Policy Conflicts in Home Care Systems

Feng Wang and Kenneth J. Turner

Computing Science and Mathematics, University of Stirling, Stirling, FK9 4LA, UK

fw@cs.stir.ac.uk, kjt@cs.stir.ac.uk

Abstract. Technology to support care at home is a promising alternative to traditional approaches. However, home care systems present significant technical challenges. For example, it is difficult to make such systems flexible, adaptable, and controllable by users. The authors have created a prototype system that uses policy-based management of home care services. Conflict detection and resolution for home care policies have been investigated. We identify three types of conflicts in policy-based home care systems: conflicts that result from apparently separate triggers, conflicts among policies of multiple stakeholders, and conflicts resulting from apparently unrelated actions. We systematically analyse the types of policy conflicts, and propose solutions to enhance our existing policy language and policy system to tackle these conflicts. The enhanced solutions are illustrated through examples.

Keywords: Policy-based management, policy conflict, home care system.

1   Introduction

Policies have emerged as a promising and more flexible alternative to features. Among the benefits of policies, they are much more user-oriented. However, policies are prone to conflicts much as features are prone to interactions. This paper examines the issues of policy conflict in a novel application domain: home care.

It is predicted that the growing percentage of older people will have enormous impact on the demand for care services. This will exert huge pressures on the resources of existing care services [3]. Increasingly, providing care at home is seen as a promising alternative to traditional healthcare solutions. By making use of sensors, home networks and communications, older people can prolong independent living in their own homes. Remaining in a familiar environment while being taken care of also improves

their quality of life. Their families and informal carers can also be relieved of constant worry whether those in care are well.

The hardware to enable home care services, such as sensor technologies and communications, has matured in terms of cost and availability. Providing software solutions to deliver home care service, however, remains a challenging task. Most home care systems have been created in an *ad hoc* way. The systems are usually hand-crafted and manually customised to the needs of individual scenarios. Because the solutions for home care services are hard-coded, even simple changes in services requires an on-site visit by specially trained personnel. They are therefore costly to change.

Proprietary, off-the-shelf telecare products suffer from similar problems. The functions of a product are typically fixed in special-purpose devices. Data from these devices cannot easily be accessed, and the devices work only with products from the same company. Domestic health monitoring and home automation are currently very limited.

The major issues in home care delivery are flexibility, adaptability, customisability and cost. We have successfully demonstrated that it is possible to use a policy-based system to integrate data from a variety of home sensors. Sensor data is used to support a variety of home automation and home care services [3]. Considerable research remains to realise the potential of this work and to demonstrate its value in supporting care of older people. One major issue is the detection and resolution of policy conflicts, which is the focus of this paper.

Essentially, policies are rules that define the behaviour of a system. A typical policy consists of a trigger, a condition and an action. There are two basic types of policies: authorization policies and obligation policies. Authorization policies give a set of subjects the authority to carry out some actions upon a set of target objects; in negative form, they require subjects to refrain from doing so. Obligation policies specify that a set of subjects is responsible for taking some actions upon the target objects when a certain trigger event is received and the some conditions are satisfied.

When enforcing the policies, it is possible that multiple policies may conflict with each other. We use the following general definition of *policy conflict*: two policies are said to conflict with each other if there is inconsistency between them. The classification of conflicts by Moffet et al. [1] is discussed later.

When applying policy-based management to home care systems, we observe that certain classes of policy interaction are unique to this domain:

- policy rules of multiple stakeholders may conflict

- policy actions resulting from apparently different triggers may interact according to changing *situations*
- policy actions may conflict over time.

The issues in a policy-based home care system are as follows. What types of the policy interactions should be tackled inside the policy system? What type of interactions should be tackled outside the policy system? If the policy interaction is tackled by the policy system, how should it be handled?

Based on the analysis of the problems, we propose a solution to tackle the above issues. Our solution is built on top of our previous work on the ACCENT project [4]. In order to explain our solution, we will first introduce the previous work on ACCENT. The paper is organized as follows. Section 2 briefly describes the policy language for home care. Section 3 presents how policies are deployed and enforced inside the home care system. In section 4, policy conflict issues in home care systems are identified and analysed. A solution for resolving these conflicts is proposed in section 5. Related work is discussed in section 6. Finally, in section 7 we describe the current status of the work.

2   Policy Language for Home Care Systems

The policy language for care at home builds on our previous work for call control [4]. To allow use in different application domains, the policy language has two parts:
- the domain-independent core policy language defines the structure of policies (e.g. their combinations) and general attributes of policies (e.g. metadata)
- domain-specific extensions reflect specialisations for each kind of application.

A policy rule consists of three parts: trigger, condition and action. Although the core language defines some of these, specific elements are normally defined per-domain. The core policy language is defined in [4], with its specialization for home care in [3].

A home care policy consists of a set of policy attributes and a set of policy rules. The attributes of a home care policy include the following:
- *id* uniquely identifies the policy in the policy store.
- *description* explains the purpose of the policy in plain text.
- *owner* indicates the entity that the policy belongs to. A notation similar to email addresses is used, e.g. *tom@house1.address2.city3.country4*.
- *applies_ to* identifies the entities to which a policy applies (e.g. sensors, people, virtual entities 1 computer programs). An email-like notation is used for entities:

1@movement.kitchen.house1.address2.city3.country4 means movement sensor *1* in the kitchen of *house1*. Omitting '1' means any movement sensor in this kitchen.

- *preference* states how strongly the policy definer feels about it, and represents the modality of the policy. Examples are *should*, *should not*. Internally the value of this attribute is represented as integer (which may be positive or negative) .
- *valid_from* and *valid_to* specify the time period during which a policy is valid
- *profile* is used to group the policies. A policy with an empty profile is always applicable, while one with a non-empty profile must match the user's current profile.
- *enabled* states whether the policy system should consider a policy or not.
- *changed* indicates the last-modified time of a policy.

For home care policy rules, a generic trigger *device_in* is used: its arguments indicate the trigger type and the sensor that caused it. A trigger sets environment variables to reflect the current state of the environment. A policy condition can make use of these variables to check whether it is eligible for execution. A generic action *device_out* is defined to instruct actuators to execute actions. This action has arguments to indicate the actuator, the action to be executed and the parameters of the action.

In our home care system, a home care service is a rule-based application described by policy rules. An example policy for home care is shown in Figure 1. Dementia patients often wander at night, and this worries their relatives. The policy in figure 1 states that, if movement is detected in the bedroom when it is night (10PM–7AM), remind the patient to go back to bed. The obvious closing tags are omitted in the XML definition.

```
<policy_rule>
    <trigger arg1="bedroom" arg2="movement">
        device_in(arg1,arg2)
    <condition>
        <parameter>time
        <operator>in
        <value>22:00:00..07:00:00
    <action arg1=":bedroom_speaker"
      arg2="please go to bed">speak(arg1,arg2)
```

Figure 1.   Night-Time Wandering Reminder Policy Example

3 Deployment and Enforcement of Policies

Figure 2 illustrates how policies are deployed and enforced in the policy system.

Error! Objects cannot be created from editing field codes.

Figure 2. Policy Deployment and Enforcement in A Home Care System

3.1 Policy Deployment

At design time, a policy is defined using editing tools such as a policy wizard. This policy is then passed to the policy deployment module (step 1). Since the policy may conflict with the existing policies in the policy store, it is passed to the static analysis module to check for conflicts (step 2). The static analysis module retrieves related policies from the policy store (step 3), performs conflict detection analysis, and returns the result to the policy deployment module (step 4). If there is conflict, the user is notified. If there is no conflict, the policy is saved in the policy store (step 5).

3.2 Policy Enforcement

The policy enforcement module makes decisions on which actions should be executed and issues them for execution. At run time, a sensor sends out an event through the event service (step 1). The event is passed to the policy enforcement module (step 2). The policy enforcement module retrieves relevant policies from the policy store (step 3). For each retrieved policy, the policy enforcement module checks the trigger part and the condition part of the policy against input triggers and the current environment setting. If the trigger matches and the policy conditions hold, the corresponding policy action is added to the set of potential actions. Once the policy enforcement module finishes checking the relevant policies, there will be a set of potential actions. If there is more than one action in the set, this is passed to the dynamic analysis module for detection and resolution of conflicts (step 4).

Our existing policy system uses resolution policies to detect and resolve conflicts among actions. The structure of a resolution policy is very similar to that of an ordinary policy. The major difference is that in the resolution policy, the triggers are the actions of regular policies rather than ordinary triggers from sensors. A detailed description of resolution policies can be found in [5]. The dynamic analysis module retrieves resolution policies from the policy store (step 5), and applies these to the set of potential actions to select the most appropriate actions if there are conflicts. This action is then passed back to the policy enforcement module (step 6). The policy enforcement module

sends the actions to the event service (step 7). The event service acts as a broker, passing commands to actuators for execution (step 8).

A resolution policy supports two types of resolution actions: specific actions and generic actions. For specific actions, a resolution policy specifies what to do when there are conflicting actions. The outcome is not limited to the set of conflicting actions. For generic actions, the resolution is chosen from among the conflicting actions. This relies on comparing the attributes of conflicting policies. Borrowing from our previous work, the following generic actions are used in home care:

- *apply_newer, apply_older:* decides whether the newer or older policy is chosen.
- *apply_one*: chooses some action from the set of potential actions.
- *apply_negative, apply_positive, apply_stronger, apply_weaker*: decides the action by checking the value of policy *preferences*.
- *apply_inferior, apply_superior*: uses the *applies_to* attribute to decide within one hierarchy whether the superior's policy or inferior's policy is chosen

4   Policy Conflicts in Home Care Systems

4.1. Detection and Resolution of Policy Conflicts in General

To simplify our analysis, for now we only consider a policy with a single rule. A home care policy has the following elements: subject, target, trigger, condition, action, owner and preference. Much as for Ponder (*http://ponder2.net*) we consider two types of policies: authorisation (A) and obligation (O). For authorisation policies, the subject is authorised to take an action on a target object. For obligation policies, the subject is obliged to take action on the target when receiving a trigger and the condition is satisfied. If we combine the type of policy with the modality, we get the following policy modes: positive authorisation (A+), negative authorisation (A-), positive obligation (O+), and negative obligation (O-).

Type of Policy Conflicts in Home Care. According to Moffet and Lupu's classification [1] [6], there are two types of policy conflicts: modality conflicts and goal conflicts.

Modality conflicts can be detected by looking at the policy alone. For modality conflicts, the following attributes (subject, target, action) of two policies overlap, but

the mode of the policy contradicts. The other attributes of the policy may be different, including trigger, condition and owner. There are three possible modality conflicts:

- A+, A- : one policy states that the subject is authorised to take some action, but the other policy prohibits the subject from performing this action.
- O+, O- : one *policy* states that the subject is obliged to take some action, but the other policy states that the subject is obliged not to take this action.
- A-, O+ : one *policy* states that the subject is obliged to take some action, but the other policy states that the subject is not authorised to do this.

A+/O- is not a policy conflict, since no actions result from this combination: the subject is *authorized* to take some actions, but must *refrain* from taking these [1] [6].

Goal conflicts need application-specific information to be detected. Moffet *et al*. [1] identify four types of conflicts: conflicts of imperative goals, in particular for resources; conflicts of authority goals, including conflict of duty and conflict of interest; multiple managers; and self-management.

In the home care domain, we consider actuators as the target of policy. A sensor, person or computer program is considered as a subject of policy-based management. Inside the policy system, there will be an agent for each such entity to act on its behalf. Due to lack of computation power on sensors, our policy system employs a centralised server for enforcing policies. This implies that the policy server acts as an agent for all subjects of the policy system.

In home care systems, modality conflicts may arise in one owner's policies due to overlapping situations. They also may arise between multiple owners' policies. For goal conflicts, we are currently particularly interested in multiple managers and conflicts for resources since many care services are represented as obligation policies. These services are triggered by events from sensors.

Detection Conflicts: Statically vs. Dynamically. Modality conflicts can be detected at definition time or at run time. Detection is achieved by comparing the subject, target, action and preference of two policies. This indicates whether there are potential conflicts. For modality conflicts, if the situations of two policies are exactly the same, this potential conflict becomes definite; the conflict should be eliminated at definition time. If conflict depends on the evaluation of a run-time situation, this type of potential conflict may still need to be detected and resolved at run-time.

Detecting goal conflicts needs application-specific information. This may use an explicit definition of conflict situations. It may also use automatic reasoning about the

effects on goals if the semantics of these is properly specified. Our policy system supports the specification of conflicting situations by the user.

As seen in section 3, our policy system supports both static analysis and dynamic analysis. Dynamic analysis is performed when a trigger from sensors is received and processed. It requires resources and time, and may slow down the decision making process of the policy system. Comparing with dynamic analysis, static analysis is more desirable as it reduces the burden on dynamic conflict. However, not all conflicts can be detected by static analysis, especially potential conflicts.

Resolving Conflicts. Once a conflict is detected, conflicting policies need to be resolved. This can be by achieved by notifying the user and asking for a decision, or it can be done automatically.

For automated resolution, our policy system supports both specific actions and generic actions. For policies that belong to a single owner, several policy attributes can be used to choose the resolution action. For example, the policy language supports choosing an action with the strongest preference. For policies that belong to different users in one organization, the 'distance' between a policy and the managed object can be used to choose the resolution action. In our policy language, this distance is derived from the *applies_to* attribute. Suppose one policy applies to *@cs.stir.ac.uk* and the other policy applies to *john@cs.stir.ac.uk*. An *apply_superior* resolution action will choose the policy which applies to *@cs.stir.ac.uk* as the higher domain in the hierarchy.

4.2 Special Issues for Policy Conflicts in Home Care

In home care systems, beside the modality conflicts discussed above we observe the following three special types of conflicts between policy actions. How to deal with these conflicts is the focus of this paper. Should the interaction be tackled inside the policy system, or should it be dealt with outside the policy system (e.g. by the actuators)? If the interaction is tackled by the policy server, how can we enhance our existing policy system to handle it? If the interaction is tackled outside the policy server, what functionalities are required from the external system?

Dependency among Situations. The actions resulting from different situations may conflict with each other, and situations may have interdependencies. The situation of obligation is common in a home setting. These situations rely on context information. As Dey points out [7], there are different levels of context information. High-level situations can be inferred from low-level sensor data, and the trigger from one sensor can be used to infer multiple situations. As an example, a 'door open' sensor can detect the situation of the door being left open. Suppose a policy states that when the front door is left open, a reminder should be given to the resident to close the door. Combining the door sensor and the sensor in the door lock, a new situation can be detected: the door has been broken open. Suppose another policy states that, when the door is broken open, the resident should be advised stay in his/her room to call for help. When a door is broken into, which action should be taken [8]?

In the above case, if we specify the two situations as two separate triggers, there will be no policy conflict and two actions will be executed. It does not make sense to remind the user to lock the door, while at the same time stating that there has been a break-in.

In this example, we can see that situations in home care can have logic relationships between them. One situation may be implied by another, or two situations may be implied by triggers originating from the same sensor. Besides logical relationships, there are other relationships such as containment. For example, one policy reacts to movement in the bedroom, while another reacts to movement in any room of the house. The situation of the second policy contains that of the first.

A policy system needs to be able to detect and resolve policy conflicts due to dependent situations. The issue is how to specify the triggers and conditions of the policies properly so that the conflicts are detected.

Multiple Stakeholders. In home care systems, policies may be defined by multiple organizations (e.g. a social work department, a surgery, a clinic). Their policies may conflict with each other. How can the conflicts of multiple stakeholders be handled?

In fact, conflicts among multiple stakeholders are not much different from the case of a single stakeholder. The difference is in resolution of the conflicts. If the resolution action is chosen from one of the conflicting actions, dealing with multiple stakeholders is an issue. When policies are defined by different organizations, there is no hierarchy among the stakeholders. Some solution is needed to decide how one stakeholder's policies should be evaluated compared to other stakeholders' policies.

Interactions between Actions over Time. The policy actions in a home care system take time to complete. It is therefore possible for new actions to conflict with ongoing actions. Suppose a medical reminder service can alert a patient to take medicine at certain times. This system will remind the patient again if there is no response to the first reminder. While the medical reminder is running, a more urgent situation such as a fire may be detected in the house. Following the fire alarm policy, the system will remind the user to leave the house immediately.

How to deal with these interactions in a policy-based system? A fundamental question is whether they should be dealt within the policy system or not.

5 Enhancement to the Home Care Policy Systems

5.1 Tackling Dependencies among Situations

A situation is specified jointly by the trigger and the condition of a policy. To tackle dependencies among situations, we introduce a situation dependency graph (see figure 4). The nodes on the left are the sensors. The nodes in the middle and on the right are situation nodes. A situation node receives inputs from the nodes on its left and evaluates its function to get a new value. If the value of a situation node has changed, it will send the update to other situation nodes that depend on it. Each situation node also supports queries for its current value. In figure 4, situation B depends on sensor A, thus there is a directional link from A to B.

For a policy system to detect conflicts among dependent situations, the trigger part of a policy must specify all the sensors that are used to derive a situation. In the dependency graph, these sensors are the root nodes of the situation node. In the condition part of the policy, an environment variable with the name of the situation node is used. This environment variable is set up by the policy system to keep the most current value of the situation node. In figure 4 for example, if a policy requires situation F, then the trigger part of the policy is the list {A, C, E}. The parameter of the condition is F.
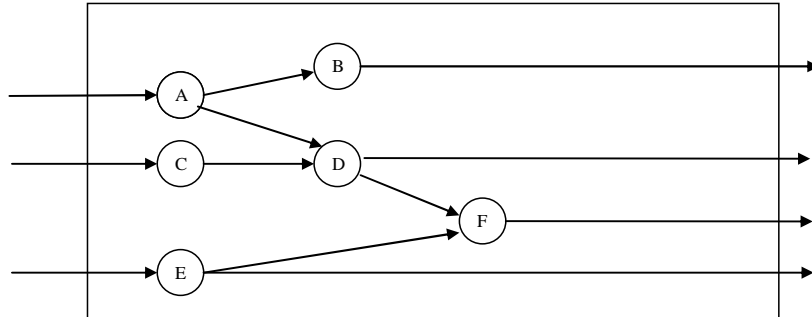
Figure 3.   Situation Dependency Graph

The dependency graph is designed and maintained by the service designer. If there are new sensors or new software module installed to detect new situations, the dependency graph is updated to reflect the changes. This may affect the triggers and conditions available to the policy definer when using the policy editing tool.

The example of 'door open' vs. 'door broken into' will show how this works. The 'door open' reminder policy has the following elements:

```
applies_to:[door1]
trigger: [door1:]
condition: door_open eq true
action:remind(reminder_bedroom, door left open)
preference: 3
```

The 'door broken into' policy has the following elements:

```
applies_to: [door1, lock2]
trigger: [door1:open, lock2:broken]
condition: broken_into eq true
action:remind(reminder_bedroom, door broken into)
preference: 5
```

In the above policies, the trigger of the first policy must originate from a specific door sensor, but does not require a particular type of trigger as this is implied by the condition. The trigger of the second policy originates from *door1* with type *open*, and from *lock2* with type *broken*. When the door is broken into, the policy server will receive triggers from the door sensor and door lock sensor at the same time. It will retrieve policies that apply to any of these sensors. Since the triggers and conditions of

both policies are satisfied, there will be two actions. These two actions compete for the same reminder service, so there are conflicts. In our system, these conditions are handled by part of a resolution policy. Other parts of the resolution policy decide which action to choose. For example, a policy preference may be used in the generic action *apply_stronger*. In this case, the action from the second policy would be executed.

5.2   Resolving Conflicts of Multiple Stakeholders

Detecting conflicts among multiple stakeholders is the same as for a single stakeholder except that the owners of the policies differ.   We therefore add a new generic resolution action: *apply_stakeholder*. This relies on the partial ordering among stakeholders to choose one action among the conflicting ones according to a predefined order among stakeholders. We believe that a total ordering of stakeholders in all situations is not sensible in home care. In a multiple organization setting, the ordering among stakeholders is not fixed and is valid only under certain conditions (e.g. when performing certain actions).

Specify the Order among Stakeholders. We make use of resolution policies to specify the order among stakeholders. The condition of the ordering rule can compare the attributes of the action and the policy. The new action *set_order* has been introduced to specify the ordering among stakeholders. This action has three parameters: the two different owners of the policies, and the relational operator between the owners (*gt*, *lt*, *eq*, unspecified). *gt* means the first owner ranks higher than the second, with the other operators having the obvious interpretation.

The example of Figure 4 shows that the warden has higher priority over the tenant for setting the TV volume at night time (from 23:00 to 7:00).

```
<conditions>
  <and>
  <condition>
    <parameter>time
    <operator>in
    <value>23:00..07:00
  <condition>
    <parameter>action
    <operator>eq
    <value>device_out(TV, setVolume)
```

```
<action arg1=":warden" arg2=":tenant"
  arg3="gt">set_order(arg1,arg2, arg3)
```

Figure 4.   Example of specifying Order among Stakeholders

Multiple orders among stakeholders can be specified under the same conditions. The relative orders among stakeholders are transitive. That is, if owner A is ranked higher than owner B and owner B is ranked higher than owner C, then owner A is ranked higher than owner C. This can help to simplify specification of the ordering.

The stakeholder parameters of the *set_order* action can also be roles. In the above example, a policy owner whose role is *warden* has higher priority than a policy owner whose role is *tenant*. Our policy system support roles through policy variables, which can contain a single value or a list of values.

Applying Order among Stakeholders. When conflicts are detected between policies of different owners, the resolution action *apply_stakeholder* can be used. Suppose there are conflicting policies that want to set the volume of the TV differently. The condition part of a resolution policy would check whether both their targets are the same TV; whether both actions are *set_volume,* and whether the volume levels are different. The policy preferences would indicate if both policies are positive obligation policies. The action part of the resolution would use *apply_stakeholder* to ensure that the warden's policy is respected.

5.3   Handling Interactions of Policy Actions over Time

To handle the policy interactions over time inside the policy system, we need to be able to detect the conflicts and then resolve the conflicts. According to Moffet's classification of policy conflicts, the interactions between a new action and the existing actions are goal conflicts (or resource conflicts) rather than modality conflicts. These are action conflicts, not policy conflicts. Detecting conflicts between new actions and existing running actions needs the policy system to keep a record of all running actions.

To find out whether a new action conflicts with ongoing actions, application-specific information is needed. This can be achieved by asking the user to specify the specific conditions to check. Similarly, resolving conflicts can be achieved by asking the user to resolve the choice of action manually. Even after an action is stopped due to conflict how to deal with it later also needs to be specified. In addition, only certain kinds of actions may suffer from this kind of conflict. For simplicity, we

therefore move the handling of action conflicts from the policy server to the actuators in our system. In the example given earlier, the alarm system would be an actuator and would use a priority-based approach to handle actions that conflict over time. A new alarm with a higher priority would stop an existing alarm with lower priority.

6   Related Work

Policy-based management has been applied in many areas, for example network and distributed systems management [2], telecommunications [14] [5], pervasive computing environments [10, 11, and 12], semantic web services [13], and large evolving enterprises [9].

The present paper has used the taxonomy of policy conflicts in [1] that describes how policy conflicts can be detected and resolved at definition time or at run time. [6] reviews policies in distributed systems and proposes using meta-policies to detect and resolve conflicts in one organization. However, this work does not tackle the problem of conflicting policies among multiple stakeholders. In addition, the solution in [6] is mostly for static detection and resolution of policy conflicts. Dunlop *et al*. have proposed a solution to detect conflicts dynamically using deontic logic, but this tackles only modality conflicts. [10, 11] propose a solution based on reasoning about the effects of actions to detect and resolve goal conflicts in pervasive computing environments. The aim is to guarantee the execution order of actions resulting from a single trigger. However, this work also does not deal with policy conflicts among multiple stakeholders.

7   Conclusion

The paper has focused on conflict issues when using policy-based management in home care systems. Three specialised kinds of conflict have been identified in home care: multiple stakeholder conflicts, policy conflicts due to dependency among situations, and conflicts among actions over time. Based on the analysis of these conflicts, we have proposed a solution to enhance our existing policy system to handle these conflicts. This has included extensions to the policy language and the policy system. The enhancements also have implications for other part of the home care system, in-

cluding sensors and actuators. We plan to evaluate the approach through field trials in actual homes.

References

[1] J. D. Moffett and M. Sloman. Policy Conflict Analysis in Distributed System Management. *Organizational Computing*, 4(1):1–22, 1994.

[2] N. Damianou, N. Dulay, E. Lupu and M. Sloman. Ponder: A Language specifying Security and Management Policies for Distributed Systems, Technical Report, Imperial College, London, UK, 2000.

[3] F. Wang, L. S. Docherty, K. J. Turner, M. Kolberg and E. H. Magill. Service and Policies for Care At Home, *Proc. Int. Conf. on Pervasive Computing Technologies for Healthcare*, pp. 7.1–7.10, Nov 2006.

[4] K. J. Turner, S. Reiff-Marganiec, L. Blair, J. Pang, T. Gray, P. Perry and J. Ireland. Policy Support for Call Control, *Computer Standards and Interfaces*, 28(6):635–649, Jun. 2006.

[5] K. J. Turner and L. Blair. Policies and Conflicts in Call Control, *Computer Networks*, 51(2):496-514, Feb. 2007

[6] E.C. Lupu and M. Sloman. Conflicts in Policy-Based Distributed Systems Management, *IEEE Trans. on Software Engineering*, 25(6), 1999.

[7] A. K. Dey, D. Salber and G. D. Abowd. A Context-based Infrastructure for Smart Environments. In *Proc. 1st Int. Workshop on Managing Interactions in Smart Environments*, pp. 114–128, Dublin, Dec. 1999.

[8] M. Perry, A. Dowdall, L. Lines and K. Hone. Multimodal and ubiquitous computing systems: supporting contextual interaction for older users in the home. *IEEE Trans. on IT in Biomedicine*, 8 (3):258–270, 2004.

[9] N. Dunlop *et. al*. Dynamic Conflict Detection in Policy-Based Management Systems, *Proc. EDOC '02*, 2002.

[10] C. Shankar, A. Ranganathan and R. Campbell. An ECA-P Policy-based Framework for Managing Ubiquitous Computing Environments, *Proc. 2nd Int. Conf. on Mobile and Ubiquitous Systems*, 2005.

[11] C. Shankar and R. Campbell. Ordering Management Actions in Pervasive Systems using Specification-enhanced Policies, *Proc. 4th Int. Conf. on Pervasive Computing and Communications*, Pisa, Mar. 2006.

[12] L. Kagal, T. Finin and A. Joshi, A Policy Language for Pervasive Systems, *Proc. 4th Int. Workshop on Policies for Distributed Systems and Networks*, Lake Como, Jun. 2003.

[13] A. Uszok, J. Bradshaw, R. Jeffers, M. Johnson, A.Tate, J. Dalton and S. Aitken. KAoS Policy Management for Semantic Web Services. *Intelligent Systems*, 19(4):32–41, 2004.

[14] Special Issue on Feature Interactions in Telecommunications Systems, *IEEE Communications Magazine*, 31(8), 1993.