A Lightweight Approach to the Early Detection and Resolution of Feature Interactions

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 the 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 each feature 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

The feature interaction problem has been recognized as a general problem of software engineering [1] [2] [3] [4], 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 gas/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 gas leak or 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 IV, 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

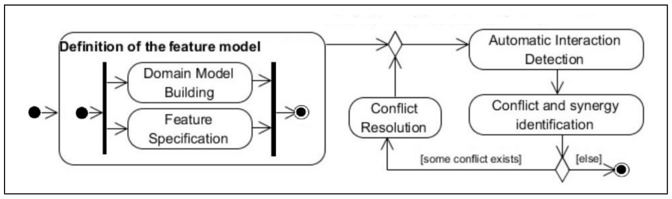

Definition of the feature model
Domain Model Building
Feature Specification
Automatic Interaction Detection
Conflict and synergy identification
Conflict Resolution
[some conflict exists]
[else]

Figure 1. Activities of the lightweight approach.

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 [5].

The work required to build the abstract feature 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 fleshed-out as requirements analysis proceeds.

In the following, we use the Smart Home domain described in [6] 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.

The next section describes the approach. Section III assess the correctness and completeness of the approach, and Section IV discusses related work. Finally, we draw some conclusions and discuss future work.

II. Approach

The lightweight approach to the detection and resolution of feature interaction consists in the activities presented in Figure 1 and elaborated in the next subsections.

Note that, from the point of view of the development process, there is no constraint on how the abstract feature model is built: in other words, domain model building and feature specification 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.

A. Domain model building

The description of the domain is an integral part of the abstract feature 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.

Figure 2 shows the class diagram of part of the Smart Home design domain. The shared state is made up of the states of the all the resources, which may structured, like MainDoor, 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 with refinement (Section II-E5).

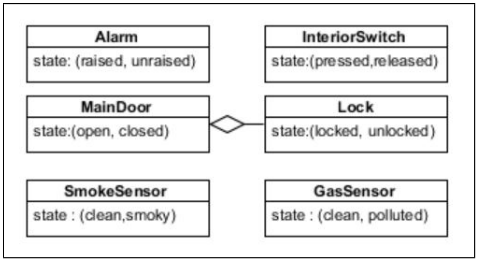


Figure 2. Smart Home Domain.

TABLE I. Feature specification template.

⟨name⟩ ⟨acronym⟩	read	write
⟨feature goal⟩	⟨label⟩ ⟨resource⟩ ⟨access reason⟩	⟨label⟩ ⟨resource⟩ ⟨access reason⟩

B. Feature specification

We model a feature defining: its goal; the resources in the domain model it accesses (r/w); the reason for each access. To make references short, we provide an acronym to each feature, and an integer label to each resource access. 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. The three features introduced in the previous section are represented in Table II.

Note that the accesses are numbered only for reference: no sequencing is implied, as the order of the accesses is abstracted away, as part of the simplicity of the model.

C. 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. By definition,

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

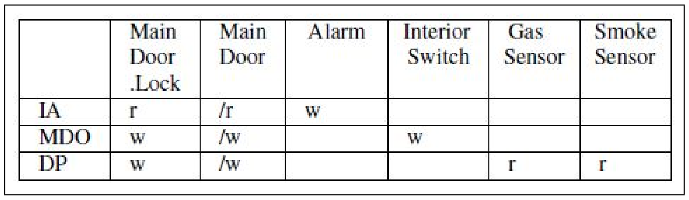
Let us reconsider the features defined above and the discussion in the previous section that led to detect some

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

Intruder Alarm (IA)	**read**	**write**
To raise an alarm when the main door is unlocked.	(1) main door lock To know when to raise an alarm	(2) alarm To raise the alarm

Main door opening (MDO)	**read**	**write**
To manually unlock the door.	(1) InteriorSwitch To receive the command	(2) MainDoor.Lock To unlock

Danger prevention (DP)	**read**	**write**
To automatically unlock the door in case of danger	(1) GasSensor (2) SmokeSensor To know when there is an alert	(3) MainDoor.Lock To unlock



	Main Door .Lock	Main Door	Alarm	Interior Switch	Gas Sensor	Smoke Sensor
IA	r	/r	w			
MDO	w	/w		w		
DP	w	/w			r	r

Figure 3. Interaction detection matrix.

TABLE III. INTERACTING ACCESS TO MAINDOOR.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.	r	To unlock

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.

We are now ready to see how interaction detection can be automated: we build a matrix with a row per feature and a column per resource, and put *r* (*w*) in cell (F_i, R_i) when F_i accesses R_i in *read* (*write*) mode. The matrix is completed to take into account the composited resources of the domain. Indeed, potentially, the access to the field of a resource is an access to the resource itself and vice versa. We put */r* or */w* in a cell when the design domain entails that potentially there is a *derived* resource access. As an example, Figure 3 shows the matrix for IA, MDO, and DP.

In the interaction detection matrix, it is possible to identify all the pairs of interacting features: any pair of non empty entries in the same column with at least a *w* (or */w*) denote an interaction of the features in the selected rows. In the example, from the first column, we have (IA, MDO), (IA, DP), and (MDO, DP).

As an example where derived accesses are essential, let us assume a different version of DP: open the main door when gas/smoke is sensed. In order to find the interaction between the write on the main door (to open it) and the read on the door lock of feature IA, we need the derived read of IA on the main door.

The superclass relation (an example is given in the extended case study in Figure 4) is dealt with in a similar way: the access to a superclass is also an access to its subclasses.

D. 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 III 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 if the interaction is a conflict, as clearly in this case, since we do not want the alarm to be sent when the opening is authorized, or a synergy. Instead, sending the alarm is useful when some danger sensor is triggered. Hence, there is a synergy between *Intruder Alarm* and *Danger Prevention*. Finally, 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.

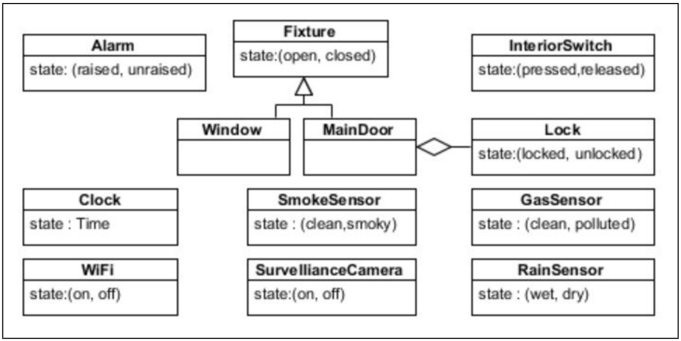


Figure 4. Extended Smart Home Domain.

E. 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 them, we need to extend the working example. In addition to IA, MDO, DP we consider also 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 IV.

Various routes to resolution have been proposed in the literature (see [7] [8] [9] 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.

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 do not want the windows to be open.

3) Sequencing: This technique applies when the conflict is caused by a bad order of application of features that are triggered at the same time. In this case, to solve the conflict it is sufficient to force the features to be applied in the correct order.

4) 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.

5) 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 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, features are to be checked again to detect if the changes have solved the conflicts without introducing new ones.

The first three strategies do not change the features, but extend the model adding relations between them. A new structure is introduced that records mutual exclusions, priorities, and sequencing between features, as done, e.g., in [10]: This structure is used in the detection phase to disregard the pairs that might interact but will not, since incompatibilities have already been solved by the introduced relations.

III. 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 [2]:

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

Soundness depends on the expert competence: 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 any shared resource? Consider the following example dealing with air conditioning (AC):

Natural AC (NAC)	If the room temperature is above 27 degrees and the temperature outside is below 25, open the windows.
AC switch-on (ACS)	If the room temperature is above 27 degrees switch-on the air conditioner.

These two features read the same resource, and act differently under identical conditions, but they do not interfere according to our definition. Do they 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. Similarly, in our case, a relation between open windows and air conditioning can be recognized during domain description, permitting the conflict to be detected.

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.

TABLE IV. MORE SMART HOME FEATURES

Air Change (AC)	**read**	**write**
To ventilate the house	(1) Clock To know when to open/close	(2) Window To open/close
Close window with rain (CW)	**read**	**write**
To close windows in case of rain	(1) RainSensor To know when to close	(2) Window To close
Video surveillance (VS)	**read**	**write**
To remotely control the house	(1) VideoCamera To read the recorded data (2) Wifi To access the camera	
Wifi switch-off (WSO)	**read**	**write**
To switch off the wifi when not used	(1) Clock To know when to switch-off	(2) Wifi To switch-off

IV. RELATED WORK

A. Programming features

Bruns proposed to address the problem at the programming language level, by introducing features as first class objects [1]. 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 [3] [11]. 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 [12]. 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 [13]. 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) [6]. 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 [14]. 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 [15].

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 [9]: for instance, [16] tackle the problem with SIP based distributed VoIP services; in [17] 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 [8], 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 [18] 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 the current advances in formal static analysis by theorem proving and model checking. The need for experimentation along this line has been recognized by Layouni et al. in [19], where they exploit the model checker Alloy [20] for automated conflict detection. In [21], we show how to express APPEL [22] policies in UML state machines, and exploit the UMC [23] model checker to detect conflicts. In [24], 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 management policies.

A feature interaction detection method close to model checking is presented in [25]: 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 [26], where the starting point is a detailed model, which is simplified, by abstracting away the information that is not needed for the intended analysis. What is proposed here is to start with an abstract view in terms of feature goals and resource accesses, and to perform conflict analysis and resolution up-front.

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 [27] 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.

V. 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] 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.

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

[3] 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.

[4] V. Editors, "Feature Interactions in Software and Communication Systems," ser. Int. Conference series.

[5] 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: http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-27.html [retrieved: Feb, 2015].

[6] 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.

[7] 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.

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

[9] 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.

[10] P. Asirelli, M. H. 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.

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

[12] 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.

[13] M. Nakamura, T. Kikuno, J. Hassine, and L. Logrippo, "Feature interaction filtering with use case maps at requirements stage," in [28], May 2000, pp. 163–178.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] Alloy Community, URL: alloy.mit.edu/community/ [retrieved: Feb, 2015].

[21] 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.

[22] 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.

[23] 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.

[24] 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, M. A. et al., Ed., vol. 7155. Bordeaux: Springer, 2012, pp. 128–138.

[25] 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.

[26] P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," in Proc. 4^{th} ACM Symp. Principles of Programming Languages, 1977, pp. 238–252.

[27] 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 International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland, 2012, pp. 167–177.

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