Teleo-Reactive Policies for Managing Human-centric Pervasive Services

Srdjan Marinovic, Kevin Twidle, Naranker Dulay, and Morris Sloman
Department of Computing
Imperial College London, UK
{srdjan, k.twidle, n.dulay, m.sloman}@imperial.ac.uk

***Abstract*—Event-Condition-Action (ECA) policies are often used to manage various aspects of adaptation and execution of pervasive systems. Such policies are well suited for services where: 1) given actions are reliably executed when they are requested, 2) there is no priority ordering amongst multiple available actions, and 3) execution is instantaneous with respect to the validity of conditions under which they were initiated. However, for a pervasive service that integrates human agents and human activities, these assumptions do not generally hold. Humans may *misbehave* by postponing the execution of certain actions or ignoring them all together. Performing an action may take a long time so that the action is no longer needed or more important actions may need to be executed. Managing such behaviours through ECA policies is complex and difficult to implement. This paper introduces a new management policy type, called a Teleo-Reactive policy, whose semantics are based on continuous monitoring of the environment and prioritising available actions. The semantics result in more flexible and concise formulation of management policies for human-centric pervasive services. We demonstrate how these policies can be applied in a real-world use case scenario set in a nursing home and describe the underlying implementation based on the Android's Java platform.**

***Index Terms*—human-centric systems, pervasive services, management policies, Teleo-Reactive policies**

I. Introduction

Human-centric pervasive services place a much greater emphasis on modelling and integrating both human agents and pervasive devices within the application. Devices such as smartphones may prompt or remind humans to perform actions or warn about dangers, monitor and log activities, activate local devices to aid the user e.g. open doors or switch on lights. However, many application such as patient healthcare[1], [2], logistics [3], military applictions and personal human workflows [4], [5] require human agents to perform some aspects of the overall service. These applications depart from *classical* pervasive applications that model humans and their actions as entities who are part of the external environment rather than the application itself.

When modelling and integrating humans, it is tempting to view them as *managed* objects, who are issued commands to perform actions and expect them to reliably execute the actions, as with automated components. However, humans are fundamentally different from computers and the actions they are asked to perform are of a different nature. For example, humans may choose to ignore certain actions, or simply postpone them, often for entirely valid and unanticipated reasons. The performance of an action — the time taken, quality of the performance, resources used, can vary widely based on the human performing the action and the conditions under which they perform it. Human actions require more time than computational actions (minutes and hours instead of seconds) so conditions governing the start of the action may no longer be valid. Humans are not good at multi-tasking so services need to carefully schedule human actions to ensure that parallel actions are feasible.

Policy-based management has been successfully used in managing distributed systems and computer networks, and this success has been carried over to autonomous pervasive systems such as: body-sensor networks (BSNs) [6] and unmanned autonomous vehicles (UAVs) [7]. The management is usually based on Event-Condition-Action (ECA) policies that issue actions to managed objects. Hence, a possible approach to integrate the management of humans is to carry on using ECA policies and model humans as traditional managed agents.

In this paper we argue that ECA policies are too primitive to use when managed *agents* are humans and that we need new approaches that are better able to model and integrate the roles of humans in pervasive services. In particular, the main contribution of this paper is to propose and develop a new type of management policy, called a *Teleo-Reactive* (TR) policy whose semantics are based on continuous monitoring of conditions and continuous execution of actions. The paper's second contribution is to introduce a TR evaluation strategy based on observing changes rather than the polling (sampling) approach [8]. The paper extends our earlier investigation [9] that used TR programs to describe human-centric workflows. We apply our approach in a small scenario and evaluate our TR policy implementation for Android smartphones.

The rest of this paper is organised as follows. Section II discusses the related work regarding the human management in pervasive applications. Section III presents a use case scenario. Section IV introduces the Teleo-Reactive management policies; while Section V presents how the use case scenario can be modelled with these policies. Section VI discusses the differences between ECA and TR policies with respect to scenario's management requirements. Section VII examines the implementation of the TR policy framework and Section VIII looks at performance of the framework on the Android platform. Finally, Section IX concludes the paper with an outlook on the future work.

II. Related Work

Arguably the most elaborate integration of human actions in pervasive applications and services is found in human-oriented pervasive workflows such as [4], [3], [10]. These workflows specify a structured plan of activities and conditions under which these can be started. However, they are based on traditional workflow models and thus use a strict ordering of tasks where humans are expected to obediently follow given commands. Coping with *unpredictable* humans is attempted through adapting the workflow structure [3], or skipping over certain tasks [4]. But these are seen as exceptional situations and thus more elaborate integration with action prioritisation and long-running activities is not feasible within the confines of traditional workflow models. A more flexible workflow model, without a strict task ordering, was presented by Pesic et. al [11] and Aalst et al. [12]. However, this model does not express prioritisation between tasks nor does it cope with reorganising the execution of running tasks.

There has been considerable interest in the use of people-centric sensing which relies on people carrying smartphones as a mobile sensing platform to monitor the environment, traffic flows as well as determining the activity of the person carrying the phone [13]. This does not integrate any form of people management or control but the activity monitoring is similar to that required for our approach.

Policy management, based on the Event-Condition-Action (ECA) policy paradigm quickly emerged and established itself as a very successful management paradigm for distributed systems [14], [15], [16], [17]. However, humans and their activities have peculiar characteristics, namely: 1) human actions take minutes rather then seconds, 2) conditions during the executions change, and 3) humans prioritise sequential actions rather than multitask. ECA rules are not suited for specifying long running actions where conditions and priorities change during the action execution and agent behaviour can be unpredictable, as discussed further in Section VI.

Some of the issues with ECA policies were recognised by Chomicki et al. [18] which focused on addressing the conflict resolution between the executable actions. Conflicting actions must never be executed concurrently so priority can be associated with actions to determine which ones will be executed or ignored when conflicts occur. Conditions can also be used to specify whether an event triggering an action can be ignored. However, this work is still aimed at computer-based distributed systems and thus all actions are considered instantaneous. It is difficult to specify ECA rules for stopping and restarting actions in case higher-priority ones need to be run or to terminate long running actions if the initiation conditions no longer hold.

This paper builds upon our initial investigation [9] on using the Teleo-Reactive (TR) paradigm to specify adaptive human-oriented workflows. This gave us a flexible way of supporting long and centralised workflows that attempted to organise all of the people's activities within one workflow specification. This approach does not scale for complex environments involving many different people and their obligations. Thus in this paper we focus on smaller, *policy* TR structures that are distributed to people, and we find this *decentralised* approach to be a much more powerful way to integrate humans within pervasive system.

III. Case Study: Nursing Home Patient Care

People and their actions are often unpredictable and it is unrealistic to try to classify all the possible situations that might occur in practice, particularly in environments dominated by human-to-human interactions. More importantly, in most workplaces, people are given responsibilities and are expected to use their own judgement in carrying out their duties. But, people do make mistakes and do forget, particularly when working under pressure or heavy workloads. Building on these observations, unexpected activities should be considered the norm in pervasive applications. In most situations the people that cause deviations will be non-malicious known users rather than malicious adversaries.

To illustrate the need for policies and how they may be formulated, we shall focus on a scenario centred around the patient care in a nursing home in Mainkhofen, Germany. This case study is taken from the EU sponsored Allow project[1] [19].

The patients in the nursing home suffer from dementia and require constant care and assistance with almost all of their daily activities, such as: changing, bathing, eating and so forth. These activities are, by their nature, centered around physical interaction between care-givers and patients. The pervasive system is able to monitor which activities are taking place, who the individuals are and automatically log activities performed by nurses relieving them of the burden of manually writing up treatment logs. The monitoring of patients' vitals is also undertaken. All care-givers are equipped with a mobile device to monitor their activities, receive alarm messages and prompt them to perform obligations and tasks. The main goal of the system is to help and assist the care-givers with their daily chores as well as introduce automatic book-keeping of treatments that patients received.

The people (roles) modelled in the case study are: nurses, head-nurse, students and patients. The patients' care is formulated by the following statements:

1) A nurse that is assigned to a patient, executes the treatment tasks for that patient.
2) A student is allowed to interact with a patient when the assigned nurse is present in the same room.
3) When a patient's vital signs indicate an abnormality, any nurse or student is allowed to treat the patient.

As far as the nursing home is concerned, this formulation represents an ideal norm, and as is the case with many pervasive scenarios, it is subject to numerous deviations such as nurses or students misunderstanding which patient they should attend to. Sometimes a nurse will attend to a patient, for whom she is not directly responsible, as she determines

[1] http://www.allow-project.eu/

something may be wrong or that the patient needs something. Because the nursing home is not a high-level security building, it is quite possible that visitors forget to sign in and take their ID badges. In this case the system will treat them as unknown people and it is important that the system can quickly determine whether they are genuine visitors or intruders who should not be allowed into the nursing home.

It is clear that all of these cases represent deviations to normal behaviour/policies, and that there is a need for appropriate responses. For example, in this paper we model the following:

1) A student, who is assisting a patient without supervision, should be instructed to seek approval. At the same time the patient's nurse should be informed and allowed to approve the activity. If approved, the student should be informed and allowed to proceed. If the student continues without approval for 5 minutes the head nurse is informed of the activity of the student.
2) When a nurse is assisting a non-assigned patient, the head nurse should be notified and given the option to approve the activity. If the head nurse does not approve it, the nurse should be asked to provide a reason (through text or voice) for undertaking the activity. If one is not given the head nurse and the patient's nurse should be informed.
3) When an unknown person is detected in the ward, the video feed from that ward is immediately streamed to the ward nurse's mobile device (a ward has one assigned nurse to oversee it) and to the head-nurse's mobile device as well. A display in the ward shows a message asking the unknown person to authenticate themselves either by showing their badge or by going to the reception to get a badge.

IV. Teleo-Reactive Policies

As it was suggested in the previous sections, the main three requirements for managing human agents and their activities are: 1) prioritising between available activities, 2) monitoring conditions while the activities are being performed, 3) responding to condition changes while the activity is being performed. This paper argues that systems based on ECA policies are not well-suited to tackle these problems and put a high burden on the policy writer to understand and specify correct policy specification. Furthermore, this paper proposes the use of management policies which are based on, and inspired by, Teleo-reactive (TR) programs [8] to tackle the human actions' management challenges. These programs were introduced in behavioural robotics to govern a robot's continuous responses to various context changes in its environment, so a robot would reach its goal.

A. Syntax

A TR policy is written as an ordered list of condition-action rules; where every rule consists of a condition part and an action part (as depicted in the Figure 1). These parts are separated by the $\rightarrow$ symbol, and a rule is ended with a mandatory full-stop. $cond_i$ is a predicate that is evaluated over the policy's *percepts* (discussed in the next subsection) and $action_i$ is a function (method) that the policy can invoke. Conditions can form an expression with the logical *and* operator denoted with a comma , symbol. The $\neg$ symbol can be used as a shorthand for a condition returning *false*. Concurrent actions are indicated by the $||$ operator. Every policy specification has a name, and we employ a syntactic convention of indenting rules belonging to a policy. All policy rules are priority ordered (high to low) according to their syntactic position in the policy specification.

tr-policy $name(Par_1, .., Par_n)$
$cond_{1a}(Var_1), cond_{1b} \rightarrow \texttt{action}_1(Var_1)$.
$cond_2(Par_1) \rightarrow \texttt{action}_{2a} \;||\; \texttt{action}_{2b}$.
$cond_n \rightarrow \texttt{action}_n$.

Fig. 1: Teleo-Reactive (TR) Policy

All conditions, actions and policies start with a lower-case letter. All variables start with an upper-case letter and variables appearing in the action part of a rule must appear in the condition part of the same rule. A condition binds a value to a variable so that the condition with that value evaluates to *true*. Policies can be instantiated with parameters. The value of the parameters cannot be changed while the policy is active. A single-line comment starts with the % character.

B. Evaluation Semantics

Every TR policy is paired with the policy's *percepts* — a set of facts about the state of the system and the environment over which policy conditions are evaluated. The runtime system receives various events originating from the system or the environment which can change the values of contained facts. These changes can result in a condition becoming true and thus causing a new higher-priority rule's action to be executed or a condition relating to the currently executing rule may become false allowing a lower-prioirty rule with a true condition to start executing its action.

The policy life cycle consists of loading a policy specification, enabling the policy, disabling it and finally removing it. When the policy is enabled, it is said to be *active*. As soon as a policy becomes active, it is treated as an independent process, and is evaluated according to the five rules given in the Figure 2.

It is also possible for a policy to be nested as an action of another rule (please see Figure 6). In this case all conditions at all levels of the TR policy hierarchy are continuously evaluated. A called TR policy thus has no control over when it will be terminated; any parent (calling) TR policy would terminate it if higher-priority conditions become true. Such hierarchical nesting of TR policies and conditions leads to easier development of policies that are required to react robustly to an unpredictable changing environment since outer-level conditions take precedence over inner-level conditions.

The implicit prioritisation of TR policies frees the policy writer from writing additional prioritisation rules and it also

1) *All rule condition expressions are continuously evaluated.*
2) *There can only be one action expression (which may contain an action or parallel actions) running at any time, and it is must be the one belonging to the highest priority rule whose conditions are evaluated to true.*
3) *Actions are durative – they run forever.*
4) *As soon as the condition for the currently active rule becomes false, its action is terminated and the action expression for the next higher priority rule whose condition is true is started. There should always be a final rule with a true condition.*
5) *As soon as the condition for an higher-priority rule becomes true, its action expression is started and the currently running action expression is terminated.*

Fig. 2: Evaluation rules for a single TR policy

provides conflict-free action execution as only one policy is effectively executed at a time within a node. More elaborate priority schemes can be realised through parallel and/or nested TR policies. However, this usage can lead to conflicts where actions have conflicting semantics (e.g. switch-on and switch-off) or there may be conflicts between actions performed by multiple nodes executing policies. There is other on-going policy analysis (such as [20]) work within our group to determine conflicts and it is not covered in this paper. It may appear that executing only one action is too restrictive but humans usually need to be directed in a single step-by-step fashion rather than giving them a multi-tasking assignments.

A TR policy can be considered a priority-ordered, goal-oriented set of steps which are sequenced by reactions to changing conditions rather than through a fixed and rigid workflow-like structure.

V. Managing the Nursing Home with TR Policies

This section presents five TR policies that capture the deviation management for the nursing home case study. These TR policies rely on an activity recognition system that can identify and notify which activities a person is doing, developed within the Allow project [19]. In the case of a nursing home these activities come from a closed set of nursing tasks such as washing a patient, feeding a patient, taking blood pressure, cleaning a room etc.

The approach taken to managing deviations is as follows. Each active participant (such as a nurse, ward, etc.) is given a particular tailored policy. The rules of this policy are constructed in such way that conditions reflect a particular deviation pattern and the actions represent activities that either the human (manual actions) or the device (issuing notifications and alarms) ought to be doing. These actions can be considered as having a goal to falsify the deviation's conditions. The rules are ordered (prioritised) based on the seriousness of a particular deviation. Hence every TR policy, in this scenario, has a goal to correct deviations in the order of their seriousness and the goal can be considered reached when the policy is idle and not running any actions.

Following the TR semantics, all actions (both human and computer ones) are perceived as continuous. Hence both human and computer agents ought to be doing them as long as the deviation conditions hold. The humans are notified via alerts and screen messages which activities they are supposed to be doing. In theory this notification is continuous, in a sense that it is performed as long as the activity is needed, but in practice, notification implementation may do this by periodically reminding the user rather than generating a continuous audible warning which would be irritating. Some activities are impractical to stop instantaneously, for example washing a patient. TR policies give the policy writer the ability to specify additional rules to deal with such situations such as sending a warning message when a user does not finish an action on time.

To make a syntactic distinction between human actions and the device's ones, the human actions will be specified with a *dot* expression, eg. *Human.action(...)*. In this case *Human* is a variable that uniquely identifies a person and the *action* is the activity that the person ought to be doing.

```
tr-policy studentPolicy(Student) =
 %% monitor the student, after the approval,
 %% for a later review
 doing(Student, Activity, Patient, Ward),
 approved(Student, Activity, Patient, Ward)
   → monitor(Student, Activity, Patient, Ward).

 %% if the student is persistently deviating
 %% inform the head nurse
 ¬present(Patient.nurse, Ward),
 doing(Activity, mins(5))
   → Student.stop(Activity)
   || inform(headNurse, needsSupervision(Student, Activity
        , Patient,Ward)).

 %% a nurse needs to be present for student's actions
 doing(Student, Activity, Patient, Ward),
 ¬present(Patient.nurse, Ward)
   → Student.seekApproval()
   || inform(Patient.nurse, needsSupervision(Student,
       Activity, Patient, Ward)).
```

Fig. 3: *Student Policy: manages student's deviations and their progression.*

Figure 3 shows a student policy instantiated on a personal mobile device (see Figure 8a). The *doing(Student, Activity, Patient, Ward)* condition is *true*, while the student is performing an *Activity*. The *seekApproval* instructs the user to ask for an approval and it also tells him which is the closest nurse that can approve his activity. The student's device may need to perform the *inform* action which keeps notifying the patient's nurse about this deviation.

The *nurse* policy (Figure 4), specifies that if a nurse is doing an unapproved activity she needs to provide a reason for it, before dealing with a student's violation (if one exists). However, if a stranger is detected in the ward, the policy dictates that such a violation is the most urgent one and it instructs the device to show the streamed CCTV video while

```
tr-policy nursePolicy(Nurse) =
 %% display the live video on the device
 %% and have nurse go to the ward to identify
 %% the stranger
 strangerIn(Ward)
   → stream(Ward.videoCam)
   || Nurse.identifyStranger(Ward).

 %% monitor the approved activity for a later review
 doing(Nurse, Activity, Patient, Ward),
 approved(Nurse, Activity, Patient, Ward)
   → monitor(Nurse, Activity).

 %% prompt the nurse to give a reason for the deviation
 doing(Nurse, Activity, Patient, Ward),
 ¬approved(Nurse, Activity, Patient, Ward)
   → Nurse.seekReason(Activity).

 %% a nurse can assist only her patients otherwise
 %% it is a deviation
 doing(Nurse, Activity, Patient, Ward),
 ¬my(Nurse, Patient)
   → warn(Nurse, Activity)
   || inform(headNurse, needsApproval(Nurse, Activity,
        Patient, Ward)).

 %% student is deviating and needs approval
 needsApproval(Student, Activity, Patient, Ward)
   → Nurse.approve(Student, Activity, Patient, Ward).
```

Fig. 4: *Nurse Policy: manages nurse's deviations and approvals for students' deviations.*

instructing the nurse to attempt to identify the stranger. The Figure 8b shows how the *approve* action could be presented on a nurse's mobile device.

```
tr-policy headNursePolicy(HeadNurse) =
 strangerIn(Ward)
   → stream(Ward.videoCam)
    || HeadNurse.identifyStranger(Ward).

 %% in case a head nurse needs to deal with both a
 %% student's and a nurse's deviations, higher importance
 %% is given to the nurse's deviation
 needsApproval(Nurse, Activity, Patient, Ward)
   → HeadNurse.approve(Nurse, Activity, Patient, Ward).

 needsApproval(Student, Activity, Patient, Ward)
   → HeadNurse.approve(Student,Activity,Patient,Ward).
```

Fig. 5: *Head Nurse Policy: manages approvals for nurses' and students' deviations.*

The *headNurse* policy (Figure 5) follows the same logic as the *nurse* policy, except that a head nurse is permitted to interact with all patients and thus she never causes any deviations by herself.

A nurse can be assigned to a head nurse *role* based on a fixed or an ad-hoc schedule. This assignment is not done by the TR policy system, but by an independent role assignment system which deploys the policies relevant to a role. When a particular nurse assumes the head nurse role, the relevant (preloaded) policies are activated. To capture this behaviour, we use nested TR policies where, depending on a role a nurse is supposed to play her device enforces a particular policy. This is done via a special *tr* action which simply keeps a specified TR policy instantiated while the conditions are true.

In our implementation, a nurse's device contains both nurse and head nurse policies and which one is enforced is governed by the *nurseRole* policy (Figure 6).

```
tr-policy nurseRole(Nurse) =
 %% while activated executes headNurse TR policy
 assignedAsHeadNurse() → tr headNursePolicy(Nurse).

 %% while activated executes nurse TR policy
 onDuty() → tr nursePolicy(Nurse).
```

Fig. 6: *Nurse Role Policy: switches between head nurse's and nurse's policies based on the current nurse's assignment.*

VI. Discussion

In order to provide a clearer contrast between ECA and TR policies with respect to managing human agents and actions, this section encodes the *nurse* policy (Figure 4) as a set of ECA policies. The policies, shown in the Figure 7, are described with the syntax that follows the common pattern of *ON event IF conditions are true DO actions*. It is also assumed, in tradition with the common implementations, that the conditions are evaluated when the event is received and corresponding actions are executed once if the conditions are satisfied.

```
ON strangerIn(Ward)
  DO stream(Ward.videoCam).start()
    || identifyStranger(Nurse).start()

ON strangerLeft(Ward)
  DO stream(Ward.videoCam).stop()
    || identifyStranger(Nurse).stop()

ON approved(Activity,Patient,Ward,Period)
  IF Patient.nurse != Nurse
     & doing(Nurse, Activity, Patient, Ward)
     & !strangerIn(Ward)
  DO display("Approved", Activity, Period)

ON declined(Activity,Patient,Ward)
  IF Patient.nurse != Nurse
     & doing(Nurse, Activity, Patient, Ward)
     & !strangerIn(Ward)
  DO seekReason(Nurse, Activity)

ON started(Nurse, Activity, Patient, Ward)
  IF Patient.nurse != Nurse
    & !strangerIn(Ward)
  DO alert() || inform(HeadNurse)

ON needsSupervision(Student, Activity, Patient, Ward)
  IF !doing(Nurse, Activity, Patient, Ward)
     & !strangerInWard(Ward)
  DO approve(Student, Activity, Patient, Ward)
```

Fig. 7: *Nurse Policy represented as a set of ECA policies.*

The given policy set needs to explicitly prioritise ECA policies by augmenting the conditions with additional checks to make sure that no higher priority policies are applicable. In this example most policies are applicable only if the stranger is not present in the ward and that condition needs to be made explicit in most of the policies.

The second explicit consideration is needed to make sure that actions and their effects are properly terminated. In the

presented formulation, for the sake of brevity, we have only done this for the *stream* and *identifyStranger* actions. Even if one attempts to model these actions as threads that are simply initiated, they still need to be explicitly killed when new ones are started. The *Nurse* global variable is also included for this policy set to hold the currently assigned nurse. This works for one nurse, but a more complex mapping would be needed to cope with multiple nurses.

Finally, additional rules are needed: 1) to terminate running activities if conditions change, and 2) to automatically re-evaluate other policies to see if any other actions are executable. In the given example, once a stranger has left the ward, the nurse should automatically be given the option to approve any students' deviations, if they have been occurring.

Such concerns and the resulting complexities, in both the ECA specifications and implementations are the main reasons that prompted us to investigate other policy formulations and approaches. Our current experience, with human-centric pervasive services, indicates that TR policies address these problems in a more understandable and straightforward way, and are much closer to higher level goal based specifications.

VII. Implementation

To validate the proposed TR policy management approach we have developed a Java-based TR policy framework and used it to implement the presented scenario's policies. The framework targets the Standard Java 1.5+ distribution (Java SE) and Android's Java distribution, Dalvik 1.6+. The main reasons for supporting a mobile platform are: 1) to be able to cater for pervasive scenarios where a dedicated centralised infrastructure is not present, and 2) to be able to scale to systems where there are potentially hundreds of human agent by moving policy evaluation onto personal mobile devices. The policy framework can be downloaded from *http://www.ponder2.net/cgi-bin/moin.cgi/TrPonder*.

Our implementation of the nursing home scenario assumes that all nurses and students have an Android smartphone, which also contains an activity recognition module [19]. Therefore users are not required to manually input which activities they are performing. Every Android device runs the TR policy framework to evaluate the policies for the owner's role – nurse, head-nurse, student etc.. Figure 8a shows a student's device executing the policy's lowest rule. Similarly, Figure 8b shows a nurse's device executing also the nurse policy's lowest rule as well.

A. Evaluating TR Policies

In the current implementation each policy is treated as a Java thread. When started, a policy asks each rule, in turn, if it can be run. The top-most rule to respond positively is run in a separate thread (referred to as the *action* thread). The policy's evaluation thread then proceeds to re-evaluate the rules continually. Each time the rules are evaluated, the top-most rule is run. If the top-most rule is the currently running rule then that rule is simply left alone to continue. If, however, a higher priority rule is ready to run then the current action

(a) A student TR policy (b) A nurse TR policy

Fig. 8: The student policy is running its lowest rule since a student is doing an unapproved activity, and the nurse policy is prompting a nurse to approve a student's activity while the student is deviating.

thread is told to stop and when it stops, the newly selected rule's actions are run as the action thread.

The percepts are implemented as hash-based *key-value* stores and every time a value is changed, the policy attached to these percepts receives a notification.

In the described TR semantics, policies ought to continuously re-evaluate their rules as if they were implemented as an electronic circuit. Obviously, this *polling* approach, can be quite costly in terms of the CPU usage. Also, sometimes it is not desirable to immediately kill certain action threads as they may need additional time to finish using potentially critical resources.

For these highlighted reasons the evaluation can be configured to be either continuous or discrete, and the killing of actions can be delayed. These options are:

- *Continuous* – the policy repeatedly passes over its rules executing one as necessary, polling its percepts for every condition. The most straight-forward way to implement the needed semantics.
- *WaitForChange* – this is the option for discrete evaluation of the conditions, where the policy only re-checks the rules when its percept's values are changed. The change can be introduced by other components or even low-level ECA policies that have access to percepts. If an action thread completes and no change is perceived then the policy waits until one arrives.
- *WaitForRule* – this option tells the policy that it cannot terminate an action and that it has to wait for it to finish

even if a higher priority rule is ready to run. The only exception is if the policy, itself, has been told to stop then the currently running rule will be forced to stop.

The reason for having the *Continuous* mode of evaluation is to allow the policy's conditions to query other parts of the system directly if needed, rather than always having to go through percepts.

When a rule has two or more concurrent actions, a corresponding number of action threads are spawned. From the policy's point of view they are all treated as one action, and thus if the policy is waiting for that rule to finish, it is effectively waiting for all action threads to finish.

A rule's action can be another TR policy. In this case the action thread will instantiate this sub-policy and it will start evaluating its conditions and running its actions. However, when the original *parent* action thread needs to be terminated, it will send a terminate signal to the running sub-policy, which in turn will terminate its running action thread. Following this, the sub-policy is terminated.

B. Implementing TR Actions

Our current implementation views policy actions as Java methods. Once the action thread is started, it simply invokes the action method and then it waits to be terminated. At this point it is up to the actual method's code to implement its continuous running. For example the presented notification actions simply put the message on the screen and then periodically vibrate the phone to draw a user's attention to the screen. Hence, an action's concrete implementation is free to *appropriately* define it's own continuous behaviour. We experimented with running action methods in a loop inside the action thread, but abandoned this as every action can have a slightly different notion of what continuous means. Invoking a method continuously made the system keep state on how often it was called which made the actual action code quite hard to debug and understand.

Finally, activities that humans are supposed to carry out are most often implemented as methods that notify a user about the activity, or offer some input method for the user to signal that the activity is done. Based on the notification method's view on what continuous means, the user may be asked to do the action again repeatedly or a certain fixed number of times.

VIII. Performance Evaluation

The targeted pervasive environment, for TR policies, are mobile smartphone devices to which the policies can be deployed and then evaluated in a distributed manner. This implies that the TR policy environment should use minimal device CPU time so as not to impact other, more critical, applications and to prolong the devices battery life. Section VI showed that the ECA approach is inherently harder to use for managing human agents and human activities. Therefore this section will concentrate on assessing whether TR policies are a feasible tool for use on mobile devices rather than their direct performance comparison with ECA rules seeing the application domains for the two approaches are different.

TABLE I: *Running **1** and **20** HeadNurse TR policy (Figure 5)*

		1 TR policy	20 TR policies
	# of Changes	% of CPU Time	% of CPU Time
WaitForChange	4	1.39	2.20
	6	1.43	2.51
	12	1.51	3.62
	24	1.66	5.18
Continuous	4	21.13	81.41
	6	20.92	81.51
	12	20.52	81.26
	24	20.69	81.52

To analyse the TR policy framework's CPU usage on an Android smartphone device (with respect to *WaitForChange* and *Continuous* evaluation modes), we have taken the *HeadNurse* TR policy (Figure 5) and varied the number of changes in the percepts that the policy has to deal with. The changes are scripted in such a way that at the end of a test, the policy ended up doing the deviation handling (running an action thread) half of the time and half of the time it ended up in an idle mode (no action running). Note that the action thread is running a notification action which puts up an input screen and waits for the user feedback (as depicted in the Figure 8b). This is a very lightweight action and thus all the Figures can be used as a baseline for the policy frameworks usage rather than the action implementations' usage. These tests were run over a fixed time period of 5 minutes. They were all run on the Google Nexus One phone[2] with the Android 2.1 update.

The table I shows the percentage of the CPU usage by the policy framework when running 1 and 20 policies respectively, in both evaluation modes. It is immediately clear that the *WaitForChange* evaluation approach is a very economical one. Even with the most extreme case of 20 policies dealing with $20 * 24$ changes, the framework occupies only around 5% of the CPU time. On the other hand dealing with more than one policy in the *Continuous* mode will quickly drain the battery and slow other applications down. Thus this mode should only be used with one policy per device if there is an explicit need for it.

The previous tests highlighted the CPU usage when an action thread was running half of the total policy evaluation time. The table II shows the CPU usage when there is an action thread running continually during the policy evaluation time. We have kept the same minimal notification action so the test is not dependent on the complexity of the actual action. As the table shows additional cost is very small and can be considered negligible.

During all the executed tests the policy framework's memory usage varied between 4.5 MBs and 6MBs. The numbers varied within these bounds with no clearly observable correlation to the number of policies or the policy evaluation mode.

[2]http://www.google.com/phone/static/en_US-nexusone_tech_specs.html

TABLE II: *WaitForChange HeadNurse policy (Figure 5)*

	# of Policies	% of CPU Time
12 Changes	1	1.48
	10	2.72
	20	3.92
24 Changes	1	1.55
	10	3.88
	20	6.01

IX. Conclusion and Future Work

Traditionally, pervasive systems have managed computer-based agents (such as robots or sensors), and these frameworks have been largely based on Event-Condition-Action (ECA) policies. Current pervasive applications are attempting to integrate humans and their activities as an intrinsic part of a pervasive service rather than treating them as an external and independent environment. However, managing people provides new challenges, since people are often unpredictable and may decide to delay or postpone certain obligations. Also, they may act proactively and perform activities that the system does not expect them to do.

This paper has argued that traditional ECA policies are not suitable for managing human agents and their activities. The argument is based on the fact that capturing the management requirements for human services with ECA policies leads to a complicated and verbose policy specification that is tightly bound to the underlying system implementation. Furthermore, the policy writer has to *invent* his own mechanisms to deal with continuous actions and their prioritisation. To cope with these management requirements, this paper presents a new management policy type called Teleo-Reactive policies which view actions as continuous while the their activation conditions are continuously checked. The conditions in TR policies can be seen as goals which need to be achieved by actions so can be considered them as a higherl level abstraction than ECA rules.

Our current experiences with TR policies and comparisons with similar ECA formulations lead us to conclude that the TR approach offers a more concise and flexible formulation of the human agents' management requirements and its semantics lend themselves more naturally to these formulations.

We still need to evaluate the effectiveness of the above approach within the Allow project and the extent to which this is accepted by users in the nursing home scenario. For the future work, we plan to investigate additional scenarios to better understand management patterns that the TR policies are suitable for. We are also currently formulating a formal model of TR policies based on Event Calculus. The goal of this formal specification is to give a policy writer ability to verify that certain actions will take place after a particular narrative of changes. This formal model could be further used to check and validate the TR implementations.

Acknowledgment

This research was supported by EU FP7 research grant 213339 (ALLOW).

References

[1] J. E. Bardram and N. Norskov, "A context-aware patient safety system for the operating room," in *UbiComp '08: Proceedings of the 10th international conference on Ubiquitous computing*. New York, NY, USA: ACM, 2008, pp. 272–281.

[2] J. E. Bardram and H. B. Christensen, "Pervasive computing support for hospitals: An overview of the activity-based computing project," *IEEE Pervasive Computing*, vol. 6, no. 1, pp. 44–51, 2007.

[3] A. Marconi, M. Pistore, A. Sirbu, H. Eberle, F. Leymann, and T. Unger, "Enabling adaptation of pervasive flows: Built-in contextual adaptation," in *ICSOC/ServiceWave*, 2009, pp. 445–454.

[4] S. Urbanski, E. Huber, M. Wieland, F. Leymann, and D. Nicklas, "Perflows for the computers of the 21st century," in *PerCom Workshops*, 2009, pp. 1–6.

[5] Y. Li and J. A. Landay, "Activity-based prototyping of ubicomp applications for long-lived, everyday human activities," in *CHI '08: Proceeding of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*. New York, NY, USA: ACM, 2008, pp. 1303–1312.

[6] S. L. Keoh, N. Dulay, E. Lupu, K. Twidle, A. Schaeffer-Filho, M. Sloman, S. Heeps, S. Strowes, and J. Sventek, "Self-managed cell: A middleware for managing body-sensor networks," in *MobiQuitous 2007.*, Aug. 2007, pp. 1–5.

[7] E. Asmare, A. Gopalan, M. Sloman, N. Dulay, and E. Lupu, "A policy based management architecture for mobile collaborative teams," in *PerCom*, March 2009, pp. 169–174.

[8] N. J. Nilsson, "Teleo-reactive programs for agent control," *J. Artif. Intell. Res. (JAIR)*, vol. 1, pp. 139–158, 1994.

[9] S. Marinovic, K. Twidle, and N. Dulay, "Teleo-reactive workflows for pervasive healthcare," in *PerCom Workshops*, 2010.

[10] J. Han, Y. Cho, E. Kim, and J. Choi, "A ubiquitous workflow service framework," *Computational Science and Its Applications - ICCSA 2006*, pp. 30–39, 2006.

[11] M. Pesic, H. Schonenberg, and W. M. P. van der Aalst, "Declare: Full support for loosely-structured processes," in *EDOC*, 2007, pp. 287–300.

[12] W. M. P. van der Aalst, M. Pesic, and H. Schonenberg, "Declarative workflows: Balancing between flexibility and support," *Computer Science - R&D*, vol. 23, no. 2, pp. 99–113, 2009.

[13] A. T. Campbell, S. B. Eisenman, N. D. Lane, E. Miluzzo, R. A. Peterson, H. Lu, X. Zheng, M. Musolesi, K. Fodor, and G.-S. Ahn, "The rise of people-centric sensing," *IEEE Internet Computing*, vol. 12, no. 4, pp. 12–21, 2008.

[14] M. Z. Hasan, "An active temporal model for network management databases," in *Proceedings of the fourth international symposium on Integrated network management IV*, 1995, pp. 524–535.

[15] N. Damianou, N. Dulay, E. Lupu, and M. Sloman, "The ponder policy specification language," *Policies for Distributed Systems and Networks*, pp. 18–38, 2001.

[16] T. Koch, C. Krell, and B. Kraemer, "Policy definition language for automated management of distributed systems," in *SMW '96: Proceedings of the 2nd IEEE International Workshop on Systems Management (SMW'96)*. Washington, DC, USA: IEEE Computer Society, 1996, p. 55.

[17] J. Lobo, R. Bhatia, and S. Naqvi, "A policy description language," in *AAAI '99/IAAI '99*. Menlo Park, CA, USA: American Association for Artificial Intelligence, 1999, pp. 291–298.

[18] J. Chomicki, J. Lobo, and S. Naqvi, "Conflict resolution using logic programming," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 15, no. 1, pp. 244–249, Jan.-Feb. 2003.

[19] "First approach to integration of context," in *ALLOW Project Deliverable D2.2. http://www.allow-project.eu/*, Nov. 2009.

[20] R. Craven, J. Lobo, J. Ma, A. Russo, E. C. Lupu, and A. K. Bandara, "Expressive policy analysis with enhanced system dynamicity," in *ASIACCS*, 2009, pp. 239–250.