discussion

A discussion is needed on the soundness and completeness of our detection method with respect to existing ones. We restrict to design-time techniques, since we are interested in early detection. The most common way to define a feature interaction is based on behaviours [3]:

> A feature interaction occurs when the behavior of one feature is affected by the presence of another feature.

We consider behaviours too, but abstract from their details. Soundness is related with false positives: the rough detection based on the shared resources access model can indeed render false positives, e.g., synergies. These will have to be discarded during the subsequent analysis. However, also the approaches analyzing the concrete behaviour cannot automatically distinguish between conflicts and synergies and some human intervention is still needed to complete the analysis.

On the other side, the completeness problem can be stated as: is it possible that the behaviour of two features interfere even if they do not access, directly or indirectly, any shared resource? This can happen, for instance, if an *hidden resource* is not elicited and is not included in the model.

Often, there are *hidden classes* in a domain description. This is a well known problem in software engineering. In general, when analyzing and modeling a domain, some classes may be intrinsic to the problem, but never explicitly mentioned in the documentation. These classes cannot be found with the noun/verb analysis, and, to be exposed, must be discovered by the analyst.

Let us consider NAC and ACS. The interference between these features is detected since we included the air in the room that has to be cooled in the domain model, and derived an access of both features to this shared resource. If the hidden resource was not elicited, the interference was not found. However, do these feature interfere according to the behaviour based definition? The answer is no, the behaviour of each feature is not affected by the other one. Indeed, the conflict between the actions of opening the windows and switching on air conditioning can be stated only by an expert. So, the situation is similar: with both kind of approaches, the interference can detected thanks to some expert intervention.

Sometimes features interactions are defined in an even more abstract way:

> Features interactions are conflicts between the interests of the involved people.

We express the personal interests in the feature goals, and base the analysis on it. Hence, we are compliant with respect to this notion. Understanding if the persons involved have conflicting interests is a different problem.

Finally, we have restricted our analysis to pairs of features. One further aspect to consider, and this is again based on experience in feature interaction, is the question as to how many features are required to generate a conflict. In the community discussions have taken place around a topic called "three-way interaction". In the feature interaction detection contest at FIW2000 [8] this was an issue, and the community decided that there are two types of three-way interaction: those where there is already an interaction between one or more pairs of the three features and those where the interaction only exists if the triple is present. The latter were termed "true" three-way interactions. Nothing has been written about true three-way interaction, as only one, quite contrived, example of such an interaction has been found. We can hence consider as realistic the assumption that no "true" three-way interaction may occur.

Three-way interactions can occur among features implemented with directives to the preprocessor as done for instance in [12] but this is strictly related with the implementation technique. On the contrary, in our abstract setting, any interaction in a set of three (ore more) features is always caused by the interaction between two of them.

XII. Related Work

In [1], we first described the main concepts and ideas of an early and light analysis of features to detect interactions. Here, the approach is extended along various dimensions:

- We added the feature model definition in the first phase: the feature model describes relations between features such as priorities and mutual exclusions. It helps to accommodate in a unique model various version of a feature based system. At the same time, it is used to discard from the analysis those pairs of features that will never be applied in the same system and hence never interfere.

- We have considered derived accesses to the resources: An interaction can occur between features that are somehow related in the domain. The domain structure can help in coming up with all interactions automatically, instead of needing manual analysis by an expert.
- We have given a formal rule to fill the interaction detection matrix.
- We have discussed the complexity of the implementation.

A. *Programming features*

Bruns proposed to address the problem at the programming language level, by introducing features as first class objects [2]. Our view is that such an approach is worth pursuing, but needs be complemented by introducing features for features in the early stages of the development process, namely in requirements analysis.

B. *Requirements interaction*

Taxonomies of feature interaction causes have been presented in the literature [4] [13]. Among the possible causes, there are interactions between feature requirements. We address here a special case of the general problem of requirements interaction. A taxonomy of the field is offered in [14]. It is structured in four levels, and identifies 24 types of interaction collected in 17 categories. It assumes that the requirements specification is structured in system invariants, behavioural requirements, and external resources description. Their analysis is much finer grained than ours. Should the two analysis be performed in sequence, our own should prevent the appearance of some interaction types in the second one, like those of the non-determinism type.

Nakamura et al. proposed a lightweight algorithm to screen out some irrelevant feature combinations before the actual interaction detection, on the ground that the latter may be very expensive [15]. They first build a configuration matrix that represents concisely all possible feature combinations, and is therefore similar in scope to our interaction matrix. However, it is very different in contents, since it is derived from feature requirements specifications in terms of Use Case Maps, which give a very detailed behavioural description of the features. The automatic analysis of the matrix lends to three possible outcomes per pair of features: conflict, no interaction, or interaction prone. In our approach, the automatic analysis gives only two outcomes: no interaction or interaction prone, as one might expect, given the simpler model.

Another similar approach is Identifying Requirements Interactions using Semi-formal methods (IRIS) [7]. Both methods are of general application, and require the construction of a model of the software-to-be. In IRIS the model is given in terms of policies, but the formality is limited to prescribing a tabular/graphical structure to the model. Both methods leave large responsibility to the engineers in the analysis. However, larger effort is required, and larger discretion is left to them in IRIS: in our approach, interaction detection is automatized, and the engineer can focus on conflict identification and resolution. Finally, the IRIS model is much more detailed than ours, so that resolving the identified conflicts may entail much rework, while resolution in our case provides new hints to requirements specification. The last consideration applies as well to the two previous approaches.

C. *Design and run-time techniques*

As another example of the ubiquity of the feature interaction problem, Weiss et al. show how it appears also in web-services [16]. The approach to design-time conflict detection entails the construction of a goal model where interactions are first identified by inspection, and the subsequent analysis is then conducted on a process algebraic refined formal model. Also in this case, our model is more abstract, and the two techniques may be used synergically.

In a visionary paper, Huang foresees a runtime monitoring module that collects information on running compositions of web-services, and feeds it to an intelligent program that, in turn, detects and resolves conflicts [17].

Several run-time techniques to monitor the actual behaviour of the system and detect conflicts and possibly apply corrective actions, are reported in the literature, as surveyed in [11]: for instance, [18] tackle the problem with SIP based distributed VoIP services; in [19] policies are expressed as safety conditions in Interval Temporal Logic, and they can be checked at run-time by the simulation tool Tempura. These techniques should be seen as complementary to the design-time ones, like ours: the combined use of both approaches can provide the developers with very high confidence in the quality of their product, as suggested also by [10], which discusses the need for both static and dynamic conflict detection and resolution.

D. *Aspect oriented techniques*

A related topic is that of interactions between aspect-oriented scenarios. A scenario is an actual or expected execution trace of a system under development. The work described in [20] is similar to ours, in so far as they place it in the phase of requirements analysis, propose a lightweight semantic interpretation of model elements. The technique relies on a set of annotations for each aspect domain, together with a model of how annotations from different domains influence each other. The latter allows the automatic analysis of inter-domain interactions. It is likely that, if feature and aspect orientation are combined in the same development, the two techniques could be integrated.

E. *Formal methods*

A recent trend of design-time conflict detection exploits formal static analysis by theorem proving and model checking. The need for experimentation along this line has been recognized by Layouni et al. in [21], where they exploit the model checker Alloy [22] for automated conflict detection.

In [23], we presented a formal semantics for the APPEL policy language, which so far benefited only from an informal semantics. We also presented a novel method to reason about conflicts in APPEL policies based on the developed semantics and modal logic, and have touched on conflict resolution.

In [24], we show how to express APPEL [25] policies in UML state machines, and exploit the UMC [26] model checker to detect conflicts. In [27], we automate the translation from APPEL to the UMC input language, and address the discovery and handling of conflicts arising from deployment-within the same parallel application-of independently developed policies.

A feature interaction detection method close to model checking is presented in [28]: a model of the features is built using finite state automata, and the properties to be satisfied are expressed in the temporal logic Lustre. The environment of the feature is described in terms of the (logical) properties it guarantees, and a simulation of its behaviour is randomly generated by the Lutess tool; the advantage is that such an approach helps avoiding state explosion.

F. Abstract Interpretation

We remark a difference with the usual way of performing abstract interpretation [29], where the starting point is a detailed model, which is simplified, by abstracting away the information that is not needed for the intended analysis. An analysis by abstract interpretation defines a finite abstract set of values for the variables in a program, and an abstract version of the program defined on this abstract domain. Here, we abstract the actions to read or write but we only consider the variables (resources) name, and not their values, concrete or abstract.

G. Interactions affecting performance

Recently, work has been done on detecting and resolving interactions that, thought not disrupting the behaviour, impact on the overall performance of the system. The approach described in [30] is based on a simple black box model: interactions are detected using direct performance measurements designed according to few heuristics. It would be interesting to assess whether our technique may supplement advantageously the heuristics to the point of balancing the cost of the required domain model.

H. Best practices in requirements engineering (RE)

We refer to [31], since it emphasizes the identification of the product goals in the early phases of the analysis, and addresses explicitly the problem of conflicts in requirements. More precisely, the advocated RE process foresees the construction of a set of models of the system-to-be, each addressing a dimension of concerns. The most relevant one for our purposes is the Goal Model, which captures the system objectives in a structure of system goals and refines them to software requirements (SR). In this context,

- a goal is a prescriptive statement of intent that the system should satisfy through the cooperation of its agents,
- some of the agents are software ones, that is, they are part of the software-to-be, and
- a (software) requirement is a goal under the responsibility of a single (software) agent.

To see how our approach may fit into this scenario, it is enough to consider each feature as a software agent, whose requirement is given by the associated goal. Moreover, one can easily see that the software requirements in the running example may be the result of refining more general goals, like avoid that people are trapped into the house in case of fire.

The standard validation of the RE Goal Model includes a process to manage goal conflicts, which consists of four steps:

1) identify overlapping statements, i.e., those that refer to some inter-related phenomena;
2) detect conflicts among overlapping statements, possibly using some tool supported heuristics;
3) generate conflict solutions;
4) evaluate solutions and select the best ones.

In our approach, the first step is the construction of the interaction detection matrix, the second one is the process of pairwise feature interaction analysis described in Section VIII, and the third one what suggested in Section IX.

According to the agile nature of feature oriented software development, our approach to conflict detection entails also the construction of modelling items that belong to down stream activities in the RE process of [31], namely, building an Object and an Operation model.

The Object model provides a structural view of the *system-to-be*, showing how the concepts involved in the relevant phenomena are structured in terms of individual attributes and relationships with other concepts. As such, it is often conveniently represented by UML Class diagrams. Among the types of object to be considered in this model we find the already mentioned Agents, and the Entities, i.e., passive objects. The collection of the entities defines the state-space of the system, in terms of the object instances that may be present, each with its own internal state, as defined by the values of its attributes. Being at the RE level, there are no issues of information hiding, that is, a shared state is assumed.

The Operation model provides an operational rather than a declarative view of the system-to-be, unlike the previous models. This one is essential in Goal *operationalization*, that is, the process of mapping the requirements (leaf goals) to a set of operations ensuring them. Here, an operation is characterized by necessary and sufficient conditions for its application, which yields a state transition, in turn characterized by the operation post-condition.

In our approach, the collection of the resources and of the features constitutes the Object model. Any resource is an entity and any feature is an Agent. However, we depart from van Lamsweerde's process with respect to the Operation model, since we do not share his goal that the model contains enough information to allow its validation, that is, providing evidence that the operations of each agent ensure the goals. As we have shown, abstracting operations to their mode (read/write) and goal is sufficient to support feature conflict detection and resolution.

The integration of the support to interaction detection described here with the standard tools that support Requirements Engineering (RE), like DOORS, can be foreseen to occur in two modes, namely, loosely or tightly. In either cases, the information collected in the RE tool can be exploited to provide the engineer in chase of interactions with the structure of the tables of the features, i.e., names and definitions. In the case of loose coupling, these information need be exported in a dedicated tool: The engineer can then complete the tables adding the affected resources and the access modes, which are unlikely to be available in a standard RE tool. Once the analysis and resolution have been performed, the relevant information have to be fed back into the RE tool. A dedicated tool, equipped with interfaces supporting the most popular standard RE data interchange XML based standard, would support the interaction detection technique presented here for a wide range of RE tools. To get a tight coupling, one has to rest on the extension features the RE tool at hand offers: given that the computations needed to put our technique to work are essentially simple, there should be no major problem with most RE tools.

XIII. Conclusions

We present a state based approach to the early detection, analysis and resolution of interactions in feature oriented software development. Starting with a light model of the state that the features abstractly share, the main steps of our approach are the generation of an interaction matrix, the assessment of each interaction (conflict or synergy), and the update of the model to resolve conflicts. The abstraction is such that only the mode (read or write) of an access to the shared state is considered; each access is characterized by its contribution to the overall goal of the feature it pertains to.

We provide a proof of concept of how interactions can be detected automatically, as well as of how the developers can get support in their assessment of the interactions and resolution of the conflicts, looking at the well known Smart Home domain.

An interesting development will be to evaluate whether to formalize the goal model, and how, in view of a (partial) automatic support to the developers' analysis tasks. Another line of development of the approach would be to supplement each resource in the shared space with a standard access protocol, to prevent conflicting interactions. Inspiration in this direction may come from well established practices, like access control schemes and concurrency control.

Acknowledgments

The work was partly supported by the Italian MIUR PRIN project "Security Horizons".

References

[1] C. Montangero and L. Semini, "A lightweight approach to the early detection and resolution of feature interactions," in International Conference on Advances and Trends in Software Engineering (SOFTENG 2015). Barcelona, Spain: ThinkMind digital library, 2015, pp. 72–77.

[2] G. Bruns, "Foundations for Features," in Feature Interactions in Telecommunications and Software Systems VIII, S. Reiff-Marganiec and M. Ryan, Eds. IOS Press (Amsterdam), June 2005, pp. 3–11.

[3] S. Apel, J. M. Atlee, L. Baresi, and P. Zave, "Feature interactions: The next generation (dagstuhl seminar 14281)," vol. 4, no. 7. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2014, pp. 1–24, URL: drops.dagstuhl.de/opus/volltexte/2014/4783/ [retrieved: Nov, 2015].

[4] A. Nhlabatsi, R. Laney, and B. Nuseibeh, "Feature interaction: the security threat from within software systems," Progress in Informatics, no. 5, 2008, pp. 75–89.

[5] Various Editors, "Feature Interactions in Software and Communication Systems," international conference series.

[6] D. Coetzee, A. Bhaskar, and G. Necula, "A model and framework for reliable build systems," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2012-27 arxiv.org/pdf/1203.2704.pdf, Feb 2012, URL: www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-27.html [retrieved: Nov, 2015].

[7] M. Shehata, A. Eberlein, and A. Fapojuwo, "Using semi-formal methods for detecting interactions among smart homes policies," Science of Computer Programming, vol. 67, no. 2-3, 2007, pp. 125–161.

[8] P. Asirelli, M. ter Beek, A. Fantechi, and S. Gnesi, "A compositional framework to derive product line behavioural descriptions," in 5th Int. Symp. on Leveraging Applications of Formal Methods, Verification and Validation, ser. LNCS, vol. 7609. Heraklion, Crete: Springer, 2012, pp. 146–161.

[9] D. O. Keck and P. J. Kuehn, "The feature and service interaction problem in telecommunications systems: A survey," IEEE Transactions on Software Engineering, vol. 24, no. 10, Oct. 1998, pp. 779–796.

[10] N. Dunlop, J. Indulska, and K. Raymond, "Methods for conflict resolution in policy-based management systems," in Enterprise Distributed Object Computing Conf. IEEE Computer Society, 2002, pp. 15–26.

[11] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec, "Feature interaction: A critical review and considered forecast," Computer Networks, vol. 41, 2001, pp. 115–141.

[12] S. Apel, D. Batory, C. Kästner, and G. Saake, Feature-Oriented Software Product Lines: Concepts and Implementation. Springer, 2013.

[13] S. Reiff-Marganiec and K. J. Turner, "Feature interaction in policies," Comput. Networks, vol. 45, no. 5, 2004, pp. 569–584.

[14] M. Shehata, A. Eberlein, and A. Fapojuwo, "A taxonomy for identifying requirement interactions in software systems," Computer Networks, vol. 51, no. 2, 2007, pp. 398–425.

[15] M. Nakamura, T. Kikuno, J. Hassine, and L. Logrippo, "Feature interaction filtering with use case maps at requirements stage," in [32], May 2000, pp. 163–178.

[16] B. E. M. Weiss, A. Oreshkin, "Method for detecting functional feature interactions of web services," Journal of Computer Systems Science and Engineering, vol. 21, no. 4, 2006, pp. 273–284.

[17] Q. Zhao, J. Huang, X. Chen, and G. Huang, "Feature interaction problems in web-based service composition," in Feature Interactions in Software and Communication System X, S. Reiff-Marganiec and M. Nakamura, Eds. IOS Press, 2009, pp. 234–241.

[18] M. Kolberg and E. Magill, "Managing feature interactions between distributed sip call control services," Computer Network, vol. 51, no. 2, Feb. 2007, pp. 536–557.

[19] F. Siewe, A. Cau, and H. Zedan, "A compositional framework for access control policies enforcement," in Proceedings of the 2003 ACM workshop on Formal Methods in Security Engineering. NY, NY, USA: ACM Press, 2003, pp. 32–42.

[20] G. Mussbacher, J. Whittle, and D. Amyot, "Modeling and detecting semantic-based interactions in aspect-oriented scenarios," Requirements Engineering, vol. 15, 2010, pp. 197–214.

[21] A. Layouni, L. Logrippo, and K. Turner, "Conflict detection in call control using first-order logic model checking," in Proc. 9th Int. Conf. on Feature Interactions in Software and Communications Systems, L. du Bousquet and J.-L. Richier, Eds. France: IMAG Laboratory, University of Grenoble, 2007, pp. 77–92.

[22] Alloy Community, URL: alloy.mit.edu [retrieved: Nov, 2015].

[23] C. Montangero, S. Reiff-Marganiec, and L. Semini, "Logic-based conflict detection for distributed policies," Fundamenta Informaticae, vol. 89, no. 4, 2008, pp. 511–538.

[24] M. ter Beek, S. Gnesi, C. Montangero, and L. Semini, "Detecting policy conflicts by model checking uml state machines," in Feature Interactions in Software and Communication Systems X, International Conference on Feature Interactions in Software and Communication Systems, ICFI 2009, 11-12 June, 2009, Lisbon, Portugal. IOS Press, 2009, pp. 59–74.

[25] K. J. Turner, S. Reiff-Marganiec, L. Blair, J. Pang, T. Gray, P. Perry, and J. Ireland, "Policy support for call control," Computer Standards and Interfaces, vol. 28, no. 6, 2006, pp. 635–649.

[26] M. ter Beek, A. Fantechi, S. Gnesi, and F. Mazzanti, "A state/event-based model-checking approach for the analysis of abstract system properties," Sci. Comput. Program., vol. 76, no. 2, 2011, pp. 119–135.

[27] M. Danelutto, P. Kilpatrick, C. Montangero, and L. Semini, "Model checking support for conflict resolution in multiple non-functional concern management," in Euro-Par 2011 Parallel Processing Workshop Proc., ser. LNCS, vol. 7155. Bordeaux: Springer, 2012, pp. 128–138.

[28] L. du Bousquet, F. Ouabdesselam, J.-L. Richier, and NicolasZuanon, "Feature interaction detection using a synchronous approach and testing," Computer Networks, vol. 32, no. 4, 2000, pp. 419–431.

[29] P. Cousot, "Abstract interpretation based formal methods and future challenges," in Informatics - 10 Years Back. 10 Years Ahead, ser. Lecture Notes in Computer Science, R. Wilhelm, Ed., vol. 2000. Springer-Verlag, 2001, pp. 138–156.

[30] N. Siegmund, S. S. Kolesnikov, C. Kästner, S. Apel, D. S. Batory, M. Rosenmüller, and G. Saake, "Predicting performance via automated feature-interaction detection," in 34th Int. Conf. on Software Engineering, ICSE 2012, Zurich, Switzerland, 2012, pp. 167–177.

[31] A. van Lamsweerde, Requirements Engineering. John Wiley & Sons, Chichester, UK, 2009.

[32] M. Calder and E. Magill, Eds., Feature Interactions in Telecommunications and Software Systems VI. IOS Press (Amsterdam), May 2000.