Gavin A. Campbell and Kenneth J. Turner. Policy Conflict Filtering for Call Control, in L. du Bousquet and J.-L. Richier, editors,
Proc. 9th. Int. Conf. on Feature Interactions in Software and Communication Systems, IOS Press, Amsterdam, September 2007.

Policy Conflict Filtering for Call Control

Gavin A. Campbell and Kenneth J. Turner

Computing Science and Mathematics, University of Stirling, Stirling FK9 4LA, UK
gca | kjt @cs.stir.ac.uk

Abstract. Policies exhibit conflicts much as features exhibit interaction. Since policies are defined by end users, the combinatorial problems involved in detecting conflicts are substantially worse than for detecting feature interactions. A new, ontology-driven method is defined for automatically identifying potential conflicts among policies. This relies on domain knowledge to annotate policy actions with their effects. Conflict filtering is performed offline, but supports conflict detection and resolution online. The technique has been implemented in the RECAP tool (Rigorously Evaluated Conflicts Among Policies). Subject to user guidance, this tool filters conflicting pairs of actions and automatically generates resolutions. The approach is generic, but is illustrated with the APPEL policy language for call control. The technique has improved the scalability of conflict handling, and has reduced the effort required of the previous manual approach.

Keywords: Call Control, Conflict Detection, Ontology, OWL, Policy

1 Introduction

1.1 Policies and Features

Policies are rules used to control a system dynamically through a set of actions to be performed in specified circumstances. Policies are typically defined by an event, a condition and an action. Historically, policy-based systems have been developed in domains such as access control, quality of service, security and system management. In all these applications, policies are typically created and maintained by administrators. However, the authors' approach is unusual in being designed for ordinary system users.

During the past decade, many policy languages and systems have been developed to decentralise the control of system behaviour, to automate system management, and to give more control to end users. This added flexibility has the advantage that users can tailor services more accurately to their needs, reducing reliance on generic system facilities. Traditional feature-based approaches lack flexibility. In telephony, for example, the features are mostly defined by the network operator. Users have little choice except to select the features they wish and to define a few feature parameters.

Systems that offer multiple, independently-defined features are prone to interactions – a well-known situation where the behaviour of one feature may affect another. Many feature interactions have been identified in call control. Detecting these interactions is often problematic due to the large numbers of features (several hundred in a typical PBX). Resolving the interactions can also be problematic because features are low-level units of functionality.

It is often necessary to understand the user's true intention before obtaining a satisfactory resolution. For example, consider the well-known interaction between Do Not Disturb and Alarm Call. The user's intention was presumably to avoid calls from others, but not the alarm call from the exchange. Policies are closer to user goals (e.g. 'I do not wish to be called by anyone') and so more faithfully reflect user intentions. Resolving interactions or conflicts is facilitated by the higher-level approach of policies.

This paper presents an approach to conflict handling using domain knowledge captured in an ontology. Collecting this knowledge is a manual step. However, conflict detection is then fully automated using the RECAP tool (Rigorously Evaluated Conflicts Among Policies). Conflict resolution is partially automated by RECAP – outline resolution policies are automatically generated, for completion by the domain expert using a policy wizard. The general idea is that conflicts are identified and specified through offline filtering. The resulting conflict resolution policies are then use online.

1.2 Ontology Support for Policies

The authors use a policy system called ACCENT (Advanced Component Control Enhancing Network Technologies). This includes a policy server that supports the APPEL policy language, a wizard for creating and editing policies, and a variety of supporting interfaces for various application domains.

In recent research, the authors have extended APPEL to support new and multiple domains. As the core schema of APPEL is generic, it can be extended for different applications by adding further schemas. However, this does not adequately deal with concepts in the application domains. The authors have therefore developed additional support for APPEL through a range of ontologies.

The new approach uses OWL (Web Ontology Language) to describe the core APPEL language. The core ontology is then extended hierarchically to define user interface information and to specialise the language for particular domains. This has increased the extensibility and precision of the policy language. APPEL is supported by a wizard that offers a web-based interface for creating and editing policies. This has been re-engineered to replace hard-coded domain information (for call control) with information stored within the ontologies. The result is a highly flexible user interface, easily adaptable to reflect new application domains.

1.3 Related Work

Policy conflict is equivalent of feature interaction in telephony and related domains. Since policies are defined in a decentralised manner, the potential for unwanted interaction is far greater than that of conventional feature-based systems. The increased flexibility that policies offer to users is offset by more pervasive, complex and subtle conflicts among policies.

Conflicts in a policy-based environment are often caused by the simultaneous execution of policies with contradictory actions. (Conflicts can also arise between actions and system state, i.e. the result of previous actions.) Policy conflict requires study of three different aspects: filtering conflict-prone policies, defining conflict detection mechanisms, and defining a conflict resolution strategy. Although policy filtering is a

new departure, conflict detection and resolution have already been studied. In system management, for example, conflict detection and resolution techniques include [**?**,**?**]. Enhancements to COPS (Common Open Policy Service, RFC 2748) are aimed at managing policy conflict through rigorous definition of actions.

Many techniques have been developed to automate feature interaction detection at the specification stage. Techniques in feature interaction detection have focused heavily on a variety of formal methods such as process algebras, automata and (temporal) logic. Of these, techniques for filtering interaction-prone features are the most relevant. However, few are directly relevant to policy-based control. Nonetheless, the ideas have influenced the work reported here.

The notion of interaction filtering was initially presented in [**?**]. The filtering process is followed by detailed checking and refinement of conflicts. Several tools support an automated approach to filtering feature interactions. One example is a prototype designed to detect interactions in a call environment [**?**]. This filters interactions among IN services, using simple descriptions of the static structure for each service. Interactions are detected for groups of services used in particular call scenarios.

Formal approaches have been followed by a number of researchers. FIX (Feature Interaction Extractor [**?**]) is an example of a domain-independent approach, although only application to telephony has been reported. This uses the model checker COSPAN to run consistency tests on feature specifications. In a further stage, the tool user can investigate the generated scenarios and decide on their accuracy. [**?**] presents a filtering technique based on Use Case Maps and applies it to telephony features. [**?**] uses preconditions and postconditions to identify inconsistencies in features for LESS (Language for End Systems Services).

[**?**] describes work that is directly relevant to this paper as it uses temporal logic to formalise the semantics of APPEL. This leads to a formal basis for automated detection of conflicts. In other work on APPEL, [**?**] presents a method for discovering conflicts based on the pre/post-conditions of actions. This allows semantically-based inferences to be drawn about the compatibility of actions. However, it is technically more complex than the simple and intuitive approach of the work reported here. As complementary techniques, future study will investigate how [**?**,**?**] can be reconciled and integrated with the authors' approach.

The work reported here differs in important respects from the foregoing:

- Policies rather than features are used for control. These support higher-level statements of user intentions, and facilitate the resolution of conflicts.
- The approach is adapted to many domains, including ones outside telephony. For example, the authors use it to detect conflicts in home care and in sensor networks.
- A formal specification of the system and its policies is not required. In practice a precise specification is usually infeasible because the system is too complex, is proprietary, or is open-ended because users can define their own features or policies.
- The approach is intentionally less formal. This has the advantages of being simpler to set up and more intuitive, i.e. relying only on domain knowledge. Domain experts, rather than formalists, can define the information needed for conflict filtering. The analysis is efficient and domain-oriented.

1.4 Paper Outline

Section 2 presents an overview of the ACCENT policy system, the APPEL policy language, and its approach to conflict detection and resolution. Section 3 introduces ontologies, and outlines how they were used to model APPEL. Section 4 explains how ontologies are used to identify policy conflicts. Section 5 discusses the approach to conflict filtering and the associated tool support. Section 6 evaluates the results.

2 The ACCENT Policy Approach

2.1 Policy System and Language

The ACCENT policy system (Advanced Component Control Enhancing Network Technologies, *www.cs.stir.ac.uk/accent*) was originally designed to allow users to tailor (Internet) call handling to their own preferences. As illustrated in figure 1, the ACCENT system is split across three layers. At the lowest level, the system layer connects the policy system to its external environment. Policy enforcement is handled by the policy system layer that incorporates the policy server, policy store (where policies reside) and policy database (containing user login and server configuration data). At the top level, the user interface layer is where users create policies and where contextual information is obtained. Policies are defined and edited via a web-based policy wizard [**?**]. Each policy is saved as an XML document and uploaded to the policy store. The general approach of ACCENT is described in [**?**].

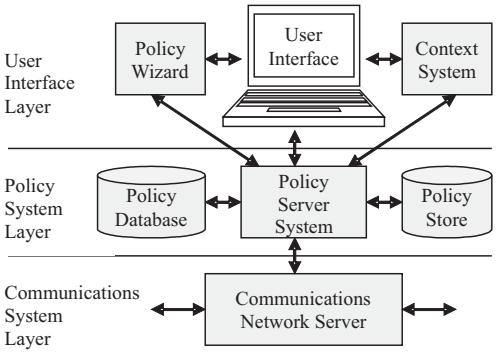


Fig. 1. ACCENT System Architecture

APPEL (ACCENT Project Policy Environment/Language [**?**]) is a comprehensive and flexible language, designed to express policies within the ACCENT system. Key factors in the design of APPEL include a simple but concise structure, ease of extension, and orientation towards ordinary users. APPEL comprises a core language and its specialisations for different application domains. The original specialisations were for call control and conflict resolution, but new specialisations have been developed for home care and sensor networks.

APPEL defines the overall structure of a policy document: regular policies, resolution policies, and policy variables. A policy consists of one or more rules in ECA form (Event-Condition-Action). Each rule has a combination of triggers (optional), conditions (optional), and actions (mandatory). The core language constructs are extended through specialisation for new applications.

A policy is eligible for execution if its triggers occur simultaneously and its conditions apply. Additional conditions may be imposed, such as the period during which the policy applies, or the profile to which the policy belongs. When the policy system is informed of an event, the applicable policies are retrieved and applied if eligible. As multiple policies can be triggered, conflicts may arise among their actions.

2.2 Conflict Detection and Resolution

Conflicts result from clashes between pairs of policy actions. As an example from call control, the caller may wish to conference in a third party whom the callee does not wish to speak to. The caller/callee policies propose *add/remove_party(person)* for some individual. These contradictory actions must be identified as conflicting. They must also be resolved, e.g. by giving the caller (as the bill payer) priority.

The ACCENT system allows for both static and dynamic conflict detection. Static detection is performed when a policy is defined and uploaded to the policy system, while dynamic detection occurs at run-time. Although both methods are permitted, only dynamic detection is currently implemented. This focus was intentional since run-time conflict handling is the more challenging task. Dynamic conflicts also subsume static conflicts. The actions resulting from a policy trigger are checked pairwise for conflicts. (The design of the language means that the order of comparison is irrelevant, and that only pairs need be checked.) The outcome is a set of non-conflicting actions.

Human guidance is almost inevitably required to determine how best to handle conflicts. Only certain 'technical' conflicts might be detected fully automatically. Even then, the treatment of a conflict requires judgment. As an example, suppose one user wishes to add video to a call but the other user wishes to avoid this. This is clearly an add/remove conflict. A trivial resolution would be to permit one or other policy to prevail. However, an acceptable resolution might be much more complex, e.g. using a third party to adjudicate the conflict.

As a further example, suppose one user wishes to add the G.723 audio codec to a call but the other user wishes to avoid it. This appears to be an identical kind of add/remove conflict. In fact it is not, because both parties (in H.323) must be willing to support the G.711 audio codec. There is therefore no need to treat this as a conflict. This illustrates that conflict detection requires domain knowledge and human intuition.

Conflict handling in ACCENT is defined by resolution policies that are distinct from regular policies. Resolution policies express when and how the system should respond to conflicts. Their effect is to process a set of proposed policy actions, selecting those that are compatible with the conflict handling rules. Resolution policies are specified as an extension of the core APPEL language, and therefore use the same syntax as regular policies. However, resolution policies use a different vocabulary because they serve a different purpose. The domain-specific actions of regular policies are the triggers of

resolution policies. Resolution policies can dictate generic outcomes (selecting among the proposed actions) or specific outcomes (dictating domain-specific actions).

APPEL has a built in notion of policy preference which allows a user to indicate how strongly they wish a policy to be applied. This allocates priorities to policies as one means of resolving conflicts. However, other resolutions may be used such as choosing the policy of a superior user, or choosing a longer-standing policy. Resolution policies gives considerable flexibility in that conflict handling is not hard-coded into the policy system. It is defined externally and can be domain-specific. To avoid infinite regress, resolution is performed just once. The approach ensures that the outcome is conflict-free, and does not require resolutions to be checked again for conflicts.

Conflict handling within ACCENT is described in [**?**]. The main limitation of this previous work was that resolution policies had to be defined manually. This was tedious and error-prone. The new work reported here describes an ontology-driven mechanism to automate conflict detection. The RECAP tool provides automated support for detecting conflicts and for creating outline resolution policies. The details of resolution require human judgment and are added in a further manual step.

3 Ontology Support for Policies

3.1 Ontology Background

An ontology is the set of terms used to describe and represent an area of knowledge, together with the logical relationships among these [**?**]. It provides a common vocabulary to share information in a domain, including the key terms, their semantic interconnections, and the rules of inference. Ontologies enable separation of domain knowledge from common operational knowledge in a system.

A variety of specialised languages are used to define ontologies. OWL (Web Ontology Language [**?**]) is a standard XML-based language. It is supported by a wide range of software, and can be integrated with other techniques. In addition, OWL provides a larger function range than any other ontology language to date. For these reasons, OWL was used to define the ontologies in the work reported here.

An OWL ontology defines classes, properties and individuals. A class represents a particular term or concept in a domain, while a property is a named relationship between two classes. An individual is an instance or member of a class, usually representing real data content within an ontology. Properties are defined for classes in the form of restrictions that specify the nature of a relationship between two classes. OWL supports inheritance within class and property structures. OWL can also import shared ontologies. The ontological basis for APPEL exploits this, using multiple documents for different aspects of the core language and its specialisation in various domains.

Ontology support for policies is provided by POPPET (Policy Ontology Parser Program Extensible Translation [**?**]). This uses the PELLET ontology reasoning engine (*pellet.owldl.com*) and the Jena ontology parser (*jena.sourceforge.net*). POPPET parses and integrates ontologies on behalf of the ACCENT system. Figure 2 illustrates the relationship between ACCENT and POPPET.

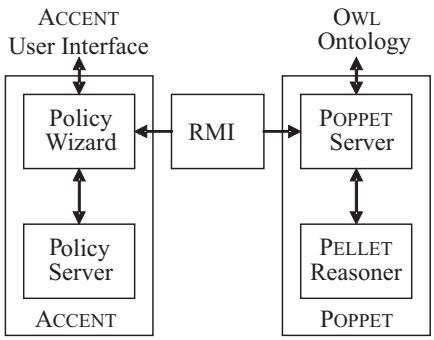


Fig. 2. Ontology Support by POPPET for ACCENT Policies

3.2 Ontologies for Policies

Ontologies were defined for the core of APPEL and its domain specialisations. Using OWL, three layers of ontologies were developed [**?**].

At the lowest level, *GenPol* (generic policy) defines core language elements such as variables, rules, triggers, conditions and actions. This includes the basic elements of a policy and the cardinality rules relating these. Each core element is defined as an ontology class. Relationships between classes are defined using ontology properties that link them. Using properties to describe the associations between concepts is a powerful means of modelling the structure of APPEL. The *GenPol* ontology contains no domain knowledge, only a definition of how high-level concepts may be combined to form a regular policy or resolution policy.

The ACCENT policy wizard [**?**] is a user-friendly front-end for creating and editing policies. Such a facility is key in supporting policy definition by non-technical users of the system. The wizard presents policy and domain information using near natural language. The user interface is not part of APPEL proper, but is essential for the system to be usable. Additional, wizard-related knowledge is therefore defined in *WizPol* (wizard policy) as an extension of *GenPol*. This specialises the core language for use with the wizard. Examples of wizard-specific facilities include the categorisation of triggers, conditions, actions and operators. In addition, a subset of the language functionality is matched to the skill or authorisation level of a user.

The *GenPol* and *WizPol* ontologies define domain-independent aspects of regular policies and resolution policies. To specialise the language for a new domain, a further ontology is created to import and extend these base ontologies; importing *WizPol* implicitly imports *GenPol* as well. A domain-specific ontology can contain arbitrary new concepts, but all policy language concepts must be subclasses within the hierarchy defined by the base ontologies. Consequently, as these ontologies are combined through an import mechanism only, they do not suffer incompatibility issues.

The *CallControl* domain ontology specialises APPEL for call handling. Significant extensions include call control triggers, conditions and actions. Using properties defined in *GenPol*, constraints may be placed on individual triggers, conditions and actions. This defines their use for certain user levels and for display categories within the wizard. In

addition, properties define which actions and conditions are permitted with a particular trigger, and the valid range of operators associated with each condition parameter. Further user interface and data type aspects may be included in a domain-specific ontology.

4 Automated Conflict Detection

4.1 Action Effects

Conflicts arise between policy actions with certain parameters. When two actions with a similar effect are executed simultaneously, the result may be a conflict. For example, actions that add and remove the same aspect are potentially in conflict. Thus, the call control actions *add_party* and *remove_party* are likely to contradict each other. Other conflicts are far more subtle, and cannot easily be identified by naming alone.

Action parameters may use enumerated types, e.g. call control parameter *medium* has possible values *audio*, *video* and *whiteboard*. Actions plus selected parameters allow a deeper exploration of conflicts. Where an action has an enumerated parameter type, conflicts between instances of the same action are likely only if the parameters are the same. For example, call control action *add_medium(audio)* could be considered to conflict with a second *add_medium(audio)*. However, if the second action wished to add *video* then this would not be an obvious conflict. For this reason, actions with distinct values in an enumerated parameter set are treated as distinct actions.

In general, an action must be considered along with a subset of its parameters. In a domain like call control, there is a rich set of action names that suggest conflicts in themselves. Even there, it is often necessary to take parameters into account. For example, adding one party and removing a different party is not problematic. In other domains such as home care and sensor networks, a much more limited selection of action names is used. This is because actions are mainly differentiated by their parameters. A simple *device_out* action, for example, carries parameters that indicate the action type, device class, device instance and action parameters. Conflict detection has to work with the domain policy language as defined. In general, a subset of parameters must therefore be considered for conflict along with the basic action name. However, for simplicity the following text mainly refers to comparing actions.

Policy actions are defined to have one or more effects on the execution environment. These effects range from the technical (e.g. bandwidth) to the social (e.g. privacy). Internal policy actions affect the policy system itself, such as setting system properties or accessing system resources. Conflicts are likely where two actions share a common effect. Any action may potentially conflict with itself. However, all action pairs must be considered too. (As noted earlier, only two-way and not *n*-way conflicts need be considered.)

Figure 3 shows the effects of internal policy actions, while figure 4 shows the effects of call control actions. Call control actions with enumerated parameters are listed separately. Effects for internal policy actions are distinct from those of domain actions, as internal and external actions do not (normally) conflict. Effect categories differ depending on the language domain.

Action	Effect
log_event(arg1)	file
restart_timer(arg1)	timer
send_message(arg1,arg2)	channel
set_variable(arg1,arg2)	variable
start_timer(arg1,arg2)	timer
stop_timer(arg1)	timer
unset_variable(arg1)	variable

Fig. 3. Internal Action Effects

As discussed in section 3.2, ontologies have been used to model policy language concepts. It is therefore convenient to define action effects in these ontologies. However, the ontologies play no role in conflict detection or resolution. As conflict detection is not an integral part of APPEL, the concept of action effect is defined in the *WizPol* ontology. This allows conflict information to be specified outside the core language, while maintaining the advantage of further specialisation in domain-specific ontologies. Effect information is defined in *WizPol* through the *ActionEffect* class and the *hasActionEffect* property. The *ActionEffect* class is a superclass of all effect categories for both internal and domain-specific policy actions. Generic action effects are defined as subclasses of this class in *WizPol*. Domain-specific action effects are defined as subclasses within a separate domain ontology that imports *WizPol*. Each policy action is linked to the appropriate effect category class using the *hasActionEffect* property. This relates actions and effects, allowing a tool to infer overlapping actions.

4.2 Conflict Detection Algorithm

Only pairs of actions need to be considered in the analysis; there are no three-way conflicts. Potential conflicts between actions can be inferred from the ontology-defined effect categories through a two-stage algorithm. Firstly, any two actions sharing at least one common effect are identified as potentially conflicting. Secondly, actions with enumerated parameter types are analysed. Where two actions share the same parameter value then they potentially conflict, otherwise it is assumed that no conflict exists.

The total number of action pairs, including self-conflicts, is $\frac{n(n+1)}{2}$ where n is the number of possible policy actions. The policy language for call control has 21 possible actions and therefore a total of 231 action pairs. Conflict handling is commutative (if *A1* and *A2* conflict, then so do *A2* and *A1*) and associative (the way in which actions are paired is irrelevant).

The ontologies allow a list of actions to be inferred for each effect category. If two actions are present in some category, they can be marked as potentially conflicting. For example, the call control actions *fork_to* and *forward_to* potentially conflict as they both affect the *route*. All action pairs deemed to conflict in this way are then automatically reviewed with respect to their parameters. As explained earlier, actions with enumerated parameter types are considered in more detail. This increases the total number of action pairings as an action may be instantiated multiple times with different parameter values. For example, the action *add_medium* with its parameter is equivalent to three distinct

Action	Effect
add_caller(conference)	party, privacy
add_caller(hold)	party, privacy
add_caller(monitor)	party, privacy
add_caller(release)	party, privacy
add_caller(wait)	party, privacy
add_medium(audio)	medium, privacy
add_medium(video)	medium, privacy
add_medium(whiteboard)	medium, privacy
add_party	party, privacy
confirm_bandwidth	bandwidth
connect_to	route
fork_to	route
forward_to	route
note_availability	availability
note_presence	presence
play_clip	medium
reject_call	call
reject_bandwidth	bandwidth
remove_medium(audio)	medium
remove_medium(video)	medium
remove_medium(whiteboard)	medium
remove_party	party

Fig. 4. Call Control Action Effects

actions. This allows more accurate analysis of potential conflicts. Where actions might be treated as potentially conflicting based on a shared effect, this might not be the case when particular parameters are considered.

To explain this more concretely, some examples for *medium* are shown in figure 5. An action may conflict with itself if there is a common parameter (e.g. both instances wish to add video), and may not conflict if the parameters are different (e.g. they wish to add video and whiteboard respectively). Different actions with a common effect and the same parameter indicate potential conflict (e.g. attempting to add and remove audio simultaneously). Actions with a common effect and dissimilar parameters are assumed not to conflict (e.g. altering the medium by adding video and removing whiteboard).

Action1	Action2	Conflict
add_medium(audio)	*remove_medium(audio)*	✓
add_medium(audio)	*add_medium(video)*	×
add_medium(video)	*add_medium(video)*	✓
add_medium(video)	*remove_medium(whiteboard)*	×

Fig. 5. Sample Call Control Conflicts with Action Parameters

5 The RECAP Conflict Filtering Tool

5.1 Automated Support for Conflict Filtering

The RECAP tool (Rigorously Evaluated Conflicts Among Policies) has been developed to automate the algorithm in section 4 for identifying conflict-prone actions. Figure 6 illustrates what the tool looks like on-screen. Taking the first line as an example, the tool shows pairs of actions (*add_medium(audio)* and *add_medium(audio)*), why they conflict (shared effect on *medium* and *privacy*), and when this conflict was last modified (automatically or manually).

Depending on the domain, the conflicts identified by RECAP may or may not be complete and correct. Conversely, subtle conflicts that are not automatically flagged can be added by hand. As noted earlier, conflict handling will always require human judgment and cannot be fully automated. Based on human guidance, RECAP produces conflict resolution policies.

RECAP is started by pointing at the relevant domain ontology. Using the action effects, the tool automatically constructs a matrix of all policy action pairs and highlights those deemed to be potential conflicts. The tool user may explore the matrix, confirming or refining each conflicting action pair. If closer inspection reveals that there is no real conflict, this pairing can be flagged as conflict-free. If an action is linked in an ontology to some effect, this may not be true of the actual implementation. Conflicts arising from this cause can be dismissed using the tool to undo the linking.

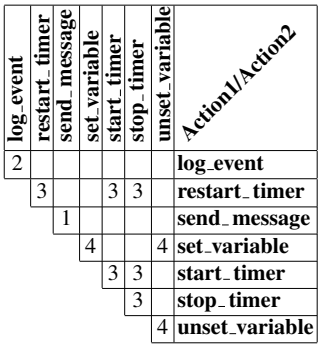
Potential conflicts are displayed in the tool matrix by noting the common effects in the appropriate cell. For convenience, internal and domain-specific actions are described here in separate figures though in practice they are combined by RECAP.

The result of filtering internal conflicts for APPEL is shown in figure 7. Conflicts are numbered in the figure according to the underlying effect. As an example of conflict, actions *start_timer* and *stop_timer* are in conflict because they both have a *timer* effect as indicated at their intersection. Some conflicts are non-obvious (e.g. *add_caller* and *add_medium*). Detailed study by a domain expert confirmed that all conflicts discovered are real, and that no conflicts had been missed. No changes were therefore needed in the analysis.

Call control actions deemed conflicting by RECAP are shown in figure 8. For simplicity, this figure shows conflicts between actions without parameters. In the tool, actions with enumerated parameter types are displayed and compared distinctly. Conflicts are numbered in the figure according to the underlying effect.

Detailed study by a domain expert confirmed that all detected conflicts but one are real, and that no conflicts have been missed. There is a possible problem in that *confirm_bandwidth* is indicated to conflict with itself due to a shared *bandwidth* effect. This could indeed be an error, as it might lead to bandwidth being allocated twice. As it happens, in the ACCENT system it is harmless to confirm bandwidth twice. Without human guidance, this action pair would be flagged as a conflict. It should be noted that the *bandwidth* effect is still required as it correctly identifies the conflict between *confirm_bandwidth* and *reject_bandwidth*.

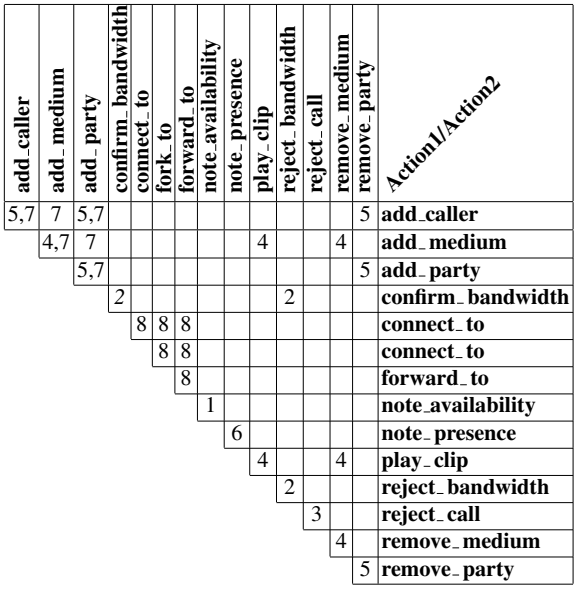
Fig. 6. Screen-Shot of RECAP



log_event	restart_timer	send_message	set_variable	start_timer	stop_timer	unset_variable	Action1/Action2
2							log_event
	3			3	3		restart_timer
		1					send_message
			4			4	set_variable
				3	3		start_timer
					3		stop_timer
						4	unset_variable

Conflict: 1 channel, 2 file, 3 timer, 4 variable

Fig. 7. Internal Conflicts identified by RECAP for APPEL



add_caller	add_medium	add_party	confirm_bandwidth	connect_to	fork_to	forward_to	note_availability	note_presence	play_clip	reject_bandwidth	reject_call	remove_medium	remove_party	Action1/Action2
5,7	7	5,7											5	add_caller
	4,7	7							4			4		add_medium
		5,7											5	add_party
			2							2				confirm_bandwidth
				8	8	8								connect_to
					8	8								connect_to
						8								forward_to
							1							note_availability
								6						note_presence
									4			4		play_clip
										2				reject_bandwidth
											3			reject_call
												4		remove_medium
													5	remove_party

Conflict: 1 availability, 2 bandwidth, 3 call, 4 medium, 5 party, 6 presence, 7 privacy, 8 route

Fig. 8. Call Control Conflicts identified by RECAP for APPEL

As demonstrated by figures 7 and 8, the automated conflict analysis (for call control) is very accurate. However, it confirms that human guidance is still needed in a small number of cases.

RECAP is mainly intended to analyse conflicts when a domain policy language is initially defined, using an ontology as the source of action effects. This initial analysis is saved to file and can subsequently be reloaded into the tool. This avoids the user and the tool from having to repeat a prior analysis, particularly if the user has manually modified the conflict list.

5.2 Automated Support for Resolution

RECAP turns the conflict list into a set of outline APPEL resolution policies that define the detection part of conflict handling. These policies define the conflicting triggers and parameter conditions, but resolution actions must be completed manually. The policies are automatically uploaded to the policy system, where the wizard is used to define the resolutions. Conversely, RECAP reads existing resolution policies and annotates the matrix with conflicts derived from these. This is a useful feature which allows conflicts defined manually via the policy wizard to be used in conjunction with conflicts identified by RECAP.

Resolution policies can be simple or complex, specific or generic, and dependent on many factors including the conflicting policies and their parameters. One or more actions may be required of a resolution. See [**?**] for a list of typical resolution policies. As an example, suppose one party wishes to add video to the call with *add_medium(video)*, while the other party wishes to conference in a third person with *add_party(person)*. This is correctly flagged as a conflict since the third party would be able to view the call parties and their workplaces (affecting *privacy*). Using human judgment, it might be decided to allow video and the third party. However, someone (e.g. a manager) should be included in the call to oversee it.

In view of this complexity, RECAP generates only outline resolution policies that specify default policy attributes, triggers corresponding to the conflicting actions, and default actions to resolve the conflict. The outline resolutions are then uploaded and customised using the wizard as normal. Resolution policy editing is dealt with by the wizard and not by RECAP. This allows RECAP to remain domain-independent and not be constrained to a particular resolution technique or policy language. An additional advantage is that resolution policies are then edited through the same interface as regular domain policies.

All default resolution parameters are defined by a properties file, and can therefore be readily modified according to local practice. The property file allows any structural components of outline resolutions to be altered. Resolution policies are normally disabled on upload. This ensures they are ignored by the policy server until they have been edited to include a specific resolution. This avoids incomplete or inconsistent resolutions from being used accidentally.

RECAP could be given a more user-friendly interface to change the default resolution policy structure and parameters. Currently this is achieved by manually editing the properties file. Although the tool is mainly intended for use during definition of a new application domain, there could be some value in easing later changes.

Policies in general are distinguished by unique identifiers, typically some phrase chosen by the user. Resolution policies automatically created by RECAP have machine-generated (but human-usable) identifiers. If the identifier of such a policy is changed manually, this could lead to duplication. The tool could detect this situation by looking for overlap of resolution triggers and conditions.

6 Conclusion

A technique and a tool have been introduced for (semi-)automated filtering of conflict-prone policies. Ontologies have been used to model the core and domain-specific aspects of APPEL – for regular and as well as resolution policies. Conflicts between policy actions are handled in ACCENT by resolution policies. Action effects defined in ontologies allow conflicting action pairs to be discovered as potential conflicts.

As has been seen, the analysis leads to very accurate results (for call control). Nonetheless, RECAP allows potential conflicts to be refined manually since a fully automated approach is impossible due to the complexity and subtlety of policy interactions. Following filtering, outline resolution policies are generated and uploaded for completion with the policy wizard.

RECAP offers an automated approach to conflict analysis and resolution where previously this was achieved manually. This has improved the scalability of APPEL, and has substantially reduced the time and complexity of dealing with conflicts. Associating actions with their effects is very simple compared to formal methods, but yields very good results. The straightforward and domain-oriented approach is much less expensive to use than one that requires a complete formal model.

RECAP provides a way of visually identifying conflicts within an arbitrary collection of policy actions. Unlike many existing approaches and tools, policies in any domain may be analysed easily by RECAP, and not just those for call control. The tool is also useful for policy applications where action parameters play a bigger role.

RECAP has been designed for stand-alone use. Although conflict data is mainly expected to derive from an ontology, conflict information may be input from a local file. Consequently, data generated by other tools or systems may be used by RECAP for conflict filtering. The only requirement is knowledge of the conflict data format used.

Although RECAP is aimed at filtering conflicts in the initial stages of specifying a new policy language, it may be used in later revisions of the language to refine conflicts and to generate resolutions.

Acknowledgements

The authors thank their colleagues Stephan Reiff-Marganiec (now at the University of Leicester) and Lynne Blair (who was on leave from Lancaster University during the development of ACCENT). Both contributed substantially to the design of the policy system that lies at the foundation of the work reported in this paper. Gavin Campbell's work on the PROSEN project was supported by grant C014804 from the UK Engineering and Physical Sciences Research Council.