Ahmed F. Layouni, Luigi Logrippo and Kenneth J. Turner. Conflict Detection in Call Control using First-Order Logic Model Checking, in Lydie du Bousquet and Jean-Luc Richier (eds.), Proc. 9th Int. Conf. on Feature Interactions in Software and Communications Systems, pp. 66-82, IOS Press, Amsterdam, May 2008.

Conflict Detection in Call Control Using First-Order Logic Model Checking

Ahmed F. Layouni[1], Luigi Logrippo[1], Kenneth J. Turner[2]

[1]Université du Québec en Outaouais, Département d'informatique et ingénierie,
Gatineau, QC, Canada J8X 3X7 *(Email: laya01 | luigi @uqo.ca)*
[2]University of Stirling, Department of Computing Science and Mathematics
Stirling FK9 4LA, Scotland, UK *(Email: kjt@cs.stir.ac.uk)*

Abstract. Feature interaction detection methods, whether online or offline, depend on previous knowledge of conflicts between the actions executed by the features. This knowledge is usually assumed to be given in the application domain. A method is proposed for identifying potential conflicts in call control actions, based on analysis of their pre/post-conditions. First of all, pre/post-conditions for call processing actions are defined. Then, conflicts among the pre/post-conditions are defined. Finally, action conflicts are identified as a result of these conflicts. These cover several possibilities where the actions could be simultaneous or sequential. A first-order logic model-checking tool is used for automated conflict detection. As a case study, the APPEL call control language is used to illustrate the approach, with the Alloy tool serving as the model checker for automated conflict detection. This case study focuses on pre/post-conditions describing call control state and media state. The results of the method are evaluated by a domain expert with pragmatic understanding of the system's behavior. The method, although computationally expensive, is fairly general and can be used to study conflicts in other domains.

Keywords: Call control, conflict detection, feature interaction, policy, APPEL, Alloy, logic model checking.

1 Introduction

1.1 Features and Policies for Call Control

Feature interactions have been discussed with respect to many types of systems, although a good part of the literature has concentrated on call processing systems. A survey of the literature on the subject can be found in [2].

Feature interaction is a complex phenomenon and can be analyzed from different points of view. Much research in the area has emphasized the behavioral aspect of the phenomenon. In this perspective, feature interactions are often seen as the result of complex behavior interleaving for the state machines that represent the features. In two feature interaction contests [10,12] the contestants were given what essentially were state machines for features. These had to be composed, and their composition had to be modeled and evaluated. The goal was to come up with behavioral traces

showing that, for example, one feature was not allowed to run to completion due to the intervention of another feature.

In the world of VoIP, users are allowed to program their own features. However, most users do not program them from scratch using VoIP facilities directly. Rather, each VoIP system offers a set of basic features that can be combined by users and enterprises, by using specifically designed languages, to implement different policies. CPL (Call Processing Language [15]) is a well-known, early embodiment of this idea. Other policy languages with different purposes are LESS [22,23] and APPEL [20, 21].

In these approaches, users can specify policies such as: 'if a call arrives from Alice during work hours, treat it as urgent', 'calls to Bob should be tried at all addresses where Bob normally works'. The familiar <trigger, condition, actions> paradigm is at the heart of these systems, and we conjecture that it will continue to be used. This paradigm is essentially identical to the ECA, or <event, condition, actions> paradigm that has been applied extensively in areas such as reactive databases, agent systems, access control systems and the semantic web.

Generally speaking, a rule is enabled when its trigger occurs and its condition holds. Note the difference between trigger and condition. The trigger can be an external or internal event. A trigger can convey parameters for use in conditions and actions. Conditions can check database or 'context' information, such as the time of day or the role of the user in an enterprise ontology. Application of the rule leads to one or more actions. This apparently simple paradigm allows many variations, and is a good match to the many requirements of call control. A policy can expand in a number of such rules.

By means of policies and rules one can define the correspondent of traditional features, though policies can be higher-level, user-oriented and more declarative.

Several actions can be proposed simultaneously, for example when one rule defines multiple actions or multiple rules are activated by the same trigger. When this happens, the different actions can direct the system to do incompatible things. Actions may also set conditions that can block other actions that should follow. Conflicts between actions imply potential conflicts between the policies that invoke the actions and are the main manifestation of *feature interactions* in policy systems. In this paper the terms *conflict* and *incompatibility* will be synonyms, and conflicts and incompatibilities will be seen as the consequences of logical *inconsistencies*.

In policy systems there are resolution methods to ensure that only one action for each event is executed. For example, this is the situation for firewalls. Here, the rule file is typically scanned top-down and only the first applicable rule is used. This leads to just one action that accepts or rejects the proposed access. Some policy languages allow the user to include meta-rules for resolving cases where several actions may become simultaneously enabled. Often these meta-rules are based on priorities. The situation is complicated by the fact that for certain events, several actions may be needed.

Nonetheless, for the validation of a policy set, all rules and actions that can become enabled for a given trigger and condition should be examined without considering resolution methods. Indeed, several cases of interest can be found in this way. For example, an important policy might be 'shadowed' by a more general but contradictory policy, or a specific case might have been added in contradiction to an important general policy. This can happen because users in these systems may be

allowed to add and delete rules when they see the need for them. When they do this, they may not have a global view of all the consequences of the changes. Such situations could lead to unwanted system behavior, even though it may be technically correct. Users should be notified with a request, and possibly suggestions, for resolution.

1.2 Related Work

Several authors have suggested that many undesirable feature interactions can be understood as the result of inconsistency in specifications. Perhaps the earliest and clearest statements in this sense can be found in [3,8], where feature interactions are modeled as inconsistencies among temporal logic specifications. According to this work, features A and B conflict if and only if a program realizing their joint specification A^B does not exist. The detection method uses the model-checker Cospan. A similar view is given a theoretical justification in [1]. But already the first classical paper on this subject [2] lists 'conflicting assumptions' as one of the main causes of feature interaction. Among others, [5, 9, 13] are based on the idea that feature interactions are the result of conflicting actions becoming enabled. But how to tell that actions can conflict? [22, 23] push the analysis to higher granularity by considering the pre/post-conditions of actions. For example, two actions having incompatible post-conditions can cause a feature interaction if they are simultaneously enabled, or two actions for which the first falsifies the pre-condition of the second can cause a feature interaction if they are enabled one after the other. Conflicts of pre/post-conditions in systems of ECA rules have also been studied in [18].

We extend the conflict identification method of [22, 23] to the language APPEL [20, 21], as well we refine some of the definitions used in these papers. We automate the conflict detection method using the first-order formal language Alloy [11]. The associated Alloy tool is used to identify the conflicts.

A pragmatic approach to handling conflicts in APPEL is described in [19]. This work provides run-time support assuming that the conflicts have already been identified in some independent way. Another very recent contribution for the same language [16] provides a denotational semantics framework for APPEL, as well as a method to address feature interaction, but again assuming that conflicts between actions have already been identified. The method described in this paper can be used in conjunction with the techniques proposed in these two other papers to provide the information that they need, concerning the conflicts existing between specific actions. This method is a contribution towards a formal semantics for APPEL, as well as to feature interaction handling in APPEL.

In a related paper [4], a technique has been developed for filtering conflicts in the same APPEL language. This other approach is founded on the intuitive notion that actions may conflict if they share a common effect. In contrast, the work reported here has a higher degree of precision. Pre/post-conditions are considered, as well as the ordering of actions. This leads to a formal model that allows semantically-based inferences to be drawn about the compatibility of actions. Still, because of our level of precision, the high-level analysis possible in [4] would be difficult with our method, as well several aspects that can be considered with that method would be difficult to

consider with ours. For the time being, we must consider these two methods as both useful and complementary. Future research will have to deal with the problem of reconciling and integrating them.

2 Ordering and conflicts between actions

In this method, the mutual consistency of actions is determined on the basis of their pre/post-conditions. We consider a system state to be characterized by a set of variables and their values. Pre/postconditions are predicates that describe these values. The pre-condition of an action describes the state(s) in which the system must be in order for the action to execute. The post-condition of an action describes the state(s) that can result from its execution. We shall see below that pre/postconditions can be consistent or inconsistent, leading to mutual consistency or inconsistency of states

The following timing relationships can apply between actions:

- *simultaneous* execution: one action starts executing at a time when the other action has not completed.
- *sequential* execution: one action starts executing after the other action has completed, i.e. one action strictly *precedes* another.

If two actions start from or lead to mutually inconsistent system states, they are incompatible and should not be simultaneously executed. Even the case in which such actions are sequentially executed could be suspect, because the second action contradicts the results of the first (although this is normal in the evolution of a system). If an action establishes a post-condition which contradicts the pre-condition of another action, then the second action cannot immediately follow the first.

More in detail, the following relations are of interest between the pre/post-conditions of two actions A and B (this is not meant to be an exhaustive list):

1. Relationship between the pre-conditions of A and the pre-conditions of B:
   (a) The conjunction of the pre -conditions of these two actions is always true. The two actions can thus be executed simultaneously always. This is perhaps a rare situation.
   (b) The conjunction is satisfiable. In certain system states, A and B can both be executed.
   (c) The pre-conditions of the two actions are not simultaneously satisfiable. There are no system states for which A and B can be executed simultaneously. For example, they both might require the same device or they can be executed only in different connection states.
2. Relationship between the post-conditions of A and the pre-conditions of B (or vice versa). The cases are similar
   (a) The conjunction is always true: then the second action can always start after the first.
   (b) The post-conditions of A are simultaneously satisfiable with the pre-conditions of B. B can follow A in the case of simultaneous truth. (A more general case of these two situations is the case in which the post-condition of A implies the pre-condition of B.)
   (c) The post-condition of A is not simultaneously satisfiable with the pre-condition of B. In other words, B cannot follow A or A ‘disables’ B. For

example, A might free a device that B needs to find reserved, or A might leave the system in a connection state that is different from the one B requires.

3. Relationship between the post-conditions of A and B:

   (a) Simultaneous truth: no problem for concurrent execution.

   (b) The post-conditions of A and B are simultaneously satisfiable. This means that the results of A and B can be compatible.

   (c) The post-conditions of A and B are not simultaneously satisfiable. This means that the results of A and B are incompatible in principle. For example, one of them disconnects the call while the other continues it. Simultaneously executing the two actions would leave the system in an inconsistent, i.e. impossible state.

Doing a thorough analysis of all the cases above would be rather complicated, and to our knowledge this has never been done for realistic call control systems.

In this work, we are interested about a partial analysis of conflicts, and we identify three situations of conflict between actions (Figure 1):

- *concurrency conflicts*: two actions have inconsistent pre-conditions, and thus cannot be executed in the same system state
- *disabling conflicts*: an action leaves the system in a state where a second action cannot be executed
- *results conflicts*: two actions would leave the system in an inconsistent (impossible) state, and thus cannot be executed simultaneously.

Further, the two aspects of pre/post-conditions to be considered are the connection state and the media state.

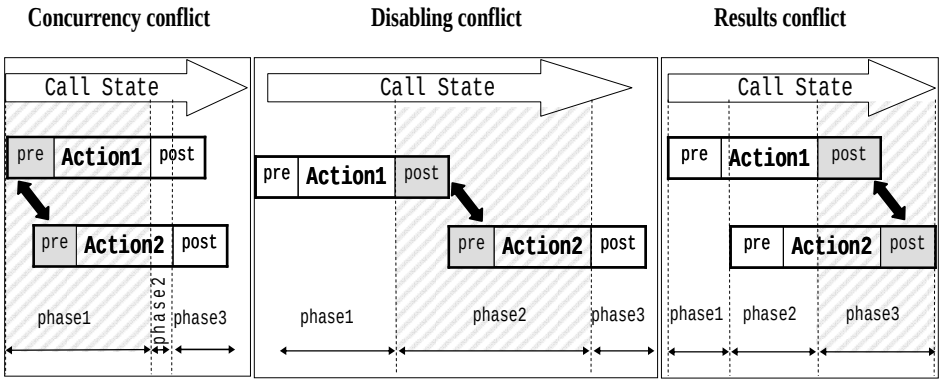

Figure 1. Three types of conflicts

Conflicts among pre/post-conditions of more than two actions are also possible. However this kind of analysis is rarely performed because it becomes complex and very few concrete examples (where three actions can be in conflict without any two of them being in conflict) are known. In addition, our case study will be on APPEL, and run-time conflict handling for APPEL is designed so that only pairwise combinations of actions need be considered.

3 The APPEL Policy Language

APPEL (ACCENT Project Policy Environment/Language) is a general-purpose language for expressing policies. The language is defined in [20], and its use for call control is described in [21]. APPEL conforms to the ECA model for policy rules. APPEL is supported by a policy system that interfaces to some system under control (e.g. a SIP server). When a trigger is received (e.g. there is an incoming call or a new party is being added to the call), the policy server retrieves all policies that apply. These are typically policies of the caller and the callee, but higher-level policies may also be retrieved (e.g. of the user's organizations). Policies are then checked for applicability. Apart from explicit policy conditions, other factors that determine applicability include the profile of a policy and its period of validity. The result is a set of actions. Triggers, conditions and actions may all be composite. Triggers and conditions may be combined by logical operators, and actions may be conditional, sequential or concurrent.

Although APPEL resembles a number of other policy languages, it differs in a number of important respects. It was specifically oriented towards the need for call control, as other approaches do not relate well to this application. For example, the Ponder policy language [6] assumes that the subject and target of a policy can be identified. However, in call control and other applications these concepts do not have clear interpretation.

APPEL was designed so that ordinary end users can formulate policies, unlike other languages that require a high degree of technical expertise. Since APPEL is XML-based, policies cannot be defined directly by a non-technical user. APPEL is therefore supported by a user-friendly policy wizard that allows creation and editing of policies using near-natural language.

Although APPEL was originally developed for call control, it is of wider applicability. For example, it has also been used for policy-based management of home care and sensor networks. This wide range of applications is possible because APPEL has a core language that is supplemented by domain-specific extensions. This is reflected in the language schemas and also in the ontologies that define domain vocabularies.

APPEL was designed with conflict handling in mind. As described in [19], the actions resulting from a trigger are filtered for compatibility. Special resolution policies are used to detect and to resolve conflicts. These policies resemble regular policies, but the trigger of a resolution policy is the action of a regular policy. Since resolutions are defined rather than being built into the policy system, there is considerable flexibility in how conflicts are handled. Generic resolutions choose among the conflicting actions, while specific resolutions propose domain-specific actions (that may differ from the conflicting ones). Although the approach supports automated run-time resolution of conflicts, it relies on resolution policies having been already defined. That is, as mentioned, the approach is dependent on already knowing what the conflicts are. In previous work, conflicts were determined manually – a tedious and error-prone task. The new work reported here provides a systematic, automated and semantically-based way of discovering conflicts that can then be used to define resolution policies.

4 APPEL Actions and Their Conflicts

4.1 APPEL Actions

Although our approach could be used with APPEL in other domains, for concreteness and familiarity we use call control as the application domain. The call control actions in APPEL are defined by [20]. Some of these depend on particular communications protocols (e.g. H.323) and on particular parameters. We choose to abstract the key call control actions as follows:

- *connect_to* initiates a new and independent call
- *reject_call* rejects a call, i.e. prevents it from completing
- *forward_to* changes the destination of the call
- *fork_to* adds an alternative leg to the call
- *add_party* adds a new party to an existing call
- *remove_party* removes a party from the call
- *add_medium* adds a new medium to the call
- *remove_medium* removes a medium from the call
- *remove_default* removes the default medium from the call
- *disconnect* disconnects the call

This list of actions provides an abstract view of the call processing cycle in APPEL: an initial connection action can be followed by reject, forward or fork. During the call, parties can be added or removed. Media can be added or removed. The call can then be disconnected. Note that 'disconnect' is not an action in APPEL at present, however our analysis has led to the conclusion that it should be added.

The action *remove_default* deserves mention, especially since there is no *add_default*. Certain actions, such as *connect_to*, implicitly reserve the default medium for the call (usually audio). Although the *remove_default* action also does not exist in APPEL, it is implicit. We have made it explicit because we will see later that it is useful to consider the availability of the default device in the pre/post-conditions.

All these actions have parameters, which can themselves cause interactions. However the treatment of parameters would add considerable complexity to our analysis. We have abstracted away from parameters in our initial analysis of conflicts. We have also omitted actions that do not directly relate to call control (e.g. those that log or send messages). Our method can be applied to them, but this has not been done here because it would have complicated the presentation of the approach with little additional insight. For one thing, our tables would have had to be much larger.

4.2 Pre/Post-Conditions for APPEL Actions

Like all real-life distributed systems, call processing systems are complex and the conditions involved are correspondingly complex. In practical terms, analysis must be limited to a few important characteristics. Following the example of [22, 23], we have decided to concentrate our analysis on two aspects: connection (or call) state and media state. We therefore characterize the *state of a system* as a pair <connection state, media state>.

Table 1 shows the table of pre/post-conditions that was developed for this study. It represents a simplified and abstract view of call processing in APPEL. Setting up this table is a delicate task which determines the results of the analysis.

Call processing progresses through three mutually exclusive connection states: *NoCall*, *CallSetup*, *MidCall*. Note that Table 1 does not describe a state machine, i.e. transitions and associated actions from state to state. For example, there is no action that leads from *CallSetup* to *MidCall*. It is assumed that this state transition will occur as a consequence of events that are not shown in the table. That is, the table intentionally does not describe how the real system works 'behind the scenes'.

The table identifies two categories of media: the default medium (e.g. audio) and media in general (e.g. video, messaging). It is useful to make this distinction because a call is always initiated with a default medium. This may later be augmented or replaced by something else (e.g. video may be added, or the call may be reduced to messaging only).

The analysis presented in the following sections identifies six cases of conflict, in the three major categories we have identified:

1: Concurrency or Pre-Condition - Connection State
2: Concurrency or Pre-Condition - Media State
3: Disabling - Connection State
4: Disabling - Media State
5: Result or Post-Condition - Connection State
6: Result or Post-Condition - Media State

Action	Pre-conditions		Post-conditions	
	Connection State	**Media State**	**Connection State**	**Media State**
connect_to	NoCall	DefaultAvailable	CallSetup	DefaultReserved
reject_call	CallSetup	DefaultReserved	NoCall	DefaultAvailable
forward_to	CallSetup	DefaultReserved	CallForwarded	DefaultAvailable
fork_to	CallSetup	DefaultReserved	CallForked	DefaultReserved
add_party	MidCall	DefaultAvailable	PartyAddedToCall, MidCall	DefaultReserved
remove_party	MidCall, PartyAddedToCall	DefaultReserved	MidCall	DefaultAvailable
add_medium	MidCall	MediumAvailable	MidCall	MediumReserved
remove_medium	MidCall	MediumReserved	MidCall	MediumAvailable
remove_default	MidCall	DefaultReserved	MidCall	DefaultAvailable
disconnect	MidCall	DefaultReserved	NoCall	DefaultAvailable

Table 1. Pre/post-conditions for APPEL actions

Connection State 1	Connection State 2
NoCall	MidCall
NoCall	CallSetup
CallSetup	MidCall
CallSetup	NoCall
MidCall	NoCall
MidCall	CallSetup

Table 2. Connection State incompatibilities

4.3 Concurrency Conflicts

As mentioned, in this case, the question is whether two actions can be executed starting from the same system state. This will not apply if they require states that are incompatible. For example, action *connect_to* cannot be concurrent with any other action, since it is the only action that can be executed before a call exists. Similarly, *add_party* requires the system to be in a state where the default medium is available, while *remove_party* instead requires the default medium to have been reserved. Note that this does not mean that the two actions are necessarily incompatible. Our analysis

connect_to	reject_call	forward_to	fork_to	add_party	remove_party	add_medium	remove_medium	remove_default	disconnect	Action Pair
	1	1	1	1	1	1	1	1	1	**connect_to**
				1	1	1	1	1	1	**reject_call**
				1	1	1	1	1	1	**forward_to**
				1	1	1	1	1	1	**fork_to**
										add_party
										remove_party
										add_medium
										remove_medium
										remove_default
										disconnect

Table 3. Pre-condition conflicts for Connection State (case 1)

is not sufficiently detailed for such certitude. Indeed in every method reported in the literature, feature interaction detection only suggests the possibility of an interaction, which must be confirmed by domain experts, in consideration also of specific contexts.

The approach requires incompatibilities in state to be defined. Table 2 shows the incompatibilities between connection states that we have used. Essentially, the table says that the three connection states are mutually incompatible.

As a consequence of this, we obtain the results shown in Table 3 for incompatibilities among connection states. We can see here that *reject_call* and *add_party* are incompatible because each requires the system to be in a different state than the other. Two different *connect_to* actions are not incompatible for this reason, although they will be incompatible for other criteria, see below. Obviously the table is symmetric.

The other aspect to be considered is media state. The table of media state incompatibilities is not shown here because it is rather simple. It indicates potential conflicts if the actions require some medium (including the default) to be both reserved and available. Here again, the necessary simplification should be understood.

A call system will have a variety of selectable media and default media. To be complete and precise, one would have to consider the specific media and defaults in the system under consideration, as well as specific operations that reserve and release them. This type of detail is possible in practice, but is irrelevant for the purpose of this paper, which is illustrating the method.

4.4 Disabling Conflicts

As mentioned, it is possible for an action to leave the system in a state where another action is impossible. This can be determined by checking post-conditions against pre-conditions. Concerning the connection state, the incompatibilities to be considered are the same as earlier: the three states are incompatible. Thus, an action that must find the system in state *MidCall* cannot immediately follow an action that leaves the system in state *CallSetup*. Similarly for media state, an action that requires default media to be reserved cannot follow an action that sets default media available, and so on.

Table 4 shows the result obtained with respect to connection state. It is not symmetric because the disable relation is not symmetric.

connect_to	reject_call	forward_to	fork_to	add_party	remove_party	add_medium	remove_medium	remove_default	disconnect	Action Pair
3										**connect_to**
				3	3	3	3	3	3	**reject_call**
				3	3	3	3	3	3	**forward_to**
	3	3	3							**fork_to**
	3	3	3							**add_party**
										remove_party
	3	3	3							**add_medium**
	3	3	3							**remove_medium**
	3	3	3							**remove_default**
				3	3	3	3	3	3	**Disconnect**

Table 4. Disabling conflicts for connection state (case 3)

4.5 Result Conflicts

Two actions are also incompatible if they lead to incompatible post-conditions. Again, these can refer to connection state or to media state. In the case of connection state, if an action leads to a certain connection state, another compatible action must lead to either the same state or to the next state. As mentioned, the cycle of states is as follows: *NoCall* leads to *CallSetup* which leads to *MidCall*, which leads again to *NoCall*. An action which leads to one of these states is incompatible with an action which jumps one link in the sequence. As an example, *reject_call* leads to *NoCall*,

while *add_medium* leads to *MidCall*. Clearly a link is skipped here, since between the two we need an operation that establishes *CallSetup*. Hence the incompatibility. The complete incompatibility table between connection states will not be given for brevity, since essentially it reflects this reasoning. Note that this definition of state incompatibility is perhaps disputable, but this does not affect the validity of the method, which can be adapted to other definitions. Table 5 shows conflicts according to this criterion.

connect_to	reject_call	forward_to	fork_to	add_party	remove_party	add_medium	remove_medium	remove_default	disconnect	Action Pair
	5								5	**connect_to**
		5		5	5	5	5	5		**reject_call**
	5								5	**forward_to**
										fork_to
5										**add_party**
5										**remove_party**
5										**add_medium**
5										**remove_medium**
5										**remove_default**
		5		5	5	5	5	5		**disconnect**

Table 5. Post-condition conflicts for connection state (case 5)

For media state, the incompatibilities are again simple. If the actions lead to some media being available *and* reserved, or the default media being available *and* reserved, there is a post-condition incompatibility because of media. To save space, the results of this analysis are given in Table 6, the recapitulative table.

4.6 Overall Results

Table 6 shows the complete results for the six types of conflicts we have discussed.

We have also analyzed other situations, for example the case where an action *enables*, or sets the pre-conditions, of another action [14]. In this case, the postcondition of the first action implies the precondition of the second one. These situations cannot be discussed for lack of space.

4.7 Assessment

How would a domain expert in call control (or APPEL) view these results? An expert is guided by a pragmatic understanding of the system's behavior, while the approach of this paper is formal and systematic, at a high level of abstraction. As mentioned, the parameters of actions are disregarded, as well the view of system state is much simplified, and this means it is not said, for example, which specific party or medium

is being added or removed. As a consequence, the method discussed here is intentionally pessimistic. However, since the goal of the work is to identify action pairs that require closer study because of potential conflicts, the approach is successful.

connect_to	reject_call	forward_to	fork_to	add_party	remove_party	add_medium	remove_medium	remove_default	disconnect	Action Pair
3,4	1,2,5,6	1,2,6	1,2	1,4	1,2,6	1	1	1,2,6	1,2,5,6	**connect_to**
1,2,6	4	4,5	4,6	1,2,3,5,6	1,3,4,5	1,3,5	1,3,5	1,3,4,5	1,3,4	**reject_call**
1,2,6	4,5	4	4,6	1,2,3,6	1,3,4	1,3	1,3	1,3,4	1,3,4,5	**forward_to**
1,2,4	3,6	3,6	3	1,2,4	1,6	1	1	1,6	1,6	**fork_to**
1,4,5	1,2,3,6	1,2,3,6	1,2,3	4	2,6			2,6	2,6	**add_party**
1,2,5,6	1,4	1,4	1,4,6	2,6	4			4	4	**remove_party**
1,5	1,3	1,3	1,3			4	2,6			**add_medium**
1,5	1,3	1,3	1,3			2,6	4			**remove_medium**
1,2,5,6	1,3,4	1,3,4	1,3,4,6	2,6	4			4	4	**remove_default**
1,2,6	1,4	1,4,5	1,4,6	2,3,5,6	3,4,5	3,5	3,5	3,4,5	3,4	**disconnect**

Table 6. Summary of conflicts

5 Detecting Conflicts in APPEL with Alloy

The method described in the previous sections could be implemented in different programming languages. Instead of using a conventional programming language, we decided to experiment with the model checker Alloy. This decision was taken for two reasons: Alloy allows high-level, conceptual modeling of systems architectures and their properties. Further, it has the capability of checking logical models, and thus is open to the possibility of extending our method to logically more complex pre/post-conditions.

5.1 Alloy language and tool

Alloy [11] is a formal method that includes a logic, a language, and a tool. The logic is primarily a relational logic. The language provides a user-friendly representation for the logic. It supports several specification styles, called *predicate calculus style*, *relational style* and *navigational style* (the last one being the most expressive and most commonly used). It includes a type system and mechanisms to favor reusability. The tool is essentially a first-order logic model-checking tool, based on the use of off-the-shelf satisfaction algorithms. Alloy allows one to describe a system model, and will check it for consistency. It is also able to check whether certain properties are true for the system. However the user of Alloy is required to specify a finite size for the model by the execution system, meaning that inconsistencies not found for the size specified could, at least in theory, appear for different sizes.

Signatures are used in Alloy to define types, e.g.

```
abstract sig Rules {
  trigger : one OBtrigger,         // there is one trigger
  condition : lone OBcondition,    // zero or more conditions
  action : some OBaction           // the set of acts is non-empty
}{
  #action = 2
}
```

defines a *rule*, and at the same time states that we are interested in generating exactly two objects of type *action* (for which there can be several, *some*), since we consider only conflicts between pairs of actions. Inheritance relationships can exist between signatures.

Facts constitute a data base of facts that are known in the system, e.g. the pre/post-conditions of the actions (see Table 1):

```
fact {
  connect_to.PreConnState = NoCall
  connect_to.PreMediaState = DefaultAvailable
  reject_call.PreConnState = CallSetup
  reject_call.PreMediaState = DefaultReserved
. . .
}
```

Or the fact that connection states are pairwise incompatible (encoding Table 2).

```
fact AC {
 IncompSet.ConcConflict_Incomp_ConnState =
    MidCall -> NoCall +
    MidCall -> CallSetup +
    NoCall -> MidCall +
    NoCall -> CallSetup +
    CallSetup -> MidCall +
    CallSetup -> NoCall
}
```

Predicates are properties that can be true or false. *Assertions* are properties that can be *checked* by the tool, and for which the tool will try to find a counterexample. For example, the following predicate is true if two actions are in concurrency conflict because of the connection state in their pre-conditions:

```
pred Conc_Conflict_ConnState ( a1 : OBaction, a2 : OBaction ) {
  some v : a1.PreConnState, w : a2.PreConnState |
    (v -> w) in IncompSet.ConcConflict_Incomp_ConnState
}
```

C12 *asserts* that predicate *Conc_Confl_ConnState* is true for the two objects *connect_to* and *reject_call*.

```
assert C12 {
  Conc_Confl_ConnState ( connect_to, reject_call )
}
```

The Alloy tool is asked to *check* this assertion with:

```
check C12
```

The result is that there is no counterexample to the predicate, thus the assertion is valid and the two actions conflict in their pre-conditions, making them unsuitable for concurrent execution.

The core specification of this problem is about 3 pages of Alloy code. A further 22 pages are required for the *check* and *assert* statements needed to determine the presence of conflicts in all cases of interest.

5.2 Alloy Execution

In its internals, the Alloy tool expresses the constraints in terms of Boolean expressions and then tries to solve these by invoking off-the-shelf SAT solvers. This problem is of exponential complexity. However, SAT solvers are improving in efficiency and many non-trivial problems can be treated. Current solvers can handle thousands of Boolean variables and hundreds of expressions, although of course much depends on the type of the expressions [11]. Thus, the Alloy user must find a judicious compromise between detail and abstraction, as well as size of model to be checked. Too many details or too large a model will cause the tool to run out of memory or time.

The Alloy tool provides a number of useful graphical representations of its results: graphical, tree, XML.

Alloy models can be checked in one of two ways:

- With the function *VerifActions* which will check the whole model, but will find at most one (arbitrarily chosen) conflict for each execution. Unfortunately Alloy cannot be asked to continue finding solutions, as Prolog can.
- By systematically checking assertions. To consider all cases for our model requires 600 executions (10 actions × 10 actions × 6 predicates). Each assertion takes about 2.5 minutes to check, for a total of around 25 hours.

The analysis was performed on a Pentium with dual 2.80GHz CPUs and 1GB of main memory. We used Alloy version 3. Version 4 offers improvements in usability, but it became available late in the progress of this work.

We are looking forward to improvements in the Alloy tool to simplify and expedite its use in a case like ours, where several hundreds of assertions have to be checked.

It should be underlined that our algorithm would be much more efficient if implemented in a procedural programming language, however we wanted to work with a formal technique which allows a view that is close to the problem specification.

6 Conclusions

We have described and justified a method for finding conflicts between call processing actions in a VoIP context, extending ideas in the work of [22,23] and others. We have demonstrated the effective application of this method to the actions of APPEL. Verification was undertaken using Alloy for first-order model checking. We have focused on APPEL and Alloy mainly because we are familiar with them. We plan experimentation and comparison with other applications, other policy languages and other formal tools. In another case study, the method was used to check the

results of [23] with regard to LESS, and happily we were able to confirm them, as well as to complete them with the detection of a few additional conflicts [14].

The contributions of this work are as follows:

- The approach allows potential conflicts among policies to be determined through analyzing the pre/post-conditions of their actions. This is a general idea that is not restricted to call control, APPEL or Alloy.
- As has been seen with APPEL, the method is successful in identifying genuine conflicts that need to be resolved by a domain expert.
- The approach provides a (partial) model of policy actions by defining their pre/post-conditions. In the context of this paper, this gives more precise meaning to APPEL.

Note that the usefulness of this method is not limited to static feature interaction filtering. Understanding which actions conflict and why is useful in a number of areas of feature interaction research. This information is useful for feature interaction avoidance, for feature interaction detection, and for feature interaction resolution. Most of the methods that have been proposed in these areas assume that it has been previously determined by some other method that certain actions conflict. Neither is the method limited to single user interactions, since in principle conflicting actions can be in different users' policies [17]. Our method can be integrated in other methods, i.e. the merge algorithm used in LESS.

More detailed presentation of these results can be found in [14].

Future work will deal with various generalizations mentioned in the paper. A more complete model should be developed for APPEL and the pre/post-conditions of its actions. In particular, action parameters and more complete state descriptions should be taken into consideration. We plan to extend the approach to other policy languages, as well as to investigate other tool support besides Alloy.

Acknowledgments

This work was funded in part by the Natural Sciences and Engineering Research Council of Canada, the UK Royal Society, and the Royal Society of Edinburgh. The authors thank Gemma Campbell (University of Stirling) for discussions about detecting conflicts in APPEL, and Xiaotao Wu for discussion about his method.

References

1. Aiguier, M., Berkani, K. and Le Gall, P: Feature specification and static analysis for interaction resolution. *Proc. Formal Methods'06*, LNCS 4085, 364–379, 2006.
2. Calder, M., Kolberg, M., Magill, E. H. and Reiff-Marganiec, S.: Feature interaction: A critical review and considered forecast, *Computer Networks*, 41:115–141, Jan. 2003.
3. Cameron, E. J., Griffeth, N. D., Lin, Y.-J., Nilson, M. E., Schnure, W. K. and Velthuijsen, H.: A feature-interaction benchmark for IN and beyond, *IEEE Communications Magazine*, 31(8):18–23, Aug. 1993.
4. Campbell, G. Turner, K.J. Policy calling filtering for call control. These proceedings.
5. Crespo, R.G., Carvalho, M., Logrippo, L.: Distributed resolution of feature interactions for internet applications. Computer Networks 51 (2), 382-397, Feb. 2007.
6. Damianou, N., Dulay, N., Lupu, E. Sloman, M.: The Ponder specification language. Workshop on Policies for Distributed Systems and Networks (Policy2001), Jan. 2001.

7. Felty, A.P., Namjoshi: Feature specification and automatic conflict detection, *in* Calder, M. and Magill, E. H. (eds.), *Proc 6th. Feature Interactions in Telecommunications and Software Systems*, 179–192, IOS Press, May 2000.
8. Felty, A.P., Namjoshi: Feature specification and automated conflict detection, *ACM Trans. on Software Engineering and Methodology*, 12(1):3–27, Jan. 2003.
9. Gorse, N., Logrippo, L., Sincennes, J.: Formal detection of feature interactions with logic programming and LOTOS, *Software and System Modeling*, 5(2):121–134, (mistakenly published as *Detecting feature interactions in CPL*), Jun. 2006.
10. Griffeth, N.D., Blumenthal, R., Gregoire, J.-C., Ohta, T.: Feature interaction detection contest of the fifth international workshop on feature interactions. *Computer Networks*, 32(4):487-510, April 2000.
11. Jackson, D.: *Software Abstractions: Logic, Language, Analysis*, MIT Press, 2006.
12. Kolberg, M., Magill, E., Marples, D., Reiff, S.: Second feature interaction context. In: M. Calder, E. Magill (Eds.) *Feature Interactions in Telecommunications and Software Systems VI*. IOS Press, 2000.
13. Kolberg, M., Magill, E.H. and Wilson, M E.: Compatibility issues between services supporting networked appliances, *IEEE Communications Magazine*, 41(11):136–147, Nov. 2003.
14. Layouni, A.F.: *Méthode formelle pour la détection d'interactions de fonctionnalités dans les systèmes de politiques*. Mémoire de maîtrise, Université du Québec en Outaouais, Département d'informatique et ingénierie, 2007 (forthcoming).
15. Lennox J, Wu X. and Schulzrinne H.: *CPL: A language for user control of Internet telephony services*, RFC 3880, Internet Engineering Task Force, Oct. 2004.
16. Montangero, C., Reiff-Marganiec, S. and Semini, L.: Logic–based detection of conflicts in APPEL policies, *Proc. Symposium on Fundamentals of Software Engineering* (FSEN'07), Feb. 2007.
17. Nakamura, M., Leelaprute, P., Matsumoto, K, Kikuno, T.: On detecting feature interactions in the programmable service environment of Internet telephony. *Computer Networks* 45(5): 605-624 (2004).
18. Shankar, C., Ranganathan, A., Campbell, R.: An ECA-P policy-based framework for managing ubiquitous computing environments. *Mobiquitous 2005*, July 2005.
19. Turner, K. J., Blair, L.: Policies and conflicts in call control, *Computer Networks*, 51(2):496–514, Feb. 2007.
20. Turner, K. J., Reiff-Marganiec, S. and Blair, L.: *APPEL: The ACCENT project policy environment/language*. Technical Report CSM-161, University of Stirling, UK, Dec. 2005.
21. Turner, K. J., Reiff-Marganiec, S., Blair L., Pang, J., Gray, T., Perry, P. and Ireland, J.: Policy support for call control, *Computer Standards and Interfaces*, 28(6):635-649, 2006.
22. Wu, X. and Schulzrinne, H.: Handling feature interactions in the Language for End System Services, *in* Reiff-Marganiec, S. and Ryan, M. D. (eds.), *Proc. 8th. Feature Interactions in Telecommunications and Software Systems*, 270–287, IOS Press, 2005.
23. Wu, X. and Schulzrinne, H.: Handling Feature Interactions in the Language for End System Services, *Computer Networks* 51 (2), 515-535, 2007.