Policy Refinement: Decomposition and Operationalization for Dynamic Domains

Robert Craven,[1] Jorge Lobo,[2] Emil Lupu,[1] Alessandra Russo,[1] Morris Sloman[1]
[1]Department of Computing, Imperial College London, UK
[2]IBM Research, T.J. Watson, NY, USA
{robert.craven, e.c.lupu, a.russo, m.sloman}@imperial.ac.uk, j.lobo@us.ibm.com

***Abstract*—We describe a method for policy refinement. The refinement process involves stages of decomposition, operationalization, deployment and re-refinement, and operates on policies expressed in a logical language flexible enough to be translated into many different enforceable policy dialects. We illustrate with examples from a coalition scenario, and describe how the stages of decomposition and operationaliztion work internally, and fit together in an interleaved fashion. Domains are represented in a logical formalization of UML diagrams. Both authorization and obligation policies are supported.**

I. Introduction

The problem of policy refinement is that of automating the movement from a high-level, abstract characterization of a policy, to policies in the languages of the various enforcement points, which refer to the concrete objects and actions performed. The policies must be fine-tuned to the specifications of devices, and they should respond to changes in the system they regulate in ways that guarantee that the high-level policy passed as input is still satisfied. The problem is increasingly important, as reflected in the growing number of papers in recent years [1], [2], [3], [4], [5], [6]. However, in spite of this trend of increasing interest, there are still very few coherent approaches covering multiple aspects of the problem.

In recent work [7], [8] we presented ideas on policy decomposition. This is the process of using the relationship between actions at a higher level of abstraction and the sequences of actions at lower levels that can implement them, in the generation of refined policies. Policies that reference subjects and targets in more abstract terms are transformed into policies that apply to the objects that compose those subjects and targets. Our ideas for decomposition used a language based on constraint logic programs, with concepts from data integration used for the form of the decomposition rules.

In the current paper, we extend this line of inquiry further to focus on two things: (i) the nature of *operationalization* as the second, crucial feature of policy refinement; and (ii) the integration of decomposition and operationalization into a complete method for the refinement of authorization and obligation policies. Operationalization is the process of selecting the objects to which policies apply. We also consider, more briefly, the problem of re-refinement—when, after an initial refinement and enforcement of policies, the information in the domain changes to force alterations to the refinements.

Our policy language is a subset of that of our analysis framework [9]. We use a logical representation of UML for domains, and modify the syntax of our earlier presentation of decomposition rules, in very minor ways, to allow the integration of decomposition and operationalization. We emphasize that the formalism, which was presented at greater length in [10], is a logic for accomplishing various analysis and refinement tasks on policies, and is not intended, in its current form, as an enforceable policy specification language.

After a brief overview (II), we present our formalism (III), and an account of decomposition rules and their application (IV). We present the technical details of operationalization, with examples (V). Next, we consider the interleaving of decomposition and operationalization, and also the question of re-refinement (VI). After this is related work, and a conclusion.

II. Refinement Overview

There are three kinds of input to the refinement process: a UML representation of the domain; an authorization or obligation policy; and decomposition rules representing how actions and objects described at a higher level of abstraction relate to those described at a lower level. All are formalized in normal logic programs with constraints. The policy given as input is typically at a high level of abstraction, and the purpose is to refine it, as automatically as possible, into low-level policies that are expressed in whatever industrial policy languages the enforcement points support.

The formal representation of the UML has both a description of the class structure, the associations possible, and the operations one can perform on instances of the classes; and an *instance repository*, that records the specific objects existing in the domain and the relations between them. There is also an account of the object and operation terminology understood by the policy enforcement points; when policies have been refined so they are expressed exclusively in terms of such vocabulary, refinement has ended and the policies are translated into specific policy languages and sent for enforcement.

Our refinement process interleaves two different phases, using the different categories of input in turn. First, a high-level policy is *operationalized* so that all the objects it applies to are found. This means comparing the conditions in the policy with the instance repository to determine the possible, specific sets of objects satisfying those conditions. For each such combination of objects, a phase of *decomposition* follows. Decomposition rules that match the operationalized policies are applied, so the policies are expressed in terms

of more concrete classes of objects—closer to policies that can be sent for enforcement. The way decomposition works depends on whether the policy being refined is an authorization or obligation policy. Decomposition rules provide information about how an action is to be implemented, and we take the view that a positive authorization policy allows the subject to perform the action in *all* ways, a negative authorization policy prohibits *all* ways of carrying out an action, and an obligation policy requires a subject to carry out an action in *one* way.

After a decomposition phase, reference to new classes at a more concrete level are introduced; the new policies need to be operationalized, so objects of those classes can be found from the instance repository. Thus, after an initial operationalization, refinement is iterated stages of decomposition and operationalization. After operationalization, the policies are tested to see whether they are expressed in terms an enforcement point understands. If so, they are translated into the specific policy languages (e.g. Ponder [11]) running on the enforcement points and sent for implementation; different translation schemes $trans_1, \ldots, trans_n$ are used for this process. Our method is illustrated in Figure 1.

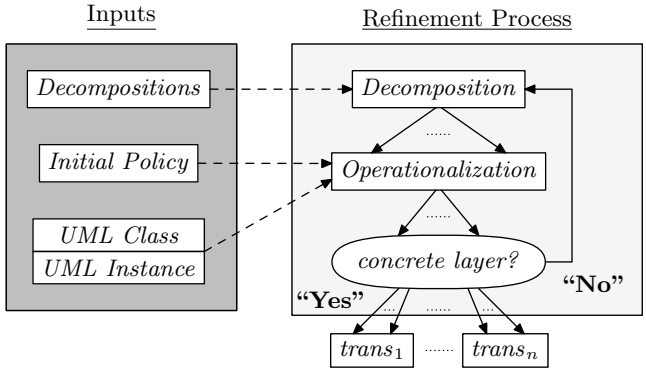

Fig. 1. Overview of refinement

If, after refinement, objects in the domain are created, destroyed, or their properties change, then information held in the instance repository, used to refine the existing policies, may no longer be valid, and the policies themselves depending on this should be replaced with others consistent with the changes. Knowledge of classes and their operations may also change: half-way through the lifetime of a system, for instance, a new type of server might be introduced. As decomposition rules relate actions performable on classes to their implementation as actions performed on those classes' components, the introduction of new classes typically means the introduction of new decomposition rules. These possibilities mean that re-refinement must be studied, as a way of automating the alterations necesarry to the existing refinements.

III. Basic Formalism

The representation of domains is a simple logical formalization of UML diagrams, with several additional features used for reasoning over the domain. Decomposition rules' structure must reflect the hierarchies and relationships implicit in the UML representation. Policies themselves are expressed in a logical language expressive enough that many widely-used policy languages can be translated into it; the language is that of normal logic programs. In the current section, we discuss the representation of domains and the policy language.

The representation of domains is itself divided into two parts. The more 'static' bit describes the classes of object present, the *isa* relationships between them, and the types of possible association and aggregation. The 'dynamic' part is liable to change more often: e.g. if a specific object $vidCam_{23}$ exists at time 4 and is a device, this can be represented as $holdsAt(obj(vidCam_{23}, device), 4)$.

Definition 1 A *class definition* $\mathcal{C}$ is a set of ground facts of *class*, *isa*, *assType*, *aggregType*, *compType*, *classOp* such that: for all $isa(c, c')$, $assType(c, a, c')$, $aggregType(c, a, c')$ or $compType(c, a, c') \in \mathcal{C}$, then $class(c)$, $class(c') \in \mathcal{C}$; if $classOp(c, o, n, l) \in \mathcal{C}$ then $class(c) \in \mathcal{C}$ and for all $c' \in l$, $class(c') \in \mathcal{C}$ and l has length n. We also include:

$$isaTrns(C, C') \leftarrow isa(C, C'). \quad (1)$$
$$isaTrns(C, C') \leftarrow isa(C, C''), isaTrns(C'', C'). \quad \lrcorner$$

$assType(c, a, c')$ is used to represent that an association a can hold between objects of classes c and c'; similarly for *aggregType* and aggregations, and *compType* and compositions. $classOp(c, o, n, l)$ is used to represent that an operation o can be performed on objects of class c; the operation has n parameters whose classes are in l. For instance, the fact $classOp(nikon_{23}, takePhoto, 2, [nikonRes, posInteger])$ may represent that Nikon cameras of model 23 support a *takePhoto* operation that takes two arguments: the photo resolution, and the number of photographs to take. (1) is used for reasoning over the transitive closure of the *isa* relation.

Definition 2 An *instance definition at* t, where t is a non-negative rational number, is a set of ground instances of *holdsAt* all of whose second arguments are t, and whose first arguments—the *fluents*—are instances of *obj*, *ass*, *aggreg* or *comp*; with the following definition:

$$holdsAt(objTrns(O, C), T) \leftarrow holdsAt(obj(O, C), T). \quad (2)$$
$$holdsAt(objTrns(O, C), T) \leftarrow holdsAt(O, C'), T), isaTrns(C', C). \lrcorner$$

Our policy language includes positive and negative authorization rules, and obligation rules. The authorization rules have the form $permitted(Sub, Act, Tar, T) \leftarrow B$ or $denied(Sub, Act, Tar, T) \leftarrow B$, and the obligation policy rules have the form $obligation(Sub, Act, Tar, T_s, T_e, T) \leftarrow B$. The language has sorts for subjects, actions and targets, and the relevant arguments in the head of the policy rules should be terms of those sorts. The bodies B are conjunctions of atoms *holdsAt* (with fluents *obj*, *ass*, *aggreg* and *comp*), *happens*, *do* and *fulfilled*, with constraints over the temporal variables. The *holdsAt* refer to properties of the domain, and the temporal constraints are used to say when all these conditions should be true: the conditions can be historical. (In previous versions [8] of our language we allowed a more expressive syntax for

policies; this is simplified here to focus on the fundamental logic of policy refinement.) Constraints c are of the forms $s_1 = s_2$, $s_1 < s_2$, $s_1 \leqslant s_2$, where the s_i have the form n, v, $s_1 + s_2$ or $s_1 - s_2$—for $n \in \mathbb{R}^+ \cup \{0\}$ and variables v.

As an example, consider the obligation policy that *[All communications officers from every platoon must file daily summary reports to their divisions within one hour of a reminder]*. This can be formalized in our policy language as:

$$\begin{aligned} &\mathit{obligation}(\mathit{Sub}, \mathit{fileRep}(R), \mathit{Tar}, T_1, T_2, T) \leftarrow \\ &\quad \mathit{holdsAt}(\mathit{obj}(\mathit{Sub}, \mathit{commsOfficer}), T), \\ &\quad \mathit{holdsAt}(\mathit{ass}(\mathit{Sub}, \mathit{belongs}, P), T), \mathit{holdsAt}(\mathit{obj}(P, \mathit{platoon}), T), \\ &\quad \mathit{holdsAt}(\mathit{obj}(R, \mathit{report}), T), \mathit{holdsAt}(\mathit{obj}(\mathit{Tar}, \mathit{division}), T), \\ &\quad \mathit{holdsAt}(\mathit{ass}(P, \mathit{part\_of}, \mathit{Tar}), T), \\ &\quad \mathit{happens}(\mathit{reminder}(\mathit{fileRep}, R, \mathit{dailySummary}), T_1), \\ &\quad T_2 = T_1 + 3600,\ T_1 \leqslant T \leqslant T_2. \end{aligned} \tag{3}$$

The first six conditions in the body define the subject, action and target the policy applies to; the next, *happens* predicate requires that the obligation depends on the occurrence of a *reminder* event; and the constraints require that the subject has sixty minutes to fulfil the obligation, that is imposed immediately the event on which it depends occurs.

Our policy language has many other features which we will not treat in detail here. However, important in what follows will be the predicate *fulfilled*, which is defined independently of the policies as follows:

$$\begin{aligned} &\mathit{fulfilled}(\mathit{Sub}, \mathit{Tar}, \mathit{Act}, T_s, T_e, T_f, T) \leftarrow \\ &\quad \mathit{obligation}(\mathit{Sub}, \mathit{Tar}, \mathit{Act}, T_s, T_e, T_f),\ \mathit{do}(\mathit{Sub}, \mathit{Tar}, \mathit{Act}, T_f), \\ &\quad \textbf{not}\ \mathit{cease\_obl}(\mathit{Sub}, \mathit{Tar}, \mathit{Act}, T_{\mathit{init}}, T_s, T_e, T'), \\ &\quad T_{\mathit{init}} \leqslant T_s \leqslant T' < T_e,\ T' < T_f, T_f < T. \\ &\mathit{cease\_obl}(\mathit{Sub}, \mathit{Tar}, \mathit{Act}, T_s, T_e, T) \leftarrow \\ &\quad \mathit{do}(\mathit{Sub}, \mathit{Tar}, \mathit{Act}, T'), T_s \leqslant T' < T \leqslant T_e. \end{aligned}$$

An obligation is fulfilled if it is currently active, and the action required by the obligation is performed at the right time. Fulfilment conditions will be important when we refine to sequences of actions, later, as a way of ensuring the actions are performed in the right order.

IV. Decomposition

We achieve decomposition by the application of *decomposition rules*, relating domain elements represented more abstractly to the components and implementations of those elements at a more concrete level. We first describe the application of a single decomposition rule.

Decomposition rules are presumed to be part of the input to the refinement process, and are properly considered to be part of the domain knowledge. The basic component of a decomposition rule is a *conditioned action*.

Definition 3 A *conditioned action* is an expression of the form $(Q_s\mathit{Sub}, \mathit{Act}, Q_t\mathit{Tar}) : C_1, \ldots, C_n$; the conditions C_i have the forms $\mathit{obj}(O, C)$, $\mathit{ass}(O, A, O')$, $\mathit{aggreg}(O, A, O')$ or $\mathit{comp}(O, A, O')$. C is a class name from the domain, and A is an association, aggregation, or composition type. Any variable occurring in *Sub*, *Act* or *Tar* also occurs in the C_i. The expressions Q_s and Q_t are either empty, or quantifiers of the form $\forall$ or $\exists!n$, where n is a positive integer; if both of Q_s and Q_t are non-empty, then one is preceded by $^+$. ⌟

These expressions define a set of performances of actions by subjects on targets, subject to the constraints $C_1, \ldots, C_n$ and relative to a particular state of the instance repository that represents the domain. Their full meaning becomes clearer when it is shown how they function in decomposition rules.

The quantifiers will be used as a cue for the operationalization process to determine how many subjects or targets to find satisfying the conditions in the policies derived using the decomposition rule. Where *Sub* is the subject of a conditioned action (whose type will be constrained by the conditions $C_1, \ldots, C_n$), a quantifier $\forall\mathit{Sub}$ means that the action should be performed by all *Sub* of the constrained type; a quantifier $\exists!n\mathit{Sub}$ would mean that precisely n subjects of the relevant type should perform the action. If both subject and target have a quantifier, the $^+$ indicates ordering, and hence scope. If the head of a conditioned action is $(^+\forall\mathit{Sub}, \mathit{Act}, \exists!3\mathit{Tar})$, then every subject of the type defined must perform *Act* on precisely 3 targets—not necessarily the *same* targets. Alternatively, if the head of the conditioned action is $(\forall\mathit{Sub}, \mathit{Act}, {}^+\exists!3\mathit{Tar})$, this means that precisely three targets should have *Act* performed on them by all subjects—in this case, the *same* three targets.

Definition 4 A *decomposition rule* is an expression $\mathsf{C} \Rightarrow [\mathsf{C}_1\ \textbf{then}\ \cdots\ \textbf{then}\ \mathsf{C}_n]$, where each C_i is a conditioned action. C is the *top*, and the C_i are the *bottoms*. There must be some $\mathit{obj}(\mathit{Sub}, C)$ and $\mathit{obj}(\mathit{Tar}, C')$ in the conditions of C, such that for some n and l, $\mathcal{C} \models \mathit{class}(C) \wedge \mathit{class}(C') \wedge \mathit{classOp}(C'', \mathit{Act}, n, l) \wedge \mathit{isaTrns}(C, C'')$. If the form is

$$\begin{aligned} &(\mathit{Sub}, \mathit{Act}, \mathit{Tar}) : \mathit{Conds} \Rightarrow \\ &\quad (Q_s^1\mathit{Sub}_1, \mathit{Act}_1, Q_t^1\mathit{Tar}_1) : \mathit{Conds}_1 \\ &\quad\ \textbf{then}\ \cdots\ \textbf{then}\ (Q_s^n\mathit{Sub}_n, \mathit{Act}_n, Q_t^n\mathit{Tar}_n) : \mathit{Conds}_n \end{aligned} \tag{4}$$

then for all Q_x^i, Q_x^i is non-empty iff the variable quantified by Q_x^i does not appear as the subject or target of either the top of the rule, or a previous conditioned action (i.e. some Sub_j or Tar_j for $j < i$). Additional constraints on the relationships between the classes for the subjects and targets in the bottoms, and those for the subjects and targets of the top, are inherited from Definition 5 of [8]. These ensure the hierarchical relationships in the UML are observed. ⌟

The meaning of such decomposition rules is that when an action $(\mathit{Sub}, \mathit{Act}, \mathit{Tar})$ is performed (as identified by the conditions *Conds*), then this can be implemented by the sequence of actions in the bottom of the rule.

An application of a rule (4) matches its top against the policy being refined, and then decomposes the policy by producing one copy of it for each of the rule's 'bottoms', but with the subject, action and target replaced by those of the conditioned action in the bottom, with their quantifiers. The process is quite straightforward, and as we have described and illustrated it at length elsewhere [8] we do not repeat the bulk

of that here. The only difference with our previous presentation is the existence of the quantifiers. These do not affect the results of the decomposition process itself, but feed into the subsequent operationalization phase, as we later describe.

The output from an application of a decomposition rule to a policy has the form $((P_1, S_1), \ldots, (P_n, S_n))$ where each P_i is a policy with quantifiers, and each S_i is the sequence information, consisting of *do* literals and constraints in the case of authorization policies, and *fulfilled* literals and constraints in the case of obligation policies.

When the action permitted (denied) by an authorization policy decomposes to a sequence of actions, the S_i ensure that the previous actions in the sequence have been executed. Consider a sample policy that *[Communications officers are permitted to file reports to their platoons within 5 mins after a report notification event]*, and suppose the following rule:

$$
\begin{array}{l}
(\mathit{Sub}, \mathit{fileRep}(R), \mathit{Tar}) : \ \mathit{obj}(\mathit{Sub}, \mathit{soldier}), \ \mathit{obj}(\mathit{Tar}, \mathit{unit}) \\
\qquad \Downarrow \\
(\mathit{Sub}, \mathit{send}(R), \exists!2\mathit{Tar}_1) : \\
\quad \mathit{obj}(\mathit{Tar}_1, \mathit{reportRep}), \ \mathit{aggreg}(\mathit{Tar}_1, \mathit{belongs}, \mathit{Tar}) \\
\textbf{then} \ \ (\mathit{Sub}, \mathit{backup}(R), \exists!1\mathit{Tar}_2) : \\
\quad \mathit{obj}(\mathit{Tar}_2, \mathit{backupServer}), \ \mathit{ass}(\mathit{Tar}_2, \mathit{belongs}, \mathit{Tar})
\end{array} \tag{5}
$$

(The *Sub* variable in the bottoms of the decomposition rule needs no associated quantifiers because it appears in the top of the rule.) The meaning of this rule is that an action of a soldier filing a report to a unit can be implemented by that soldier sending the report to two of the unit's report repositories, and then backing up the report to one of the unit's backup servers. If the policy is decomposed using the rule, the result will be:

$$
\begin{array}{l}
\mathit{permitted}(\mathit{Sub}, \mathit{send}(R), \exists!2\mathit{Tar}_1, T) \leftarrow \\
\quad \mathit{holdsAt}(\mathit{obj}(\mathit{Sub}, \mathit{commsOfficer}), T), \\
\quad \mathit{holdsAt}(\mathit{aggreg}(\mathit{Sub}, \mathit{belongs}, P), T), \\
\quad \mathit{holdsAt}(\mathit{obj}(P, \mathit{platoon}), T), \mathit{holdsAt}(\mathit{obj}(\mathit{Tar}_1, \mathit{reportRep}), T), \\
\quad \mathit{holdsAt}(\mathit{aggreg}(\mathit{Tar}_1, \mathit{belongs}, P), T), \\
\quad \mathit{happens}(\mathit{reminder}(\mathit{fileRep}, R), T'), T' \leqslant T \leqslant T' + 300. \\
\textbf{Sequence:} \ [\,]
\end{array} \tag{6}
$$

$$
\begin{array}{l}
\mathit{permitted}(\mathit{Sub}, \mathit{backup}(R), \exists!1\mathit{Tar}_2, T) \leftarrow \\
\quad \mathit{holdsAt}(\mathit{obj}(\mathit{Sub}, \mathit{commsOfficer}), T), \\
\quad \mathit{holdsAt}(\mathit{aggreg}(\mathit{Sub}, \mathit{belongs}, P), T), \\
\quad \mathit{holdsAt}(\mathit{obj}(P, \mathit{platoon}), T), \\
\quad \mathit{holdsAt}(\mathit{obj}(\mathit{Tar}_2, \mathit{backupServer}), T), \\
\quad \mathit{holdsAt}(\mathit{aggreg}(\mathit{Tar}_2, \mathit{belongs}, P), T), \\
\quad \mathit{happens}(\mathit{reminder}(\mathit{fileRep}, R), T'), T' \leqslant T \leqslant T' + 300. \\
\textbf{Sequence:} \ [\mathit{do}(\mathit{Sub}, \mathit{send}(R), \mathit{Tar}_1, T''), \ T' \leqslant T'' < T]
\end{array} \tag{7}
$$

An authorization for $\mathit{backup}(R)$ to be performed must be given only when both *send* actions (to the two repositories) have been performed. Let us suppose that the subsequent operationalization phase chooses the two targets dump_1 and dump_2 for the first policy (6)—and that the subject throughout is *alice*. Then the second policy must only give authorization after $\mathit{do}(\mathit{alice}, \mathit{send}(R), \mathit{dump}_1, T_1)$ and $\mathit{do}(\mathit{alice}, \mathit{send}(R), \mathit{dump}_1, T_2)$ are both true. This means that the operationalizations of the first policy also apply to the *do* conditions, the **Sequence** information, which is related to the second policy. These *do* conditions can therefore only be added after the operationalization has taken place.

Similar remarks apply to *fulfilled* and obligation policies; for obligation policies, *fulfilled* plays a role analogous to that of *do* for authorizations. Notice that the addition of quantifiers before variables in our policies takes us strictly outside the language of normal constraint logic programs. However, we see a phase of decomposition followed by operationalization as unified steps in a process of refinement; after each phase of operationalization, the policies are again within the syntax of constraint logic programs.

V. Operationalization

Operationalization is the selection of specific resources policies apply to, and as this depends on which resources exist, the inputs to operationalization will be a sequence of semi-refined policies (from a decomposition) with associated sequence information, and the instance repository of current facts about the domain. The algorithms for selecting resources depend on the kind of input policy, and the nature of the system used to store information about the domain. Our interest here is twofold. We want to provide one implementation of our refinement procedure, using a logic-programming system with decomposition rules and standard logic-programming queries. We also want to model the overall refinement process, to show how the stages of decomposition and operationalization fit together, even if—in practice—the algorithms used to pick resources might differ. If the methods used to operationalize policies were based on constraint satisfaction algorithms, or linear programming, or other optimizatation algorithms, these could be 'plugged in' to our overall refinement design.

Policies have a general form $H(\mathit{Sub}, \mathit{Act}, \mathit{Tar}, \ldots, T) \leftarrow \mathit{Holds}, \mathit{Happens}, \mathit{Cons}, \mathit{Seq}$. The H is *permitted*, *denied* or *obligation*. *Happens* conditions are events the policy depends on. *Seq* conditions ensure sequences of actions are performed in the right order. The *Holds* conditions will, for the most part, constrain the subjects, actions and targets of the policy. The *Holds* conditions might also contain literals that define the parameters of the various *Happens* events. For example, in policy (3), the condition $\mathit{holdsAt}(\mathit{obj}(R, \mathit{report}), T)$ refers to the report that the reminder event in that policy's *happens* condition mentions. In this policy, it would not be possible to operationalize the reports at refinement-time—the report may not appear in the instance repository, and may only be created later. The grounding of the variable R is determined by the parameters of the *reminder* event, and the operationalization procedure should ignore it. Accordingly, we allow policy authors to mark variables in policies, to indicate that the corresponding resource should not be operationalized at refinement-time. Policy (3) can be altered to reflect this, by replacing all occurrences of R with R^-. Any variable marked with a $^-$ will not be operationalized.

As stated above, operationalization takes as input:

1) A sequence of quantified policies and sequencing conditions $((P_1, S_1), \ldots, (P_n, S_n))$ from a decomposition.
2) UML formalization of class information and instance repository, $\mathcal{CI}$.

Pseudocode for the top-level of the process is in Algorithm 1. (For the notation here: sequences and lists are represented as

Algorithm 1 *operationalize*$(((P_1, S_1), \ldots, (P_n, S_n)), \mathcal{CI})$

1: $L \leftarrow (((P_1, S_1), \ldots, (P_n, S_n)))$; $O \leftarrow ()$
2: **while** L is non-empty **do**
3: $\mathsf{P} \leftarrow$ *head*(L); $(P, S) \leftarrow$ *head*(P)
4: **if** *isGround*(*subject*(P)) $\wedge$ *isGround*(*target*(P)) **then**
5: **if** *tail*$(\mathsf{P}) = ()$ **then** $L \leftarrow$ *tail*(L)
6: **else** $L \leftarrow$ *tail*(L) **end if**
7: $L \leftarrow$ (*tail*$(\mathsf{P})) \cdot$ *tail*(L)
8: $P' \leftarrow$ *addSeq*(P, S); $O \leftarrow O \cdot$ (*prune*(P'))
9: **else**
10: **if** *hasPlus*(*subject*(P)) $\vee$ *isGround*(*target*(P)) **then**
11: $L' \leftarrow$ *pick*(*subject*(P)*target*$(P), P, S,$ *tail*$(P), \mathcal{CI})$
12: **else**
13: $L' \leftarrow$ *pick*(*target*(P)*subject*$(P), P, S,$ *tail*$(P), \mathcal{CI})$
14: **end if**
15: $L \leftarrow L' \cdot L$
16: **end if**
17: **end while**
18: **return** O

$(e_1, \ldots, e_n)$. *subject* and *target* pick out those terms in the heads of a policy. Substitutions θ with unique members X/Y are sometimes represented by them; the application to a term P is $P\theta$ or $P[X/Y]$. List concatenation is $\cdot$.)

The key sub-procedures here are *addSeq* and *pick*. *addSeq* takes a fully-operationalized policy (with ground subject and target) and adds the sequence information that comes from the operationalization of previous policies to its conditions. As explained at the end of Section IV, this is a matter of adding all the ground *do* or *fulfilled* literals and constraints to the policy's body. We do not describe this process further here, but illustrate it later in the section. The procedure *hasPlus* checks whether a quantifier for a variable is preceded by $^+$ (indicating the order in which it should be operationalized). *prune*(P) removes ground literals from the body of the policy.

The main work is done by the *pick* sub-procedure. This takes as input quantifiers over subjects and targets, constraining how many resources the policy applies to, and using the conditions policy body to discover the possible subjects and targets, finds the right number of them according to whichever plugged-in form of optimization algorithm is being used. The sequenceing conditions (in the *tail*(P) also passed as input) are simultaneously ground. This process is shown as Algorithm 2. Here, the procedure $l(\cdot)$ makes a list from the members of a set (in any order). In line 16, the expressions S_i^F are transformations of the S_i in accordance with the groundings in F: where $F = \{\theta_1, \ldots, \theta_n\}$, then the sequence condition *do*(Sub, Act, Tar, T), for instance, is transformed to:

Algorithm 2 *pick*$(Q_1Q_2, P, S, ((P_1, S_1), \ldots, (P_n, S_n)), \mathcal{CI})$

1: **if** *isGround*(Q_1) **then**
2: **if** $Q_2 = \forall X$ **then** $F \leftarrow$ *get*$(X,$ *all*$, P, \mathcal{CI})$
3: **else if** $Q_2 = \exists!mX$ $F \leftarrow$ *get*$(X, m, P, \mathcal{C})$ **end if**
4: **else**
5: **if** $Q_1 = \forall X$ **then** $G \leftarrow$ *get*$(X,$ *all*$, P, \mathcal{CI})$
6: **else if** $Q_1 = \exists!mX$ **then** $G \leftarrow$ *get*$(X, m, P, \mathcal{C})$ **end if**
7: **if** *isGround*(Q_2) **then**
8: $F \leftarrow G$
9: **else**
10: **if** $Q_2 = \forall Y$ **then** $F \leftarrow G$
11: **else if** $Q_2 = \exists!lY$
12: $F \leftarrow \{$*get*$(Y, l, P[X/x], \mathcal{CI}) \mid \exists\theta \in G, X/x \in \theta\}$
13: **end if**
14: **end if**
15: **end if**
16: **return** $l(\{(P\theta, S\theta) \mid \theta \in F\}) \cdot ((P_1, S_1^F), \ldots, (P_n, S_n^F))$

do$(Sub, Act, Tar, T_1)\theta_1, \ldots,$ *do*$(Sub, Act, Tar, T_n)\theta_n,$
$T = \max\{T_1, \ldots, T_n\}$

And *fulfilled*$(Sub, Act, Tar, T_s, T_e, T_f, T)$ is transformed to:

fulfilled$(Sub, Act, Tar, T_s, T_e, T_1, T)\theta_1, \ldots,$
fulfilled$(Sub, Act, Tar, T_s, T_e, T_n, T)\theta_n, T_f = \max\{T_1, \ldots, T_n\}$

The main subprocedure used here is *get*$(X, N, P, \mathcal{CI})$, which finds N resources as bindings for the variable X, in accordance with the constraints imposed by conditions in the policy P, with respect to the class definition and instance information $\mathcal{CI}$. Specific optimization and selection algorithms

Algorithm 3 *get*$(X, N, P, \mathcal{CI})$

1: $V \leftarrow \{\theta \mid \mathcal{CI} \models q(P)\theta \wedge$ *isGround*$(q(P)\theta)\}$
2: **if** $N =$ *all* **then return** V
3: **else**
4: $V_X \leftarrow \{x \mid \exists\theta \in V,\ X/x \in \theta\}$
5: $R \leftarrow \mathsf{PLUGIN}(N, V_X, P)$
6: **return** $\{\theta \in V \mid \exists x \in R,\ X/x \in \theta\}$
7: **end if**

can be plugged in to *get* at the point PLUGIN—this procedure chooses N objects from V_X in an optimal way, subject to constraints derived from the nature of the policy P for which the objects are being chosen. The policy's action may determine the type of heuristic used to select objects; or the policy itself may contain additional information, such as timing constraints, that affects the suitability of choices for N.

The procedure $q(P)$ used in Algorithm 3 takes a policy and transforms its conditions to be used as a query to the instance repository; the query finds those values of variables in the body that represent objects in the domain. To find $q(P)$ for a policy P, the *holdsAt* conditions in P's body are used. First, conditions containing variables

marked by $^-$ are omitted. Temporal variables are ground to the current time. Literals $\mathit{holdsAt}(\mathit{obj}(O,C),T)$ are replaced by $\mathit{holdsAt}(\mathit{objTrns}(O,C),T)$. For example, a subject may be constrained to be a sensor—$\mathit{holdsAt}(\mathit{obj}(\mathit{Sub},\mathit{sensor}),T)$ occurs as a condition in the policy. Yet the instance repository may only record which specific type of sensor an object is. If $\mathit{holdsAt}(\mathit{obj}(\mathit{sensor}_1,\mathit{vidCam}),3)$ is in the instance repository, and it is known that all video cameras (class *vidCam*) are sensors, then the operationalization should be able to take account of that. (2) achieves this.

For example, let us look at the operationalization of P_1, *[All communications officers in platoons must file a report to their division within an hour of being prompted to do so]*:

$$\begin{aligned}&\mathit{obligation}(^+\forall \mathit{Sub},\mathit{fileRep}(R^-),\exists!1\mathit{Tar},T_1,T_2,T)\leftarrow\\ &\quad \mathit{holdsAt}(\mathit{obj}(\mathit{Sub},\mathit{commsOfficer}),T),\\ &\quad \mathit{holdsAt}(\mathit{ass}(\mathit{Sub},\mathit{belongs},P),T),\mathit{holdsAt}(\mathit{obj}(P,\mathit{platoon}),T),\\ &\quad \mathit{holdsAt}(\mathit{obj}(R^-,\mathit{report}),T),\mathit{holdsAt}(\mathit{obj}(\mathit{Tar},\mathit{division}),T),\\ &\quad \mathit{holdsAt}(\mathit{ass}(P,\mathit{part\_of},\mathit{Tar}),T),\\ &\quad \mathit{happens}(\mathit{reminder}(\mathit{fileRep},R^-,\mathit{dailySummary}),T_1),\\ &\quad T_2=T_1+3600,\ T_1\leqslant T\leqslant T_2.\end{aligned}\tag{8}$$

Refinement is at time 200. (8) is passed as input to Algorithm 1 as $((P_1,()))$. Its subject and target are unground, and the subject is marked by a $^+$, so line 11 in the algorithm would be reached, calling $\mathit{pick}(\forall \mathit{Sub}\exists!1\mathit{Tar},P_1,(),(),\mathcal{CI})$, where $\mathcal{CI}$ is the class information and instance repository. Passing to Algorithm 2, the test at line 1 is failed, and control moves to line 5, calling $\mathit{get}(\mathit{Sub},\mathit{all},P_1,\mathcal{CI})$. $q(\cdot)$ generates a query:

$$\begin{aligned}&\mathit{holdsAt}(\mathit{objTrns}(\mathit{Sub},\mathit{commsOfficer}),200),\\ &\mathit{holdsAt}(\mathit{ass}(\mathit{Sub},\mathit{belongs},P),200),\\ &\mathit{holdsAt}(\mathit{objTrns}(P,\mathit{platoon}),200),\\ &\mathit{holdsAt}(\mathit{objTrns}(\mathit{Tar},\mathit{division}),200),\\ &\mathit{holdsAt}(\mathit{ass}(P,\mathit{part\_of},\mathit{Tar}),200)\end{aligned}\tag{9}$$

Suppose that at time 200, *alice* and *bob* are two communications officers (*commsOfficer*) belonging to plat_1, which is a *platoon*. plat_1 is *part_of* div_1, a *division*. Algorithm 3 outputs:

$$\{\{\mathit{Sub}/\mathit{alice},\mathit{Tar}/\mathit{div}_1\},\{\mathit{Sub}/\mathit{bob},\mathit{Tar}/\mathit{div}_1\}\}\tag{10}$$

Control passes back to Algorithm 2 and to line 12. This calls $\mathit{get}(\mathit{Tar},1,P',\mathcal{CI})$ twice: first with a policy where the *Sub* is *alice*, and then where it is *bob*. In the first case, in Algorithm 3, the query (9) gives the first member of (10) as the only substitution; in the second, the second member is returned. By line 13 of Algorithm 2, F contains both of (10). At line 16, $((P_2,()),(P_3,()))$ is returned, where P_2 is

$$\begin{aligned}&\mathit{obligation}(\mathit{alice},\mathit{fileRep}(R^-),\mathit{div}_1,T_1,T_2,T)\leftarrow\\ &\quad \mathit{holdsAt}(\mathit{obj}(\mathit{alice},\mathit{commsOfficer}),T),\\ &\quad \mathit{holdsAt}(\mathit{ass}(\mathit{alice},\mathit{belongs},\mathit{plat}_1),T),\\ &\quad \mathit{holdsAt}(\mathit{obj}(\mathit{plat}_1,\mathit{platoon}),T),\mathit{holdsAt}(\mathit{obj}(R^-,\mathit{report}),T),\\ &\quad \mathit{holdsAt}(\mathit{obj}(\mathit{div}_1,\mathit{division}),T),\\ &\quad \mathit{holdsAt}(\mathit{ass}(\mathit{plat}_1,\mathit{part\_of},\mathit{div}_1),T),\\ &\quad \mathit{happens}(\mathit{reminder}(\mathit{fileRep},R^-,\mathit{dailySummary}),T_1),\\ &\quad T_2=T_1+3600,\ T_1\leqslant T\leqslant T_2.\end{aligned}\tag{11}$$

and P_3 is P_2 with *alice* replaced by *bob*. Algorithm 1 then prunes these two policies of ground conditions in ther bodies and adds them to the output.

$$\begin{aligned}&\mathit{obligation}(\mathit{alice},\mathit{fileRep}(R^-),\mathit{div}_1,T_1,T_2,T)\leftarrow\\ &\quad \mathit{holdsAt}(\mathit{obj}(R^-,\mathit{report}),T),\\ &\quad \mathit{happens}(\mathit{reminder}(\mathit{fileRep},R^-,\mathit{dailySummary}),T_1),\\ &\quad T_2=T_1+3600,\ T_1\leqslant T\leqslant T_2.\end{aligned}\tag{12}$$

is left, with the corresponding version of policy P_3—again, the same as 12 with *bob* instead of *alice*. A phase of operationalization is then over, and the policies are tested for being enforceable. If this test is not passed, there is more decomposition and operationalization.

VI. Interleaving, Distribution, Re-refinement

Decomposition breaks down the policy, replacing it by a sequence of policies that apply to objects and actions specified at a lower level; the operationalization phase then finds which of the possible objects should be selected for the policy to apply to. After operationalization, there is a test to see whether the policies are now sufficiently low-level to be sent for enforcement. If passed, then translation schemes are used to transform the policies into languages the policy enforcement points understand. Which translation schemes to use will depend on the objects the policy applies to, and the kinds of enforcement points being used to enforce policies on those objects. For example, in the case of authorization policies in our example domain, it may be that filing a report ultimately means using SCP to send it to a directory on the division's report repository, followed by backing up the report by uploading it to an SVN server using `svn+ssh`. The enforcement point associated with SCP access to the report repository may be using Ponder2 [11]; the point controlling access to the SVN server may be using XACML [12]. Translation schemes would take policies expressed in our language-independent logic, and convert them into the necessary formats.

If the test to see whether the policies are ready to be deployed is not passed, another phase of decomposition and operationalization is entered. We show how this works for (12). We first apply the decomposition rule (5) (with the R variable marked to indicate it ought not to be operationalized):

$$\begin{aligned}&\mathit{obligation}(\mathit{alice},\mathit{send}(R^-),\exists!2\mathit{Tar}_1,T_s,T_e,T)\leftarrow\\ &\quad \mathit{holdsAt}(\mathit{obj}(\mathit{Tar}_1,\mathit{reportRep}),T),\\ &\quad \mathit{holdsAt}(\mathit{aggreg}(\mathit{Tar}_1,\mathit{belongs},\mathit{div}_1),T),T_s\leqslant T\leqslant T_e,\\ &\quad \mathit{happens}(\mathit{reminder}(\mathit{fileRep},R^-),T'),T_e=T_s+3600.\end{aligned}\tag{13}$$

$$\begin{aligned}&\mathit{obligation}(\mathit{alice},\mathit{backup}(R^-),\exists!1\mathit{Tar}_2,T_d,T_e,T)\leftarrow\\ &\quad \mathit{holdsAt}(\mathit{obj}(\mathit{Tar}_2,\mathit{backupServer}),T),\\ &\quad \mathit{holdsAt}(\mathit{aggreg}(\mathit{Tar}_2,\mathit{belongs},\mathit{div}_1),T),\\ &\quad \mathit{happens}(\mathit{reminder}(\mathit{fileRep},R^-),T'),\ T_d\leqslant T\leqslant T_e.\end{aligned}\tag{14}$$

Sequence: $[\mathit{fulfilled}(\mathit{alice},\mathit{send}(R^-),\mathit{Tar}_1,T_s,T_e,T_d,T),\ T_e=T_s+3600,\ T_s\leqslant T_d\leqslant T<T_e]$.

These are now sent to the operationalization algorithm; if we suppose that the instance repository contains the information

that rep_1, rep_2 and rep_3 are report repositories (*reportRep*), that all belong to div_1, and that *bS* is a backup server (*backupServer*) belonging to the same division, then the output from the operationalization phase will give:

$$\begin{aligned} &\mathit{obligation}(\mathit{alice}, \mathit{send}(R^-), \mathit{rep}_1, T_s, T_e, T) \leftarrow \quad (15)\\ &\quad \mathit{happens}(\mathit{reminder}(\mathit{fileRep}, R), T_s),\\ &\quad T_e = T_s + 3600,\ T_s \leqslant T \leqslant T_e.\\ &\text{(same as (15) with } \mathit{rep}_2 \text{ for } \mathit{rep}_1) \quad (16)\\ &\mathit{obligation}(\mathit{alice}, \mathit{backup}(R^-), \mathit{bS}, T_d, T_e, T) \leftarrow \quad (17)\\ &\quad \mathit{happens}(\mathit{reminder}(\mathit{fileRep}, R), T_s),\ T_e = T_s + 3600,\\ &\quad \mathit{fulfilled}(\mathit{alice}, \mathit{send}(R^-), \mathit{rep}_1, T_s, T_e, T_d^1, T),\\ &\quad \mathit{fulfilled}(\mathit{alice}, \mathit{send}(R^-), \mathit{rep}_2, T_s, T_e, T_d^2, T),\\ &\quad T_d = \max\{T_d^1, T_d^2\},\ T_d \leqslant T. \end{aligned}$$

The process then repeats itself, with the three policies above being sent to be tested to see whether they are ready to be enforced, or to further decomposition and operationalization.

It is important to note that the *fulfilled* conditions in (17), which refer to actions that might not be at the most concrete level for this domain, will not themselves be further decomposed in later refinements. The decomposition of $(\mathit{Sub}, \mathit{Act}, \mathit{Tar})$ triples only applies to the heads of policies, and not to *do* or *fulfilled* literals in policies' bodies. If a condition appears in the body of a policy that refers to the completion of an action specified at a high level of abstraction, event correlation engines can be used to check the condition's holding. They will relate the performance of the various low-level actions to the condition. The information in decomposition rules, with timing constraints on how close together the actions should occur, will be inputs to these correlation checks.

At this point we can state the details of the most general algorithm for refinement, involving interleaved stages of decomposition and operationalization. The algorithm is in two parts. The first, Algorithm 4, operates on a list of policies, and defines the overall control of how the authorization and obligation policies are differently treated, and what happens when a policy P has been fully refined (i.e. *isConcrete*(P) is true). This and Algorithm 5 call each other as co-routines. Algorithm 5 must call 4 because, in general, decomposition and operationalization of a single policies produces multiple lower-level policies.

The inputs to the refinement process are therefore a policy P, marked with quantifiers and possibly with $^-$ signs next to some of the variables to prevent their having resources assigned to them at refinement-time; a set of decomposition rules R; and class and instance information $\mathcal{CI}$. The policy is operationalized in a preliminary stage, and the results are then sent to *refineList*.

The algorithms presented above, and earlier in Section V, apply when all information is stored together, and the refinement process is conducted by a unique 'engine'. However, there are clearly many sorts of domain where this assumption is inappropriate. If distinct organizations have come together, and refine shared policies that apply in different ways to the various devices belonging to them individually, then the policy refinement process might begin in a shared fashion, applying to cross-organization units, but then be shipped to a refinement module for each organization. This is clearly realistic if the organizations are unwilling to pool sensitive information about their resources. It is straightforward to adapt Algorithm 4 to take account of this. Line 5 is replaced by a test examining whether the policy is ready to be shipped to a different refinement point, and if so, then Line 7 sends P to it. Similar alterations are made to the rest of the algorithm.

It is of course also necessary to consider what happens when the instance repository changes, and refinements must be reconsidered. Changes can be of four kinds: (i) a new object is introduced (with new associations) of an existing class; (ii) an existing object is removed, or some of its associations cease; (iii) a new class of objects is added, with associated decomposition rules for their actions, together with some instances; (iv) a class of objects is removed, with

Algorithm 4 *refineList*$(Ps, \mathcal{CI}, R)$

```
1: if empty(Ps) then
2:    return ()
3: else
4:    P ← head(Ps)
5:    if isConcrete(P) then
6:       if empty(tail(Ps)) then
7:          return (P)
8:       else
9:          L ← refineList(tail(Ps), CI, R)
10:         if ¬empty(L) then return (P) · L
11:         else return () end if
12:      end if
13:   else if P is an authorization policy then
14:      for all r ∈ R do
15:         M ← refineSingle(P, CI, {r});  L ← L · M
16:      end for
17:   else
18:      L ← refineSingle(P, CI, R)
19:   end if
20:   if empty(L) then return () end if
21:   L′ ← refineList(tail(Ps), CI, R, O)
22:   if empty(L′) then return () end if
23:   return L · L′
24: end if
```

Algorithm 5 *refineSingle*$(P, \mathcal{CI}, R)$

```
1: for all r ∈ R do
2:    L ← decompose(P, r)
3:    L ← operationalize(L, CI)
4:    if ¬empty(L) then
5:       L ← refineList(L, CI, R)
6:       if ¬empty(L) then return L end if
7:    end if
8: end for
9: return ()
```

the decomposition rules relating to them. For example, (i) corresponds to a new server of a known type being added to the network; and (iii) corresponds to a new type of server being added, which supports different low-level operations, together with the rules that relate those low-level operations to those at a higher level of abstraction.

We re-refine policies in response to domain changes as follows. When refining a policy initially, we record information about intermediate stages of refinement in a *refinement tree*. The nodes are semi-refined policies, and the edges are steps of decomposition or operationalization. To each edge is linked the information used in the refinement process to make the corresponding refinement step. For decomposition steps, this is the decomposition rule used; for operationalizations, it is the various facts from the instance repository used. This enables us to associate objects, and classes of objects in the domain, with the semi-refined policies which have depended on them for their refinement. When new instances are added to the domain of classes that already exist (our case (i), above), it is possible to check whether the new object's class (whether explicitly named, or inherited transitively using the *isa* relationship) is one that was used in the refinement of existing policies. If so, we can take those semi-refined policies' nodes in the refinement tree, and refine from that point again, noticing whether the new instances added to the domain produce new policies. In the case of authorization policies, these can be enforced; in the case of obligation policies, they may be alternatives to existing refinements, and an engineer can then, supported by policy analysis techniques, decide whether to adopt the new refinement or not.

In the case of the three other possible types of change, different points in the refinement tree are affected. The most complicated case is that of (iii), where a new class and related decomposition rules are added. The most straightforward way to treat this is to re-refine from the highest level of abstraction—to perform a completely new cycle of refinement. We are currently investigating ways to improve this, and to optimize other aspects of the re-refinement process.

We have a prototype implementation, written in Prolog. It takes policies to refine, a class definition, a history of the instance repository, and a specification of when policies are enfoceable. Policies are decomposed and operationalized automatically, and Prolog's built-in backtracking enables the selection of alternative decomposition rules and operationalizations. We have designed the tool to be compatible with our policy analysis implementation, based on the abduction in constraint logic programs [9], [10]. The emphasis in the current paper is on the formal account of the algorithms and structure of the refinement process, however; more details on the implementation will be given in future work.

VII. Related Work

There has been increasing interest in the automation of policy refinement in recent years, although with few distinctive and thorough approaches—partly due to the inherent difficulty of formalizing the problem. [13] uses decomposition patterns formalized in the Event Calculus, with an abductive reasoning method for deriving refinements from goals. The work was based on the KAOS [14] approach to requirements engineering, and used planning to go from goals to implementable actions. Our approach is basaed on data integration methods for refinement and uses a UML specification of the domain model. It also caters for multiple levels of refinement. [1] compares different paradigms for a policy transformation module, and presents experimental results for one particular implementation. [5] applies concepts from planning to generate implementable tasks from requests, and though this work is relevant to policy refinement, it is not applied to it by the author. [4], [15], [2], [3] treat other aspects of the problem.

In our own work [7] we have used ideas from data integration in the formulation and application of decomposition rules. In [8] we explored the decomposition of policies further. The current paper modifies the rules (while still basing them on the same form), extends the approach to include a full treatment of operationalization, and shows how the two are linked in a total refinement procedure. [6] takes a similar approach, confined to authorization policies. The authors also represent domains in a logic-programming formalization of UML, with transformation rules to map policies from a high to lower level, and present examples drawn from distributed firewalls in MANETS.

VIII. Conclusion

We have presented a refinement process for authorization and obligation policies that has iterated phases of decomposition and operationalization as its core. The method has as input a formalization of UML information on the objects to be regulated by policies, a high-level policy, and decomposition rules that relate actions and objects at higher levels of abstraction to those at lower levels. We described in detail how these inputs are used by the various phases of refinement to produce increasingly concrete policies.

There are many avenues for future work. We want to explore optimizations in our approach to re-refinement. As mentioned elsewhere in the paper, we are merging our refinement procedures with the policy analysis framework in [9]. This will let the possible refinements of a new policy be compared with existing policies, to see whether there are conflicts or redundancies; analysis can in this way be used to guide the refinement process. We also want to relax some of the syntactical restrictions we imposed in the current paper, so that the refinement process can treat a more expressive class of authorizations and obligations. Two obvious ways to this are to allow negative literals in the bodies of policies, and recursive dependencies amongst policies.

Acknowledgment Research was sponsored by the U.S. Army Research Laboratory and the U.K. Ministry of Defence under Agreement W911NF-06-3-0001. The views and conclusions in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Army Research Laboratory, the U.S. Government, the U.K. Ministry of Defence or the U.K. Government.

REFERENCES

[1] M. Beigi, S. B. Calo, and D. C. Verma, "Policy transformation techniques in policy-based systems management," in *POLICY*. IEEE Computer Society, 2004, pp. 13–22.

[2] J. Rubio-Loyola, J. Serrat, M. Charalambides, P. Flegkas, and G. Pavlou, "A functional solution for goal-ooriented policy refinement," in *POLICY*. IEEE Computer Society, 2006, pp. 133–144.

[3] S. Davy, B. Jennings, and J. Strassner, "Conflict prevention via model-driven policy refinement," in *DSOM*, ser. Lecture Notes in Computer Science, R. State, S. van der Meer, D. O'Sullivan, and T. Pfeifer, Eds., vol. 4269. Springer, 2006, pp. 209–220.

[4] G. Campbell and K. Turner, "Goals and policies for sensor network management," *International Conference on Sensor Technologies and Applications*, pp. 354–359, 2008.

[5] D. Trastour, R. Fink, and F. Liu, "ChangeRefinery: Assisted Refinement of High-Level IT Change Requests," *IEEE International Symposium on Policies for Distributed Systems and Networks*, pp. 68–75, 2009.

[6] H. Zhao, J. Lobo, A. Roy, and S. Bellovin, "Policy refinement of network services for manets," in *IM2011*, 2011 (to appear).

[7] R. Craven, J. Lobo, E. Lupu, A. Russo, and M. Sloman, "Security policy refinement using data integration: a position paper," in *SafeConfig '09: Proceedings of the 2nd ACM workshop on Assurable and usable security configuration*. New York, NY, USA: ACM, 2009, pp. 25–28.

[8] ——, "Decomposition techniques for policy refinement," in *CNSM*. IEEE, 2010, pp. 72–79.

[9] R. Craven, J. Lobo, J. Ma, A. Russo, E. Lupu, A. Bandara, S. Calo, and M. Sloman, "Expressive policy analysis with enhanced system dynamicity," in *ASIACCS*. ACM, 2009, pp. 239–250.

[10] R. Craven, E. Lupu, J. Lobo, A. Bandara, S. Calo, J. Ma, A. Russo, and M. Sloman, "An expressive policy analysis framework with enhanced system dynamicity," Technical Report, Department of Computing, Imperial College London, 2008.

[11] N. Damianou, N. Dulay, E. Lupu, and M. Sloman, "The ponder policy specification language," in *POLICY*, ser. LNCS, M. Sloman, J. Lobo, and E. Lupu, Eds., vol. 1995. Springer, 2001, pp. 18–38.

[12] OASIS XACML TC, "extensible access control markup language (XACML) v2.0," 2005. [Online]. Available: http://xacml-2.notlong.com

[13] A. K. Bandara, E. Lupu, J. D. Moffett, and A. Russo, "A goal-based approach to policy refinement," in *POLICY*. IEEE Computer Society, 2004, pp. 229–239.

[14] R. Darimont and A. van Lamsweerde, "Formal refinement patterns for goal-driven requirements elaboration," *SIGSOFT Softw. Eng. Notes*, vol. 21, pp. 179–190, October 1996. [Online]. Available: http://doi.acm.org/10.1145/250707.239131

[15] M. Abadi and L. Lamport, "The existence of refinement mappings," *Theor. Comput. Sci.*, vol. 82, no. 2, pp. 253–284, 1991.