*Kenneth J. Turner and Gavin A. Campbell. Goals and Conflicts in Telephony,
in Masahide Nakamura and Stephan Reiff-Marganiec (eds.),
Proc. 10th Int. Conf. on Feature Interactions in Software and Communications Systems,
pp. 3-18, IOS Press, Amsterdam, June 2009.*

Goals and Conflicts in Telephony

Kenneth J. Turner and Gavin A. Campbell
Computing Science and Mathematics, University of Stirling, Stirling FK9 4LA, UK
e-mail: kjt | gca @cs.stir.ac.uk

Abstract.
Goals are abstract, user-oriented objectives for how a system should behave. To be made operational, they are refined into lower-level policies that are executed dynamically. It is explained how previous work on policy-based management has been enhanced with a separate goal system that allows goals to be specified, analysed, refined, and achieved. It is shown how conflicts can arise at several levels among goals, and how these conflicts are detected and handled. Although the approach is general, it is illustrated through an application to Internet telephony.

Keywords.
Call Control, Goal, Goal Conflict, Goal Refinement, Policy-Based Management

1. Introduction

Goals are introduced as a high-level, abstract way for users to define system objectives in a non-technical manner. The concept of goal refinement is explained. In contrast to other work, which relies on static logical analysis, the approach here views refinement as a dynamic optimisation task.

1.1. Background

The trend in communications is for people to be always contactable – in the office, at home, or on the move. As a result, there is an increasing need for users to be able to manage their communications flexibly. Traditional telephony features are network-centric, being designed by the network operator and supported in network equipment. Internet telephony is becoming increasingly widespread. Following Internet philosophy, functionality tends to be located at the edges of the network. In particular, this means that customisation of call handling can be placed much more in the hands of end users.

The authors and their colleagues have exploited this trend by developing a policy-based management system for Internet telephony (e.g. [11]). Policies operate at a higher level than features, and offer many advantages such as orientation towards user needs, flexibility, and customisability. A system called ACCENT (Advanced Component Control Enhancing Network Technologies, *www.cs.stir.ac.uk/accent*) has been developed for supporting policy-based management. The ACCENT system was designed for end users, e.g. policies are formulated through a wizard rather than requiring programming or detailed technical knowledge. Although initially developed for Internet telephony, the approach is general and has now been extended to other applications such as managing sensor networks, wind farms and home care systems.

Although policies are higher-level and more user-oriented than features, they are still relatively imperative. What is required is a more abstract way for users to formulate their needs. This has now been achieved through the design and implementation of a goal system. Goals are persistent, abstract, user-oriented objectives for how a system should behave. They are declarative statements of what is required, not operational statements of how they should be achieved. In fact, they are sufficiently abstract that they cannot be realised directly. A process of refinement is needed to map the goals onto lower-level policies that can then be used to achieve them.

The work in this paper describes how goals are specified and statically analysed to create policies that realise them. Offline analysis is also concerned with appropriate definition of goals. Support at run time is provided to choose the most relevant policies according to the dynamically changing state of the system. The approach allows users to define high-level goals for how the system should behave (e.g. how calls are handled). The details of how goals are achieved can be largely hidden from the users.

Feature interaction is a long-known phenomenon in telephony. The authors and their colleagues have investigated policy conflict as the higher-level equivalent in policy-based systems [10]. The new work reported here has raised the approach to a more abstract level. Not surprisingly, conflicts also arise with the use of goals. As will be seen, such conflicts manifest themselves at several levels.

1.2. Goals and Policies for System Management

Goals have a venerable history in artificial intelligence, where they are often used to define desired states of a system. Goals are typically used by a planning system to build sequences of actions that achieve them. A similar approach has been adopted in agent-based systems.

Goals in the context of policy-based management have received little study. Here, the notion of a goal is an objective for system behaviour. Refinement of goals into policies is typically treated through logic: policies are identified to meet the goals through a process of logical entailment. This is essentially a static (offline) analysis. In the work reported here, goal refinement is viewed as a numerical optimisation problem. The main advantage is that goals can be supported in a fully dynamic way: as the system and its environment change, the policies that achieve goals are automatically evolved to give the best match to current circumstances.

In the formulation adopted, goals are associated with numerical measures (e.g. 'call cost' or 'multimedia use') for how well they are achieved. In general, such measures are maximised (positive goals) or minimised (negative goals). Goal measures are arbitrary functions over system variables, though for practical and technical reasons they are normally weighted sums. Since there are usually multiple (often conflicting) goals, their measures are combined into an overall evaluation function that assesses how well a candidate set of policies meets the full set of goals. The evaluation function is again an arbitrary formula, though normally a weighted sum of goal measures.

Goals are sufficiently high level that they cannot be used directly. Instead, goals are realised through prototype policies (prototypes for short) that contribute to them. These prototypes form a library of 'building blocks' for accomplishing goals. When goals are first defined, they are statically analysed against the available prototypes. This identifies the prototypes that *may* contribute towards each goal. However, the actual selection is de-

ferred until run-time since the most appropriate policies are likely to depend on the actual circumstances (e.g. the current bandwidth or the cost of a call). Prototypes are therefore selected and instantiated dynamically as normal policies. In addition, prototypes may have parameters that are dynamically optimised.

The benefit of this approach is that the existing ACCENT policy server is barely affected by the new goal system. As far as the policy server is concerned, it handles policies as usual (except that some policies derive from goals). Other mechanisms such as handling policy conflicts at run time can therefore operate as normal.

Policies are written in APPEL (Adaptable and Programmable Policy Environment and Language, *www.cs.stir.ac.uk/appel*). This is designed to be extensible for new application domains. The core policy language has extensions for regular policies and resolution policies. In turn, these are specialised for each domain: call control is currently one of several rather different applications. Since policies are internally XML, extensibility is achieved by defining APPEL through a hierarchy of schemas that build on each other.

Although these schemas define the formal structure of APPEL, this is not sufficient. The schemas are therefore supplemented by ontologies that define the concepts and relationships in each application domain. Again this is hierarchical, with a core ontology that is extended for each application.

For the work reported here, the schemas and ontologies have been extended to support goals and prototypes. The latter are intentionally defined in a very similar way to policies (both regular and resolution policies). This gives a consistent structure to the language, and also allows common software support throughout all system components.

1.3. Related Work

Planning in artificial intelligence goes back about 40 years (e.g. the STRIPS system). Much more recent is work on agent-based systems. As an example, 3APL (Agent Programming Language, *www.cs.uu.nl/3apl*) defines goals and beliefs. Plan revision rules allow plans to be created from predefined rules, using actions from an action base to achieve goals. In agent-based approaches, goals are achieved through planning using various kinds of agents (goal-based, logical, knowledge-based). Goals have also been addressed in requirements engineering. Systems such as KAOS (originally Knowledge Acquisition in autOmated Specification [12]) aim to build a formal proof that the requirements derived for a system meet its goals.

Goals in a policy context are interpreted rather differently. The notion of a policy hierarchy was first identified in [8]. This led to the idea of refining higher-lever policies into lower-level ones. [6] extended this notion into a continuum of policies, with goals being policies at the highest level; however, this is currently just a framework without a concrete implementation. A formal approach to goal refinement using Event Calculus is described in [3]. This has been implemented using KAOS, with refinement patterns being used to decompose goals into subgoals. As another formal approach, [9] uses temporal logic in a two-stage refinement process from goals to subgoals, and subgoals to policies.

In the context of feature interaction, [7] identified the problem of goal conflict in telephony. This approach uses agents to negotiate a plan on behalf of end users to avoid conflicts. This is a different kind of problem from the one tackled in this paper. The User Requirements Notation for goals [2] complements the current paper, and will be studied for possible synergy in future. Fuzzy policies with weights [1] are reminiscent of the nu-

merical approach taken here. Although the work reported in this paper has concentrated on goal refinement as a practical technique, issues of goal conflict have inevitably arisen. These can be regarded as a high-level form of feature interaction. The related notion of policy conflict has been investigated in [10]. As will be seen, goal conflicts arise at various levels as the goals are achieved through refinement and execution.

The work in this paper differs in a number of important respects from other approaches. Although some offline analysis is performed, the bulk of the analysis happens at run time. This is intentional as the choice of policies should depend on the prevailing circumstances. Virtually all other work uses offline techniques, and hence cannot anticipate changes in the system environment.

The approach is based on numerical rather than logical reasoning. This is more flexible in that it does not require any absolute notion of fulfilling goals. Rather, it is possible to satisfy goals only as far as is possible (which may be the only practical possibility). Nearly all other approaches are based on strict logic.

A pragmatic rather than a theoretical approach is followed. Requiring use of formal methods is a considerable barrier for most engineers and designers, and is inappropriate for end users. The need to perform goal refinement at run time is a particular challenge, since most formal techniques require considerable computation. Almost all other goal refinement techniques are formal.

1.4. Paper Outline

Section 2 discusses the goal system architecture, how goals and prototypes are formulated, and the kinds of analysis that are performed to realise goals through policies. In particular, the nature of goal conflict is investigated. Section 3 illustrates the ideas through their application to telephony. Section 4 summarises the approach, and indicates that it has been applied in other application domains.

2. The Goal System

APPEL has been extended to define goals and prototypes. Goal refinement has also been implemented in a package called OGRE (Optimising Goal Refinement Engine). The new goal capabilities have required only small changes in the existing policy system.

2.1. System Architecture

The overall system architecture to support goals and policies is shown in figure 1. For historical reasons, and because the system primarily deals with policies, a number of components are labelled ‘Policy’. However, apart from regular policies, the system also deals with goals, prototypes, resolution policies (that deal with conflicts), and variables (that may be used in goals and policies). The major system components communicate via socket connections (shown as grey arrows in the figure). This gives flexibility in distributing them across processors, though they may all run on one physical machine. The system components have the following functions:

Managed System: the system under control, e.g. a softswitch in Internet telephony [11].

Policy Store: an XML database that stores information about goals and policies [11].

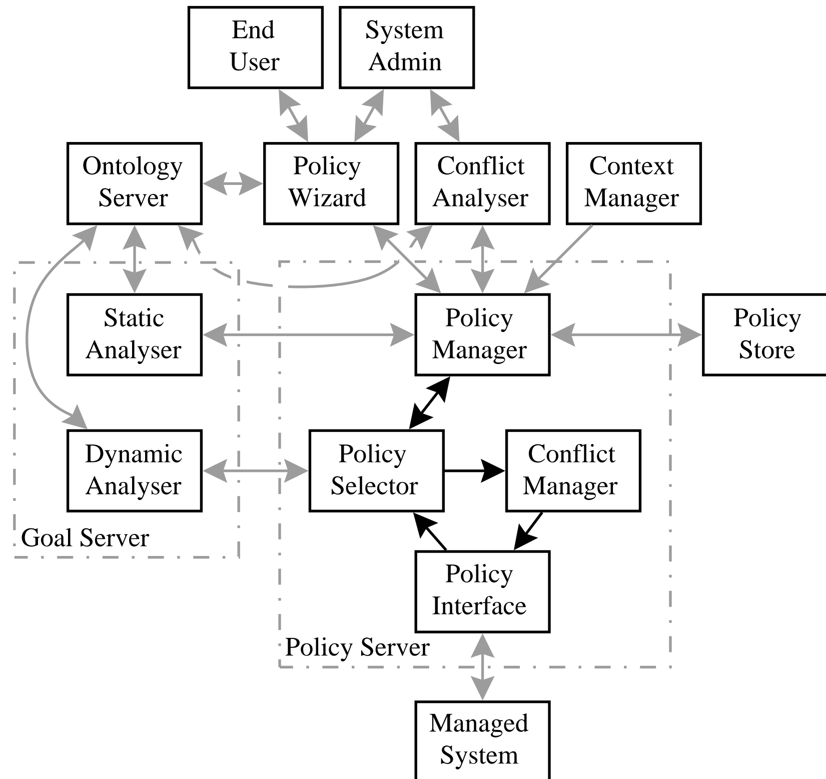

Figure 1. System Architecture

Policy Server: the heart of the policy system [11]. The policy manager is the interface to the policy store, isolating the rest of the system from the particular choice of database. It receives new or updated goals and policies from the policy wizard, and also contextual information from the context manager. When goals or prototypes are modified, the static analyser is notified. This may result in changes to the generated policies. The policy manager is also asked to query the policy store when event triggers are received. These arrive from the managed system and are passed to the policy selector. This chooses relevant policies (i.e. those associated with this trigger and whose conditions are met). If any triggered policies are derived from goals, the dynamic analyser is notified. This produces an optimal set of policies that are submitted to the conflict manager. Conflicts among policy actions are detected and resolved. Finally, a compatible and optimal set of actions is sent via the policy interface to the managed system.

Policy Wizard: a user-friendly interface for defining and editing goals and policies [11]. Besides the original near-natural language wizard, a second wizard has now been created using interactive voice response, and a third using digital pen and paper.

Context Manager: an interface for providing additional information about the managed system (e.g. user availability, schedule or work relationships) [11].

Conflict Analyser: a tool to analyse policies offline for conflict-prone interactions [5].

Ontology Server: a generic interface to ontology-based information about each application domain [4]. A domain-specific ontology is used by the policy wizard to define valid goals and policies, by the offline conflict analyser, and by the goal system both statically and dynamically.

Goal Server: the heart of the goal system. The static analyser is invoked when goals or prototypes are added, modified or deleted (see section 2.4). The dynamic analyser is invoked when goal-derived policies are triggered (see section 2.5).

Although there have been extensions to the ontology server and the policy wizard for the goal system, there has been little change to the other components. The new work has mainly been on the goal server. Besides this, the APPEL policy language and the domain ontologies have been extended to support goal refinement.

2.2. Goals

Currently, all goals must be known locally; future work will handle distributed goals. A goal is a simplified form of policy. There is no trigger because goals always apply. A goal may have a (compound) condition that uses general information like time of day or an environment value. Unlike a policy, a goal has a single action of the form 'maximise(measure)' or 'minimise(measure)'. The measure is a numerical assessment of how well a goal is achieved. Positive goals aim to maximise their associated measure (e.g. network use), while negative goals aim to minimise their measure (e.g. call cost).

A goal measure is like a sub-goal, but is *defined* rather than *refined*. A measure is a formula over relevant system variables. Some variables are held per user (or entity), while others are shared across the system.

Uncontrolled Variables: these are variables that are beyond the control of the policy system, typically 'environmental' factors (e.g. the call cost per second, the frequency of incoming calls).

Controlled Variables: these are variables controlled by the policy system (e.g. the allocated bandwidth, the media used by a call).

Derived Variables: these are pseudo-variables defined in terms of (un)controlled variables (e.g. the call cost as cents/second × duration).

Goal measures and their associated variables are, of course, domain-dependent. Since they are often relatively fixed for each domain, it makes sense to define them once in the domain ontology rather than for each individual goal. A goal definition may therefore refer to these definitions as derived variables.

As a concrete example, the definition below gives a goal that minimises call cost. The elements here are also found in policies. A goal has the following attributes: an identifier (a unique text string), whether it is enabled (allowing deactivation without deletion), when it was changed (date and time in XML format), an owner (the user who defined it), and to whom the goal applies (the stir.com domain here). Two goal conditions are given, combined with 'and': the goal applies only on weekdays (1 = Monday, 5 = Friday) when the call bandwidth exceeds 128 Kbps. At weekends or for low-bandwidth calls, the goal does not apply. The goal action is to minimise call_cost – a derived variable that calculates the cost of a call. (XML schema rules require arguments to be given as attributes arg1, etc.) The following is the XML representation of this goal (omitting the obvious closing tags for brevity):

```
<goal id="Minimise call cost" enabled="true" changed="2009-01-03T17:18:00"
 owner="admin@stir.com" applies_to="@stir.com">
  <policy_rule>
    <conditions>
      <and/>
      <condition> <parameter>day <operator>in <value>1..5
      <condition> <parameter>bandwidth <operator>gt <value>128
    <action arg1="call_cost">minimise(arg1)
```

2.3. Prototypes

Goals are not achievable directly as they are abstract. Instead they are realised through sets of policies. Regular policies can contribute to goals. However, the need to manage goal-related policies leads to defining special prototype policies. These are very similar to regular policies, but are considered separately by the goal system: only prototypes are considered in goal refinement. This gives flexibility in that the policy system may use regular policies only, prototype policies only, or (typically) a combination of both.

Prototypes have an effect attribute that defines how they modify one or more system variables (and thus how they contribute to goal measures). At definition time, this is used to identify the relationships between goals and prototypes. At run time, this is used to determine the set of policies that optimally satisfy the goals. Prototypes are also allowed to have parameters (prefixed by '$') that are chosen at run time by the goal system.

The effect of a prototype is an abstraction of the actions it can perform. More specifically, an effect is defined in the same terms as a goal measure (both being described by a domain ontology). As an example, adding video to a call has several effects: increasing bandwidth, potentially increasing cost, decreasing privacy, etc. Which of these effects needs to be stated depends on what a domain expert defines as important to goals. For example, bandwidth and cost may be considered significant, but privacy (in the context of company communications) may not.

An individual effect names a system variable, an operator, and an expression (e.g. 'call_bandwidth += 128'). The basic operators are '=' (to set a variable), '+=' (to increase it) or '-=' (to reduce it). These are the operators that have been found most useful, though others would be possible (e.g. '*=', '/=').

An interesting situation arises when the effects of multiple prototypes on a goal are considered. Most effects are independent and can be combined (e.g. adding video increases bandwidth, as does adding a digital whiteboard). However, some effects can be permitted only once per execution. For example, forwarding calls reduces the number of calls received, as does rejecting personal calls. If both of these were allowed to contribute to a goal, the number of calls received would be incorrectly reduced twice. There are therefore special 'exclusive' forms of the operators: '+~' and '-~'.

The validity of a set of prototypes is checked against a goal. If their effects are in conflict, that particular combination is not allowed. For example, two prototypes may not fix the same system variable in the same execution. Equally, a prototype making an exclusive change to a system variable (e.g. '+~') is incompatible with any other prototype that affects the same variable. These conflicts are automatically detected by the goal system and are handled by excluding these combinations.

As a concrete example, the definition below gives a parameterised prototype that manages call duration. The elements here are also found in policies. The effect of the

policy is to limit call duration to the value of the duration parameter. Three policy rules are tried in sequence until one with a matching trigger is found. On initial connection, a call timer is started with length duration. On disconnection, the timer is stopped. When the timer expires, the call is forcibly disconnected.

```
<prototype id="Call duration" enabled="true" changed="2009-01-03T17:18:00"
 owner="admin@stir.com" applies_to="@stir.com"
 effect="duration = $duration">
  <policy_rules>
    <sequential/>
    <policy_rule>
      <trigger>connect
      <action arg1="call_timer" arg2="$duration">start_timer(arg1,arg2)
    <policy_rules>
      <sequential/>
      <policy_rule>
        <trigger>disconnect
        <action arg1="call_timer">stop_timer(arg1)
      <policy_rule>
        <trigger arg1="call_timer">timer_expiry(arg1)
        <action>disconnect
```

2.4. Static Analysis

Static analysis in the goal system is activated when a goal or prototype is created, modified or deleted. The policy server informs the static analyser of the relevant identifier. When a new goal is defined, the static analyser retrieves existing prototypes and evaluates their effects against the goal measure. (A new prototype is analysed similarly.)

Whether a prototype contributes to a goal is determined by comparing its effects with how the goal measure is defined (i.e. which system variables it uses). A prototype is considered to contribute to a goal if it affects one or more system variables involved in the measure. The prototype effect may modify an arbitrarily complex measure. The sense of the effect will therefore not be known until run time, when it may worsen or improve the evaluation of including this prototype.

For each prototype, a set of the goals it contributes to is created. This makes use of information in a domain-specific ontology about system variables. Typically a prototype contributes to one or more goals. If this set is empty because there are no relevant goals, this is not an error. A library of useful prototypes is normally created by a domain expert. When goals are defined, only some prototypes may be relevant. But as future goals may require other prototypes, they are all appropriate.

The prototype is then instantiated as a regular policy. This is a copy of the prototype, with the <**prototype**> tag replaced by <**policy**>, and with a unique new identifier derived from the original one. To the rest of the policy system, this generated policy looks like any other. The policy server will therefore apply the policy at run time exactly as normal. However, it is necessary to identify such policies through a supports_goal attribute that only goal-derived policies carry. This allows the static analyser to deal with modification or deletion of a goal or prototype: the policies generated from these may need to be altered with regard to the goals they support, or may need to be deleted if these goals no longer exist.

Prototype and goal conditions are normally copied into a policy in a form such as: prototype_condition **and** (goal_condition_1 **or** goal_condition_2). This ensures that the generated policy will be triggered only if the prototype conditions are satisfied *and also* the associated goal conditions. Some goals that a prototype contributes to may not have conditions; these have conditions that are implicitly true. Absence of a goal condition therefore means that the whole set of goal conditions will always hold. In such a case, the goal conditions are not added to the prototype as they are superfluous.

When a prototype is instantiated as a policy, it retains any prototype parameters. These become relevant only during dynamic analysis.

2.5. Dynamic Analysis

Dynamic analysis in the goal system is activated when an event trigger leads to a set of policies, i.e. those whose triggers match the event and whose conditions are satisfied. This set may contain goal-derived policies (identified through their supports_goal attribute). If no policies are goal-derived, the policy server continues policy execution as normal. However if there are such policies, their identifiers are sent to the dynamic analyser.

The dynamic analyser now chooses the subset of these policies that optimises the overall goal evaluation function. This function can refer to any system variables, including those that vary dynamically (such as the current call bandwidth or the number of calls received so far). This means that the most appropriate set of policies can be selected for the current circumstances. It also means that the selection of policies may vary as the system evolves over time. For example, if the system load increases significantly or the call cost changes on entering a different time period, the goal system will automatically adapt to the changed circumstances. The dynamic approach is much more flexible than the offline, logic-based techniques used in most other approaches.

At this point, any optimisation algorithm can be used (e.g. hill-climbing, the simplex algorithm, a genetic algorithm). In optimisation terminology, the evaluation function is the objective or fitness function. The goal system is designed to use any optimisation algorithm that conforms to a defined interface.

It is possible during optimisation to discover that more than one combination of policies leads to the same optimal value. Typically this happens when two prototypes have exactly the same exclusive effect: choosing either gives the same numerical evaluation. In fact this is a kind of conflict, and the dynamic analysis reports it as a probable error. A more subtle issue is when the optimal solution is unique, but a number of near-optimal solutions are also available. This is considered further in section 3.5.

Currently, and for practical reasons, a simple optimisation algorithm is used: the overall evaluation function is calculated for all possible combinations of policies. This algorithm is exponential in the number of policies, and so is impracticable if there are many policies. However, in practice this has not been found to be a problem. The policy system can handle hundreds or thousands of policies. Even if these are all derived from goals, that is not in itself problematic. What matters is how many such policies are triggered *at the same time*.

From the potentially large number of policies, triggering typically selects only about half a dozen policies. This happens because eligible policies must be for the relevant user (or entity), must have a matching trigger, and must have satisfied conditions. In

telephony, for example, only caller and callee policies are selected (plus those for the domain they belong to); this substantially reduces the number of eligible policies. Telephony policies are triggered only on call setup, mid-call, or on call clear-down; again, this significantly cuts down on relevant policies. Finally, telephony policies usually have conditions (e.g. time of day, who the caller is) that further restrict the selection. Other domains (e.g. home care, sensor networks) work with similarly reduced sets of policies.

The result is that the dynamic analyser usually has to optimise only a modest set of policies. For up to a dozen policies, say, the simple exhaustive algorithm is more efficient than more elaborate algorithms that are best for large optimisation problems. However, if other application domains lead to the need for optimising large sets of policies, then a different optimisation algorithm could readily be slotted in.

Optimisation deals with one other aspect: prototype parameters. These are left uninstantiated when policies are created from prototypes. At run time, the dynamic analyser considers optimal choices for parameters as well as for policies. Exactly the same mechanism applies: the choice of parameters modifies the effects and therefore the overall evaluation. However, parameters have to be optimised in a different way. Given reasonable restrictions on the evaluation function (monotonicity), only the extremal values of parameters need be considered. If call bandwidth is a parameter, for example, then only the minimum and maximum permissible bandwidths should be tried: one or other will achieve the optimal value. The optimisation also allows for parameters being drawn from a discrete but ordered set (e.g. audio, whiteboard, video) rather than from a range. Knowledge about extreme values is domain-specific, and is defined in a domain ontology.

Instantiating the parameters of a policy leads to a new temporary policy that applies only during processing of the current trigger. This policy is created by copying the parameterised policy, substituting the optimal parameter values, and giving the policy a unique identifier.

The selection of policies (and parameters) with the highest evaluation determines the identifiers returned to the policy server for execution. Typically only a subset of the goal-derived policies is eligible. The subset is combined with the eligible regular policies and processed as normal by the policy server. This requires the set of actions dictated by these policies to be checked for conflict. Resolution policies detect conflicts and determine a compatible set of actions (e.g. disregarding lower-priority policies) [10].

2.6. Identifying Goal Conflicts

Now that the basis of goal refinement has been explained, it is possible to be precise about the ways in which goals can conflict with each other. Although these have been mentioned in passing, they are collected here for reference.

Conflicts do not really arise during static goal analysis (i.e. when goals are defined). However, it is possible to find that a goal cannot be achieved by any prototype (or can be only partially satisfied). This happens if prototypes cannot be found to affect all the controlled variables in a goal measure. This is probably not a conflict, though it reflects an inconsistency between the goals and the prototypes available to achieve them. However it is an error, and is reported by the static analyser.

Conflicts do arise during dynamic goal analysis (i.e. when goal-derived policies are processed). It may be found that certain goal-derived policies cannot be considered in combination (because they affect system variables inconsistently, or they make exclusive

changes to a system variable). This is a conflict that is resolved by excluding the impermissible combinations. It may also be found that different combinations of goal-derived policies yield the same optimal value. This indicates that the prototypes are too weakly characterised in their effects and cannot be distinguished. This is a kind of conflict, and is reported by the dynamic analyser.

Following dynamic goal analysis, the resulting policies may be found to have conflicting actions. Although this is part of the standard conflict detection performed by the policy server, it is a consequence of goal refinement and reflects an indirect conflict among the goals. The authors have developed a technique and tool for filtering conflict-prone policies [5]. In future work, this will be adapted to offline analysis of goals and prototypes as well as policies.

There is an interesting relationship among conflicts found through offline filtering, through dynamic goal analysis, and through online detection of action conflicts. Essentially all three techniques look for incompatibilities. It is possible to detect the same kinds of conflicts at all three levels. However, this is not normally desirable because it would require effects to be considered at a low level of abstraction (i.e. the level of individual policy actions). Both filtering rules and goal measures should be formulated more abstractly, implying that action conflicts will usually be in addition to the kinds of conflicts detected at higher levels.

3. Application to Internet Telephony

To illustrate how goals are supported, this section gives concrete examples of how a hypothetical company might wish to optimise its voice communications. For this application, the managed system is a softswitch such as SER (SIP Express Router, *www.iptel.org/ser*). The XML representation of goals and prototypes was illustrated in sections 2.2 and 2.3. This section uses pseudo-code for readability (and is similar to the web-based wizard). Triggers are indicated by **when**, conditions by **if**, and actions by **do**.

3.1. Goals

For the telephony system, suppose that the uncontrolled variables are rate (call cost, cents/second) and quality (a subjective measure, 1 = poor to 10 = best). The controlled variables are bandwidth (Kbps), duration (typical call length, seconds), handled (calls handled by the user, per day), and received (calls received by the user, per day).

The company is presumed to have the communications goals shown in table 1. This is a subset of a larger set of goals that may deal with other factors such as availability, reliability, security, etc. The textual descriptions of the goals are high-level and rather woolly. To make them operational, they need to be related to the system variables. Whether the company's goals are appropriately formulated here is not relevant. What is important is that the interpretation of each goal is made explicit and precise.

Each goal is associated with a numerical measure (network use being split into sub-measures for bandwidth use and call handling). The goal measures are defined as derived variables. These are combined into an overall evaluation function that is a weighted sum of the goal measures, again treating network use as two subsidiary measures. Note that measures to be minimised are preceded by a minus sign. For a discussion of where the scaling factors and weights here come from, see section 3.5.

Goal	Description
1	the company is keen to promote the use of multimedia in its communications **if** it is a weekday **do** maximise multimedia use **measure** 0.03×bandwidth×quality
2	the company wishes to use its communications network to the full **do** maximise network use **measure** 0.0008×bandwidth×duration **and** 6.0×handled
3	call cost can become important so the company wishes to control this **if** it is a weekday **and** the call bandwidth exceeds 128 Kbps **do** minimise call cost **measure** rate×duration
4	the company wishes to minimise interruptions to staff during lunch breaks **if** it is 1PM–2PM **do** minimise interruption time **measure** 0.02×received×duration

Table 1. Sample Goals

3.2. Prototypes

The prototype library will include a wide range of policies to meet the company's likely goals. For the sake of example, suppose that the prototypes in table 2 are those relevant to a call arriving or a call being made.

3.3. Static Analysis

As each of the goals in section 3.1 is defined, the prototypes that contribute to it are determined. The prototypes in section 3.2 are a subset of the full library; the table below shows their static analysis. The information in such a table is used to create a goal-derived policy for each prototype, stating which goals a policy contributes to:

	Prototype						
Goal	1	2	3	4	5	6	7
1	✓	✓					✓
2	✓	✓		✓	✓	✓	✓
3				✓	✓		
4			✓	✓	✓	✓	

3.4. Dynamic Analysis

As telephony events are received from the softswitch, they trigger the selection of regular policies as well as those generated during static analysis of goals. Suppose that there are only the policies derived from the prototypes in table 2. The weight associated with each goal measure is taken as 1.0 (see later discussion in section 3.5).

Imagine that a long-distance call now arrives about a contract, that the call bandwidth is low, and that the callee is available. The policy server will determine that policies 1 (add lawyer), 2 (add video), 4 (5 minute limit), 5 (parameterised limit) and 7 (remove whiteboard) are eligible for execution. This will be notified to the dynamic anal-

Prototype	Description
1	a lawyer must be conferenced into calls about contracts **when** a call arrives **if** it is about contracts **do** conference in a company lawyer **effect** bandwidth += 64
2	low-bandwidth calls should make use of video **when** a call arrives **if** the bandwidth is less than 1 Mbps **and** the call has no video **do** add video **effect** bandwidth += 512
3	if the callee is busy, forward the call to a secretary (note '-~' for an exclusive change) **when** a call arrives **and** the callee is busy **do** forward the call to a secretary **effect** received -~1
4	expensive calls should be limited to 5 minutes **when** a call is made **if** the cost exceeds 0.2 cents/sec **do** start a timer to limit the call to 300 sec **effect** duration = 300
5	calls should be limited by a duration parameter that is dynamically optimised **when** there is a call **do** start a timer that limits the call to $duration seconds **effect** duration = $duration
6	if the callee is busy for a personal call, reject the call with a suitable message **when** a call arrives **and** the callee is busy **if** it is a personal call **do** reject the call **and** play a busy message **effect** handled -~1 **and** received -~1
7	remove a digital whiteboard if it is in use **when** there is a call **if** a whiteboard is in use **do** remove whiteboard **effect** bandwidth -~96

Table 2. Sample Prototypes

yser, which will retrieve the current values of the system variables. Suppose these are bandwidth 64, duration 300, handled 5, quality 8, rate 0.1, and received 5.

The dynamic analyser will then report that the optimal set of policies is 1 (add lawyer), 2 (add video) and 5 (duration parameter 600 seconds). The actions from these policies are then checked for conflicts. A resolution policy may determine that conferencing in a third party (policy 1) conflicts with also adding video (policy 2). This is because use of video on an external call may compromise the confidentiality of the workplace. The resolution is to conference in a manager to monitor what is viewed. The actions sent to the softswitch are therefore: include a lawyer (from 1), add video (from 2), run a call timer for 600 seconds (from 5), and include a manager (from the resolution).

As can be seen, a substantial number of things happen for a simple event like a call arriving. The entire procedure of selecting policies, optimising goals, resolving conflicts,

and dictating actions takes about two seconds: one second in the policy server, and one second in the goal server. If goals are not used, only the policy server is involved. As a once-off overhead on call setup, this is believed to be acceptable – especially given the considerable flexibility and control that goals and policies offer.

However, the authors expect to be able to reduce this overhead in a number of ways. The entire system is written in interpreted Java. Profiling will identify key methods that can be made more efficient (and possibly implemented in C for use via JNI). The system could also cache key system variables, triggers and resulting actions to learn responses to particular circumstances. This would allow a cached response to be used instead of invoking the entire analysis procedure. Ontology queries could also be cached.

3.5. Defining Goal Weights

The goal measures given in table 1 are defensible, e.g. bandwidth $\times$ duration is a good measure of bandwidth use, and rate $\times$ duration is a good measure of call cost. Equally, the prototype effects given in table 2 are defensible as they are based on domain knowledge, e.g. of how different media are handled.

What is much less obvious is what scaling factors and weights are appropriate. At first sight, factors like 0.03 or weights like 1.0 might seem completely arbitrary. The absolute value of the overall evaluation function is not, of course, important – only relative values determine the outcome of optimisation.

Scaling factors are used because the units used for system variables are very different: bandwidth in Kbps, duration in seconds, etc. Without scaling factors, goal measures would yield widely varying numerical values. Instead, each goal measure is associated with a scaling factor. This can be chosen automatically so that different measures have similar values for a typical set of system variables (e.g. those assumed in section 3.4).

Each goal measure is then given a weight relative to others. The goal analysis depends on these weights. The question is whether it depends *critically* on these weights. If it did, then the choice of weights would be difficult. The outcome of goal analysis could also be unstable: small changes in weights might have a dramatic effect on the selected policies. The goal system therefore includes an automated sensitivity analysis to help the designer investigate what ranges of weights give stable results. There is also a simulator for the policy system that allows key test cases to be evaluated to check for unexpected surprises in how the system behaves.

Figure 2 shows part of the automated sensitivity analysis, looking at the effect of a system variable on the evaluation function. Here, the call rate is varied over quite a wide range: from 0.05 to 1.0 cents/second. A sample situation was considered in section 3.4, where the call rate was 0.1. It was found that the optimal set of policies was 1 (add lawyer), 2 (add video) and 5 (call limit parameterised for 600 seconds). From figure 2, it will be seen that these remain the optimal policies until the call rate rises to 0.4. For higher rate values the same policies remain optimal, but the call limit should be made short (60 seconds). Such call rates might be typical of a satellite call, where goal 3 (minimise cost) becomes predominant. The goal analysis thus yields a stable set of results with respect to call rate, differentiated only in duration limit for very high call rates.

Figure 2 shows another part of the automated sensitivity analysis, looking at the effect of a weight on the evaluation function. Here, the bandwidth weight is varied over quite a wide range: from 0.05 to 10.0. Again, the same optimal policies are generated

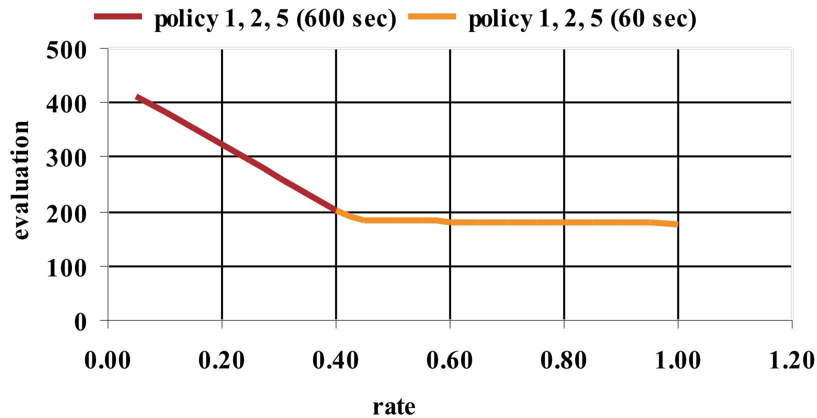

Figure 2. Evaluation of Different Call Rates

in almost all cases. Only for very low weights (below 0.35) is another optimal choice made – and again this differs only in having a short call timer. It can be concluded that the goal analysis yields a stable set of results with respect to bandwidth weight. A range of bandwidth weights could be used, but 1.0 is satisfactory.

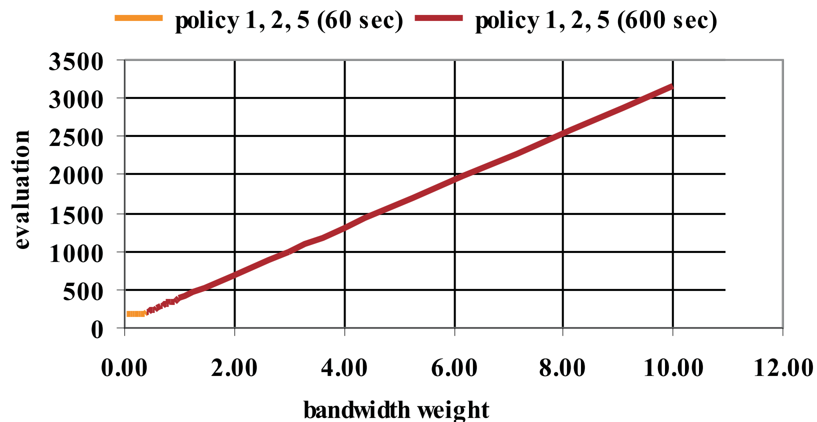

Figure 3. Evaluation of Different Bandwidth Weights

The sensitivity analysis also investigates changing multiple factors. For example, the weights for call cost and interruption time goals were separately and automatically varied over the range 0.05 to 10.0. The same optimal solution is generated as long as the call cost weight is less than 1.3; above this, a short call timer is imposed. Overall, the automated sensitivity analysis confirms that the scaling factors, the weights, and the stability of the goal analysis are appropriate. What might have appeared to be arbitrary choices are therefore defensible.

4. Conclusion

It has been seen how goal refinement into policies can be formulated as a numerical optimisation problem. The achievement of goals is assessed through measures that are de-

fined in terms of system variables (uncontrolled, controlled and derived). The contribution of prototypes to goals is made through their effects on system variables. When goals and prototypes are defined or altered, static analysis determines the relationship among these. This creates regular policies that are linked to the goals they support.

When events trigger the selection of policies, an optimal selection is generated of goal-derived policies. The dynamic nature of the analysis means that goals are achieved according to the current circumstances, and not to meet purely static criteria.

Goal conflict arises at a number of levels. Errors may arise during static analysis, dynamic analysis, or action analysis. Automated sensitivity analysis shows the approach yields stable results, with values readily chosen for scaling factors and goal weights.

The approach has been illustrated in the telephony domain. However, the techniques and tools are multi-purpose. For example, they have also been used to support goals and policies for managing sensor networks, wind farms, and home care systems. It is therefore believed that the approach is general and will find value in many domains.

Acknowledgements

The authors thank their colleagues on the PROSEN project for discussions that helped mould the approach. Gavin Campbell created goal support for PROSEN, supported by grant C014804 from the UK Engineering and Physical Sciences Research Council.

References

[1] M. Amer, A. Karmouch, T. Gray, and S. Mankovskii. Feature interaction resolution using fuzzy policies. *Proc. 6th ICFI*, pages 94–112. IOS Press, May 2000.

[2] D. Amyot. Goal-oriented requirement language (GRL) and its applications. In *Proc. 31st Int. Conf. on Software Engineering*. IEEE, May 2009.

[3] A. K. Bandara, E. C. Lupu, J. D. Moffett, and A. Russo. A goal-based approach to policy refinement. In *Proc. Policy Workshop*, pages 229–239. IEEE, 2004.

[4] G. A. Campbell and K. J. Turner. Ontologies to support call control policies. *Proc. 3rd Advanced Int. Conf. on Telecomms.*, pages 5.1–5.6. IEEE, May 2007.

[5] G. A. Campbell and K. J. Turner. Policy conflict filtering for call control. *Proc. 9th ICFI*, pages 93–108, IOS Press, May 2008.

[6] S. Davy, B. Jennings, and J. Strassner. The policy continuum – Policy authoring and conflict analysis. *Computer Communications*, 2008.

[7] N. D. Griffeth and H. Velthuijsen. Reasoning about goals to resolve conflicts. In *Int. Conf. on Intelligent and Cooperative Information Systems*, pages 1–14, May 1993.

[8] J. D. Moffett and M. S. Sloman. Policy hierarchies for distributed systems management. *IEEE JSAC*, 11(9):1404–1414, 1993.

[9] J. Rubio-Loyola *et al.* Using linear temporal model checking for goal-oriented policy refinement frameworks. In *Proc. Policy Workshop*, pages 181–190. IEEE, 2005.

[10] K. J. Turner and L. Blair. Policies and conflicts in call control. *Computer Networks*, 51(2):496–514, Feb. 2007.

[11] K. J. Turner *et al.* Policy support for call control. *Computer Standards and Interfaces*, 28(6):635–649, June 2006.

[12] A. van Lamsweerde and E. Letier. From object orientation to goal orientation. In *Proc. Radical Innovations of Software and Systems Engineering in The Future*, LNCS 2941, pages 153–166, Mar. 2003.