Decomposition Techniques for Policy Refinement

Robert Craven
Dept. of Computing
Imperial College London
SW7 2AZ

Jorge Lobo
IBM T.J. Watson
New York

Emil Lupu
Dept. of Computing
Imperial College London
SW7 2AZ

Alessandra Russo
Dept. of Computing
Imperial College London
SW7 2AZ

Morris Sloman
Dept. of Computing
Imperial College London
SW7 2AZ

***Abstract*—The automation of policy refinement, whilst promising great benefits for policy-based management, has hitherto received relatively little treatment in the literature, with few concrete approaches emerging. In this paper we present initial steps towards a framework for automated distributed policy refinement for both obligation and authorization policies. We present examples drawn from military scenarios, describe details of our formalism and methods for action decomposition, and discuss directions for future research.**

I. Introduction

The automated refinement of high level goals and policies into implementable management and security policies would help non-programmers specifying policies in situations such as service management, pervasive applications and for healthcare services. Refinement involves generating enforceable and implementable policies guaranteed to achieve and preserve the high-level security and system management goals from which the policies are derived. Some progress has been made, but as yet there is no comprehensive, accepted solution. This paper presents our ideas on the refinement of authorization and obligation policies, with a preliminary view of a formal framework, algorithms and representations.

We view the process of policy refinement as comprising three aspects: decomposition, operationalization and distribution. In *policy decomposition*—the main focus of this paper—policies expressed at higher levels of abstraction are mapped into lower-level policies; successive mappings move closer to concepts that are directly implementable. The mapping is achieved using policy-independent *refinement rules*, defined within the scope of an application-specific system model. The paper describes the syntax of the refinement rules, how they relate to and are constrained by the system model, and the way they are applied to authorization and obligation policies.

Operationalization associates abstract policy classes with specific subjects responsible for initiating actions and specific targets on which the actions are performed, obtained from the system model. Required action parameters may also depend on specific targets. The refinement process may itself need to be *distributed*, e.g. to organizations part of a collaboration: parts of the system model may be distributed, or there may be concerns about the confidentiality of policies within organizations. In a rich, distributed policy scenario, we foresee there being several phases of policy decomposition, followed by operationalization, followed by distribution.

There is a need to interleave policy refinement with analysis to ascertain that the refined specification achieves the requirements and is consistent with system properties and limitations, as well as with existing policies. Parallel, distributed refinement could lead to policies which conflict with each other as the refinement context will not be the same across all entities. Consequently, in the current paper we build on our previous work on policy analysis [1], [2], using the same representation language, with a view to the future integration of refinement and analysis in a single framework. The nature of the interleaving itself is left for future work.

II. System Models

A system model defines the scope for specification of policies: the agents (human or automated) that can be subjects and targets, services and devices on which actions can be performed, detailed specification of the action parameters (c.f. interface specification) as well as the organizational relationships of all these entities. We use UML class diagrams to define a system model as they are widely-used, well-supported and perspicuous. Subjects and targets of policies are objects of classes specified in the class diagram, with associations being used to constrain the objects to satisfy specified properties. The main relationships that refinement exploits are generalization, aggregation, association and composition, so we currently restrict the UML to these relationships between classes.

The examples used in this paper are taken from a more detailed military policy scenario [3], involving platoons, divisions, sensor networks, and the submission of daily activity reports across the organizational structure. Figure 1 shows a fragment of the UML system model for that scenario which we will use in this paper. This example contains instances of class specialization and generalization, named associations, and aggregation (there is no composition).

Our representation language is a subset of first-order logic. We use the Event Calculus [4] (EC) to describe the state of the system, and to express the conditions under which a policy applies; we therefore show how the UML used to represent domains can be translated into the EC. The variant of the EC we use is presented in full in [2]. $\mathit{holdsAt}(\cdot,\cdot)$ is used to represent the changing properties of a system's state, with the first argument taking a *fluent* (a property whose value changes over time) and the second a time at which the system has the property expressed by the fluent. Thus, $\mathit{holdsAt}(\mathit{obj}(s, \mathit{backupServer}), 4)$ could represent that the object s exists as a backup server, at time 4.

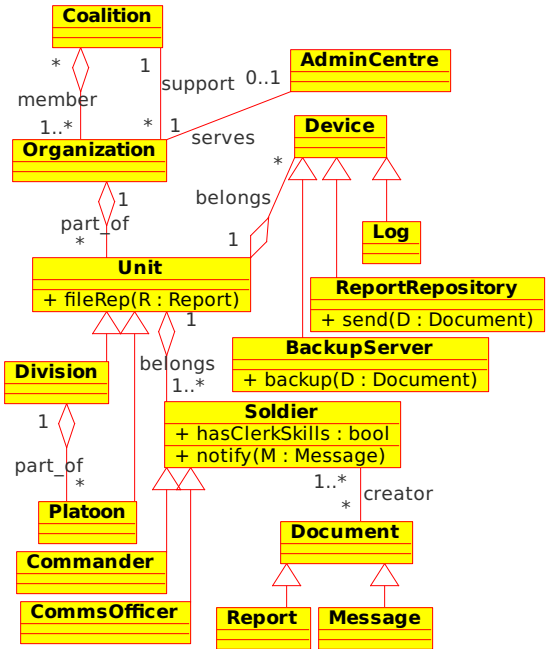

Fig. 1. UML structure for fragment of domain

There is no need to use the EC for static features of the system model, as they can be encoded directly in first-order logic, using the *class* predicate:

$$class(org) \quad class(device) \quad class(adminCentre)$$

To express the relationship between classes, we use *isa* for class specialization; *assType* for association; *compType* for composition; and *aggregType* for aggregation. The relevant relationships from Figure 1 would be represented according to the following model:

$$isa(division, unit) \qquad aggregType(org, part\_of, cltn)$$
$$isa(reportRepos, device) \qquad assType(org, supports, cltn)$$

Finally, *classOp* facts are included which describe the operations possible on instances of a given class, and the types of the arguments these operations take. For instance:

$$classOp(unit, fileRep, 1, [report])$$
$$classOp(backupServer, backup, 1, [document])$$

The first argument is the class name; the second, the operation name; the third, the arity; the fourth, a list of argument-types. To aid reasoning over the structure of our domain models, we introduce the following transitive version of the *isa* predicate:

$$isa\_trans(X, Y) \leftarrow isa(X, Y). \tag{1}$$
$$isa\_trans(X, Z) \leftarrow isa(X, Y),\ isa\_trans(Y, Z).$$

If there were a special kind of report repository called a 'mobile report repository', for instance, with an associated fact $isa(mobileRepRep, reportRepos)$ in our knowledge base, then we could conclude that this too was a device, using the transitivity of the *isa* relationship.

Specific instances of the above classes are needed for policy operationalization. For instance, there may be both UK and US organizations in a coalition; the US may have two divisions, *eastDivision* and *westDivision*, with devices, soldiers, commanders, and so on. In our example, all divisions and platoons have a commander, a communications officer, a log, report repository, and backup server. We consider this set-up to be stable over timescales of policy specification although our approach would allow for changes. Facts about instances of classes existing at a time T are represented, in the EC, as:

$$holdsAt(obj(ita, cltn), T)$$
$$holdsAt(obj(us, org), T)$$
$$holdsAt(obj(eastPlatOneComOf, commsOfficer), T)$$

We define a recursive clause for the *obj* fluent, using a new *obj_trans* fluent similar to the *isa* relationship above.

$$holdsAt(obj\_trans(O, C), T) \leftarrow holdsAt(obj(O, C), T). \tag{2}$$
$$holdsAt(obj\_trans(O, C), T) \leftarrow$$
$$holdsAt(obj(O, C'), T),\ isa\_trans(C', C).$$

This allows us to reason, for instance, that each platoon communications officer is also a soldier—required for an authorization policy giving access to a certain document to all soldiers in the Eastern division is to be correctly applied.

Relationships between instances of classes are depicted as follows (we show a representative sample).

$$holdsAt(aggreg(uk, member, ita), T)$$
$$holdsAt(aggreg(eastDivision, part\_of, us, T)$$
$$holdsAt(ass(usAdminCentre, serves, us), T)$$

Definition 1 A *class definition* $\mathcal{C}$ is a set of ground (containing no variables) instances of the predicates *class*, *isa*, *assType*, *aggregType*, *compType*, *classOp* together with the definition (1) of the *isa_trans* predicate, if that does not include cycles in the *isa* relation. We require that

- if $isa(c, c') \in \mathcal{C}$, then $class(c), class(c') \in \mathcal{C}$;
- if $assType(c, a, c') \in \mathcal{C}$, then $class(c), class(c') \in \mathcal{C}$;
- if $aggregType(c, a, c') \in \mathcal{C}$, then $class(c), class(c') \in \mathcal{C}$;
- if $compType(c, a, c') \in \mathcal{C}$, 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. ⌟

In this way, the definition of classes, and their associations and operations is self-contained. The *isa* relationship must hold between classes explicitly stated to exist; the type of associations, aggregations and compositions must be between known classes; and operations must be on known classes.

Definition 2 An *instance definition at* t, where t is a time variable or ground member of the *Time* sort, is a set of ground instances of *holdsAt*, all of whose second arguments are t, and whose fluents are instances of *obj*, *ass*, *aggreg* or *comp*. ⌟

The following defines the proper relationship between class and instance definitions. ($\models$ represents standard first-order semantic entailment.)

Definition 3 Let $\mathcal{C}$ be a class definition, and $\mathcal{I}$ an instance definition at time t. $\mathcal{I}$ is *correct with respect to* $\mathcal{C}$ if:

- If $\mathit{holdsAt}(\mathit{obj}(o, c), t) \in \mathcal{I}$ then $\mathit{class}(c) \in \mathcal{C}$
- If $\mathit{holdsAt}(\mathit{obj}(o, c), t), \mathit{holdsAt}(\mathit{obj}(o, c'), t) \in \mathcal{I}$ then $\mathcal{C} \models \mathit{isa\_trans}(c, c') \vee \mathit{isa\_trans}(c', c)$
- If $\mathit{holdsAt}(\mathit{ass}(o, a, o'), t) \in \mathcal{I}$, there are c, c' such that

$$\mathcal{C} \cup \mathcal{I} \models \mathit{holdsAt}(\mathit{obj\_trans}(o, c), t) \wedge \mathit{holdsAt}(\mathit{obj\_trans}(o', c'), t) \wedge \mathit{assType}(c, a, c')$$

- If $\mathit{holdsAt}(\mathit{aggreg}(o, a, o'), t) \in \mathcal{I}$, there are c, c' with

$$\mathcal{C} \cup \mathcal{I} \models \mathit{holdsAt}(\mathit{obj\_trans}(o, c), t) \wedge \mathit{holdsAt}(\mathit{obj\_trans}(o', c'), t) \wedge \mathit{aggregType}(c, a, c')$$

- If $\mathit{holdsAt}(\mathit{comp}(o, a, o'), t) \in \mathcal{I}$, there are c, c' such that

$$\mathcal{C} \cup \mathcal{I} \models \mathit{holdsAt}(\mathit{obj\_trans}(o, c), t) \wedge \mathit{holdsAt}(\mathit{obj\_trans}(o', c'), t) \wedge \mathit{compType}(c, a, c') \lrcorner$$

That is, in turn: if an object is said to belong to a given class, the class must be stated to exist; if an object belongs to two different classes, one must be a subclass of the other; if an object is associated with, aggregated to form, or composed of, another object, then the two objects must belong to classes between which the assocation, aggregation, or composition is possible.

In Section IV we introduce *refinement rules* to define the relationship between objects and actions seen at a higher level of abstraction, to their lower-level details, contents and implementations. These rules' forms will be constrained by the UML class structure of the domain to which they apply.

III. Policy Language

The first stage of policy refinement is to translate the most abstract representation of a policy into a formal language upon which automated refinement techniques can work. Accordingly, we assume that policies are specified, at the highest level of abstraction, in a structured natural language, having a constrained lexicon and syntax designed for policy expression; automated translation could then compile this into the abstract, logical language we describe below. Work on structured natural languages for policies is known [5].

We present a brief summary of the language we use to represent policies. This language is abstract, in the sense that it is not meant to serve as an implementable policy language, deployed into policy decision points; rather, it is intended to serve as a generic formal language into and out of which multiple policy languages can be translated. The base language is that of constraint logic programs—normal logic programs with constraints—and we use the constraints to order the times at which the conditions of the policy must be true. Policies are expressed as rules, with the head an instance of one of the predicates *permitted*, *denied* or *obligation*, and the body of the rule representing circumstances in which the policy applies. A fuller version of the language for authorization policies is in [1]. In the following, a *constraint* c is given by:

$$\begin{aligned} c &::= s_1 = s_2 \mid s_1 < s_2 \mid s_1 \leqslant s_2 \\ s &::= n \mid v \mid s_1 + s_2 \mid s_1 - s_2 \end{aligned}$$

$n \in \mathbb{R}^+ \cup \{0\}$, v is a variable. A *condition* is a literal of *holdsAt*, *happens*, *permitted*, *denied*, *obligation*, *do* or *req*.

The language is divided into two parts: one for the policies and one for the system the policies are used to control. There are bridges between these parts that model the role of a Policy Enforcement Point. Authorization policies take the form

$$[\mathit{permitted}/\mathit{denied}](\mathit{Sub}, \mathit{Tar}, \mathit{Act}, T) \leftarrow L_1, \ldots, L_m, C_1, \ldots, C_n. \tag{3}$$

where the L_i are conditions in the sense defined above, and the C_i are constraints, also as above. Where the conditions L_i are true, subject to the constraints C_i, then the *Sub* is *permitted* (*/denied* permission) to perform *Act* on *Tar* at time T. For example, consider the domain we sketched in Section II. A policy that *[Red Cross devices are permitted to access logs of coalition platoons]* might be formalized as

$$\begin{aligned} \mathit{permitted}(\mathit{Sub}, \mathit{Tar}, \mathit{read}, T) \leftarrow\ & \mathit{holdsAt}(\mathit{obj}(\mathit{Sub}, \mathit{device}), T), \\ & \mathit{holdsAt}(\mathit{ass}(\mathit{Sub}, \mathit{owner}, \mathit{redCross}), T), \\ & \mathit{holdsAt}(\mathit{obj}(\mathit{Tar}, \mathit{log}), T), \\ & \mathit{holdsAt}(\mathit{ass}(\mathit{Tar}, \mathit{owner}, C), T), \\ & \mathit{holdsAt}(\mathit{obj}(C, \mathit{org}), T), \\ & \mathit{holdsAt}(\mathit{ass}(C, \mathit{member}, \mathit{cltn}), T). \end{aligned} \tag{4}$$

In the conditions, the subject *Sub* is constrained to be a device owned by the Red Cross; the target must be a *log* belonging to a nation which is a member of the coalition. There are no constraints over T, so the permission applies at all times.

Obligations require an action to be performed within a certain period; thus, in addition to the time T at which the obligation holds, reference is needed to the times T_s and T_e which are the limits of the period in which the action should be performed. Our obligation policies have the form

$$\mathit{obligation}(\mathit{Sub}, \mathit{Tar}, \mathit{Act}, T_s, T_e, T) \leftarrow L_1, \ldots, L_m, C_1, \ldots, C_n. \tag{5}$$

The L_i and C_i are as before—conditions and constraints. These obligations have a more general form than event-condition-action (ECA) rules, as found in many systems, such as Ponder [6]. The form considered here is more common in higher-level specifications, allowing for actions performed by humans and more complex management actions (e.g. that schedule their own actions). ECA rules are a particular form of implementing these more general obligations; our refinement process can refine the obligations into the ECAs that implement them. This happens in one of two ways: either an obligation to an action *read* (for instance) is directly refined into an ECA rule to do the *read*; or else the obligation is interpreted more along the lines found in work on norm-governed systems, and the refinement is into ECA rules notifying the user of the obligation, and taking appropriate action when the obligation is fulfilled or violated.

As an example of an obligation policy, consider *[Platoon communications officers must file daily activity reports to the division their platoon belongs to between 2000h and 2200h].* This can be formalized as follows:

$$\begin{aligned} &\mathit{obligation}(\mathit{Sub}, \mathit{Tar}, \mathit{fileRep}(R), 2000h, 2100h, T) \leftarrow \\ &\quad \mathit{holdsAt}(\mathit{obj}(\mathit{Sub}, \mathit{commsOfficer}), T), \\ &\quad \mathit{holdsAt}(\mathit{ass}(\mathit{Sub}, \mathit{belongs}, P), T), \\ &\quad \mathit{holdsAt}(\mathit{obj}(P, \mathit{platoon}), T), \\ &\quad \mathit{holdsAt}(\mathit{obj}(\mathit{Tar}, \mathit{division}), T), \\ &\quad \mathit{holdsAt}(\mathit{ass}(P, \mathit{part\_of}, \mathit{Tar}), T), \\ &\quad \mathit{holdsAt}(\mathit{obj}(R, \mathit{report}), T), \\ &\quad \mathit{holdsAt}(\mathit{ass}(R, \mathit{reportType}, \mathit{dailySummary}), T), \\ &\quad 2000h < T \leqslant 2200h. \end{aligned} \tag{6}$$

In addition to obligation policies of the form above, the syntax allows for the specification of obligations defined for classes of objects, where any member of the class (as supposed to every member of the class as above) can fulfill the obligation. These obligations are not discussed further in the current paper.

An obligation on a subject to perform an action on a target is fulfilled when the action is done. An obligation is violated when the time at which it could have been fulfilled runs out; for instance, *fulfilled* is defined as follows:

$$\begin{aligned} &\mathit{fulfilled}(\mathit{Sub}, \mathit{Tar}, \mathit{Act}, T_s, T_e, T_{do}, T) \leftarrow \\ &\quad \mathit{do}(\mathit{Sub}, \mathit{Tar}, \mathit{Act}, T_{do}), \\ &\quad \mathit{obligation}(\mathit{Sub}, \mathit{Tar}, \mathit{Act}, T_s, T_e, T_{do}), \\ &\quad T_s \leqslant T_{do} < T_e,\ T_{do} \leqslant T. \end{aligned} \tag{7}$$

$\mathit{obligation}(\mathit{Sub}, \mathit{Tar}, \mathit{Act}, T_s, T_e, T)$ must be defined such that the constraints in its body imply $T \leqslant T_e$; however, the obligation may only be fulfilled at a time T_{do} before T_e.

IV. Refinement Rules

In defining the management and security policies of a system, it ought only to be necessary to specify the policies at the highest level of abstraction, and then use refinement techniques to map those higher-level policies, in successive phases, to the lowest level. Policies that refer generically to sensors as targets of configuration actions, for instance, may first be mapped to types of sensor (audio, or video, say). The policies derived for audio sensors may then be mapped onto policies for different models of audio sensors, with model-specific configuration operations. Finally, policies controlling configuration operations intended for a given model of audio sensor may be mapped into policies concerning messages sent to a certain port of all audio sensors of the relevant model known to be currently attached to a sensor network.

To achieve this gradual refinement, we use rules to relate subjects, targets and actions described abstractly, or described *en masse*, to the lower level. The basic components of such rules are known as *conditioned actions*—expressions which represent the performance of an action on a target by a subject in certain contexts, and where the contexts place constraints on the types of the subjects, targets and actions and the relationships between them.

Definition 4 Let $\mathcal{L}$ be a policy representation language.[1] A *conditioned action for* $\mathcal{L}$ has the form

$$(\mathit{Sub}, \mathit{Tar}, \mathit{Act}) : C_1, \ldots, C_n.$$

where each C_i has one of the forms $\mathit{obj}(O, C)$, $\mathit{ass}(O, A, O')$, $\mathit{aggreg}(O, A, O')$ or $\mathit{comp}(O, A, O')$. In each case, C must be a ground class name from the domain, and A must be a ground association, aggregation, or composition name from the domain. Any variable occurring in *Sub*, *Tar*, *Act* must also occur in one of the C_i; this constrains the refinement rule to apply only to objects of classes present in the UML diagram.⌟

For example, a conditioned action

$$\begin{aligned} &(\mathit{Sub}, \mathit{Tar}, \mathit{read}(\mathit{locDev}, \mathit{wQuad})) : \\ &\quad \mathit{obj}(\mathit{Sub}, \mathit{device}),\ \mathit{ass}(\mathit{Sub}, \mathit{owner}, X), \\ &\quad \mathit{obj}(\mathit{Tar}, \mathit{locationServer}),\ \mathit{ass}(\mathit{Tar}, \mathit{owner}, X). \end{aligned} \tag{8}$$

represents the performance of an action of reading information on the location of devices in the West Quadrant by devices which belong to the same organization as the location server from which the information is read. Such conditioned actions form the basic components of refinement rules.

Definition 5 Let $\mathcal{L}$ be a policy representation language, and $\mathcal{C}$ a class definition. A *refinement rule* is an expression

$$\mathsf{C} \quad \Rightarrow \quad \mathsf{C}_1 \ \mathbf{then} \ \cdots \ \mathbf{then} \ \mathsf{C}_n \tag{9}$$

where $\mathsf{C}, \mathsf{C}_1, \ldots, \mathsf{C}_n$ are conditioned actions, and $i \geqslant 1$. C is called the *top* of such a rule, and the C_i are *bottoms*. More precisely, where R has the form

$$\begin{aligned} &(\mathit{Sub}, \mathit{Tar}, \mathit{Act}) : \mathit{Conds} \Rightarrow \\ &\quad (\mathit{Sub}_1, \mathit{Tar}_1, \mathit{Act}_1) : \mathit{Conds}_1 \\ &\quad \mathbf{then} \ \ldots \\ &\quad \mathbf{then} \ (\mathit{Sub}_n, \mathit{Tar}_n, \mathit{Act}_n) : \mathit{Conds}_n \end{aligned} \tag{10}$$

1) There must be $\mathit{obj}(\mathit{Sub}, C)$ and $\mathit{obj}(\mathit{Tar}, C')$ in the conditions of C, such that for some N and L

$$\begin{aligned} \mathcal{C} \models\ & \mathit{class}(C) \wedge \mathit{class}(C') \wedge \mathit{classOp}(C''\mathit{Act}, N, L) \\ & \wedge \mathit{isa\_trans}(C', C'') \end{aligned}$$

and further, in each Conds_i, $1 \leqslant i \leqslant n$:

2. there is an expression $\mathit{obj}(\mathit{Sub}_i, C_i)$ (resp. $\mathit{obj}(\mathit{Tar}_i, C_i)$) in Conds_i, but not *Conds*, with $\mathcal{C} \models \mathit{isa\_trans}(C_i, C)$ and $\mathit{obj}(\mathit{Sub}, C)$ (resp. $\mathit{obj}(\mathit{Tar}, C)$) is in *Conds*—or $\mathit{Sub}_i = \mathit{Sub}$ (resp. $\mathit{Tar}_i = \mathit{Tar}$);
3. there is $\mathit{aggreg}(\mathit{Sub}_i, A, \mathit{Sub})$ (resp. $\mathit{aggreg}(\mathit{Tar}_i, A, \mathit{Tar})$ in Conds_i which is not in *Conds*, and such that $\mathcal{C} \models \mathit{aggregType}(C_i, A, C)$, $\mathit{Conds} \models \mathit{obj}(\mathit{Sub}, C)$ and $\mathit{Conds}_i \models \mathit{obj}(\mathit{Sub}_i, C_i)$—or else $\mathit{Sub}_i = \mathit{Sub}$ (resp. $\mathit{Tar}_i = \mathit{Tar}$);
4. there is $\mathit{comp}(\mathit{Sub}_i, A, \mathit{Sub})$ (resp. $\mathit{comp}(\mathit{Tar}_i, A, \mathit{Tar})$ in Conds_i but not in *Conds*, with $\mathcal{C} \models \mathit{compType}(C_i, A, C)$,

[1] See [2]. $\mathcal{L}$ defines the language used to specify both policies and the domains they operate on.

Conds $\models$ *obj*(*Sub*, C) and *Conds*$_i$ $\models$ *obj*(*Sub*$_i$, C_i)—or *Sub*$_i$ = *Sub* (resp. *Tar*$_i$ = *Tar*).

In addition, for each i such that $1 \leqslant i \leqslant n$

5. $\mathcal{C} \cup$ *Conds* $\cup$ *Conds*$_i$ $\models$ *classOp*(*Tar*$_i$, *Act*$_i$, N_i, L_i) for some N_i, L_i. ⌟

The meaning of such rules is that the succession of actions referred to in $\mathsf{C}_1, \ldots, \mathsf{C}_n$ correspond to a refinement of the higher-level action in C, with **then** intended to denote temporal sequence. The constraints 1–5 ensure that our refinement rules respect the UML system model.

Specifically, constraint 1 ensures that the subjects and targets of the high-level conditioned action appearing in the refinement rule must be objects of the domain, as defined by the UML; this constraint also forces the high-level action to be one of the operations the target supports. Constraints 2–4 ensure that, as we refine, we move to more specific subjects and targets; for each C_i that constitutes a lower-level action, either the subject or the target must be more specific, and neither should be *more* general or abstract. For instance, if the subject of the top of a refinement rule is defined to be a platoon, then the subject of the action into which we refine could be a soldier belonging to that platoon (this would be a case of aggregation). Or, the target might be constrained, in the top of the refinement rule, to be a sensor—a possible target of the bottom of the refinement rule could be an audio sensor, the sensor class being a generalization of audio sensors. Finally, constraint 5 ensures that the refinement rules are well-formed—the actions *Act*$_i$ of the lower-level parts of the rules must be operations it is possible to perform on the targets *Tar*$_i$.

As an illustration, consider an action of filing a daily activity report within the US military. We suppose the workflow for filing reports is predefined centrally within the US army, and the procedure must be followed throughout the organization. Seen at a more abstract perspective, soldiers file reports to units (not necessarily their own). However, for a soldier to file a report, a series of actions specified at a more concrete level must occur. He must first send the report to the report repository of the unit to which the report is to be filed. He must then backup the report to that unit's backup server. When this has been done, the soldier must notify the communications officer of the unit that the upload has been completed. The following refinement rule represents this picture:

$$
\begin{aligned}
&(\mathit{Sub}, \mathit{Tar}, \mathit{fileRep}(R)) : \\
&\quad \mathit{obj}(\mathit{Sub}, \mathit{soldier}), \mathit{obj}(\mathit{Tar}, \mathit{unit}) \\
&\quad\quad \Downarrow \\
&(\mathit{Sub}, \mathit{Tar}_1, \mathit{send}(R)) : \\
&\quad \mathit{obj}(\mathit{Tar}_1, \mathit{reportRepos}), \mathit{aggreg}(\mathit{Tar}_1, \mathit{belongs}, \mathit{Tar}) \\
&\textbf{then } (\mathit{Sub}, \mathit{Tar}_2, \mathit{backup}(R)) : \\
&\quad \mathit{obj}(\mathit{Tar}_2, \mathit{backupServer}), \mathit{ass}(\mathit{Tar}_2, \mathit{belongs}, \mathit{Tar}) \\
&\textbf{then } (\mathit{Sub}, \mathit{Tar}_3, \mathit{notify}(\mathit{upload}(R))) : \\
&\quad \mathit{obj}(\mathit{Tar}_3, \mathit{commsOfficer}), \mathit{ass}(\mathit{Tar}_3, \mathit{belongs}, \mathit{Tar})
\end{aligned}
\tag{11}
$$

Note that the same variable *Sub* occurs as subject of all actions, both in the top and bottom of the rule: the same soldier must send the report, then back it up and make the notification. However, the target varies: from a report repository, to a backup server, to a communications officer.

In [7] we used concepts from data integration to formulate rules and algorithms for the refinement of targets and actions, for the same broader goal of policy refinement. Data integration concerns the way in which heterogeneous databases are mapped to each other so their vocabularies (schemata) are inter-translatable. Refinement rules as we define them here are a generalization of that previous work.

V. Policy Decomposition

In this section, we examine in detail the decomposition stage of policy refinement. This itself divides into two parts. *Matching* verifies whether a refinement rule is applicable to a policy; and *decomposition* proper performs the refinement.

A. Matching

Recall policy (6). In order to refine this using rule (11), variables are first renamed so as no longer to be shared between policy and refinement rule. Then, an attempt is made to unify the head of the top of the refinement rule—in our example, (*Sub*, *Tar*, *fileRep*(R))—with the corresponding features of the head of the policy. Let variables in the policy (6) which are shared with the refinement rule be given a mark $'$ to separate them; the subject, target and action can be matched using the most general unifier (m.g.u.) $\theta = \{\mathit{Sub}/\mathit{Sub}', \mathit{Tar}/\mathit{Tar}', R/R'\}$.

Next, a test is made to determine whether the constraints on the action in the body of the top of the refinement rule—in our example, *obj*(*Sub*$'$, *soldier*) and *obj*(*Tar*$'$, *unit*) with θ applied—are consistent with the conditions on the subject, target, and action found in the policy. The purpose of this consistency check is to ensure that there is an overlap between the circumstances in which the refinement rule and policy apply. The consistency check is made as follows. Let *Conds* be the set of conditions in the top of the refinement rule, and *Conds*$_P$ be the set of *holdsAt* conditions in the body of the policy, with θ applied. Let *Conds*$^*_P\theta$ be the result of stripping the *holdsAt*$(\cdot,\cdot)$ predicates from the members of *Conds*$_P\theta$, leaving the fluents. In our case, we have

$$
\begin{aligned}
\mathit{Conds}\theta &= \{\mathit{obj}(\mathit{Sub}', \mathit{soldier}), \mathit{obj}(\mathit{Tar}', \mathit{unit})\} \\
\mathit{Conds}^*_P\theta &= \{\mathit{obj}(\mathit{Sub}', \mathit{commsOfficer}), \mathit{aggreg}(\mathit{Sub}', \mathit{belongs}, U_s), \\
&\qquad \mathit{obj}(U_s, \mathit{platoon}), \mathit{obj}(\mathit{Tar}', \mathit{division}), \\
&\qquad \mathit{aggreg}(U_s, \mathit{part\_of}, \mathit{Tar}'), \mathit{obj}(R', \mathit{report}), \\
&\qquad \mathit{ass}(R', \mathit{reportType}, \mathit{dailySummary})\}
\end{aligned}
$$

We now check *Conds*θ and *Conds*$^*_P\theta$ for logical consistency, with respect to the background UML formalization. (The details are given below, in Definition 6.) If this phase succeeds, then the refinement rule is known to be applicable to the policy.

Definition 6 Let $\mathcal{L}$ be a policy representation language, $\mathcal{C}$ a class description. Given an obligation policy P

$$\mathit{obligation}(\mathit{Sub}', \mathit{Tar}', \mathit{Act}', T_s, T_e, T) \leftarrow \mathit{Conds}_P, \mathit{Cons}_P.$$

or an authorization policy

$$[\mathit{permitted}/\mathit{denied}](\mathit{Sub}', \mathit{Tar}', \mathit{Act}', T) \leftarrow \mathit{Conds}_P, \mathit{Cons}_P.$$

where Conds_P are literals and Cons_P are constraints, and a refinement rule R of the form (10), whose variables have been renamed to be different, we say that P and R *head-match with* θ if there is an m.g.u. θ such that $(\mathit{Sub}, \mathit{Tar}, \mathit{Act})\theta = (\mathit{Sub}', \theta', \mathit{Act}')\theta$. Further, if P and R head-match on θ, let Conds^*_P be defined as

$$\{F \mid \exists T(\mathit{holdsAt}(F,T) \in \mathit{Conds}_P,\ F \text{ is } \mathit{obj}, \mathit{ass}, \mathit{aggreg} \text{ or } \mathit{comp}\}$$
$$\cup \{\neg F \mid \exists T(\textbf{not } \mathit{holdsAt}(F,T) \in \mathit{Conds}_P\ F \text{ is an } \mathit{obj}, \mathit{ass}, \mathit{aggreg} \text{ or } \mathit{comp} \text{ fluent}\}$$

Let $\mathcal{X}$ be $\mathcal{C} \cup \mathit{Conds}^*_P \cup \mathit{Conds}$. R is said to *match* P *on* θ if

$$\forall o[\mathit{obj}(o,c), \mathit{obj}(o,c') \in \mathcal{X} \rightarrow \models \mathcal{X} \rightarrow (\mathit{isa\_trans}(c,c') \leftrightarrow \neg \mathit{isa\_trans}(c',c))] \quad \lrcorner$$

If a refinement rule matches a policy on some θ according to Definition 6, then the rule can be applied to the policy during policy refinement.

B. Policy Decomposition

After a successful matching process, the policy is decomposed. Part of the conditions defining the scope of the refined policies comes from the original policy, part from the top of the refinement rule, and part from the conditioned actions in the bottom of the refinement rule. The circumstances in which the refined policy applies will be the intersection of these three.

Given conditioned actions $\mathsf{C}_1, \ldots, \mathsf{C}_n$ in the bottom of the refinement rule, n copies of $P\theta$ are made, with the subject, target and action of $P\theta$ replaced by those of each $\mathsf{C}_i\theta$. Conditions from the top of the refinement rule that are not implied by the conditions of the original policy are added to each of these copies. Conditions from each C_i are added to the policy. Conditions are added deriving from each conditioned action C_i that the actions at the head of each C_j, $j < i$, have been performed, and in the right order.

Returning to our running example, let *PolicyConditions* be:

$$\begin{aligned}
&\mathit{holdsAt}(\mathit{obj}(\mathit{Sub}', \mathit{commsOfficer}), T),\\
&\mathit{holdsAt}(\mathit{aggreg}(\mathit{Sub}', \mathit{belongs}, U_s), T),\\
&\mathit{holdsAt}(\mathit{obj}(U_s, \mathit{platoon}), T),\\
&\mathit{holdsAt}(\mathit{obj}(\mathit{Tar}', \mathit{division}), T),\\
&\mathit{holdsAt}(\mathit{ass}(U_s, \mathit{part_of}, \mathit{Tar}'), T),\\
&\mathit{holdsAt}(\mathit{obj}(R', \mathit{report}), T),\\
&\mathit{holdsAt}(\mathit{ass}(R', \mathit{reportType}, \mathit{dailySummary}), T),
\end{aligned}$$

These are the literals that will be inherited from the higher-level policy. The results for our example are shown below (the source of the various conditions in the bodies of the policies has been marked to their left). The policy stemming from the first conditioned action, C_1, is:

$$\begin{aligned}
&\mathit{obligation}(\mathit{Sub}', \mathit{Tar}_1, \mathit{send}(R'), 2000h, 2200h, T) \leftarrow \qquad (12)\\
&\quad \mathsf{C}_1 \left[\begin{array}{l} \mathit{holdsAt}(\mathit{obj}(\mathit{Tar}_1, \mathit{reportRepos}), T),\\ \mathit{holdsAt}(\mathit{aggreg}(\mathit{Tar}_1, \mathit{belongs}, \mathit{Tar}'), T), \end{array} \right.\\
&\quad \text{policy} \left[\begin{array}{l} \mathit{PolicyConditions},\\ 2000h < T \leqslant 2200h. \end{array} \right.
\end{aligned}$$

The policy coming from the second conditioned action, C_2:

$$\begin{aligned}
&\mathit{obligation}(\mathit{Sub}', \mathit{Tar}_2, \mathit{backup}(R'), T_1, 2200h, T) \leftarrow \qquad (13)\\
&\quad \mathsf{C}_2 \left[\begin{array}{l} \mathit{holdsAt}(\mathit{obj}(\mathit{Tar}_2, \mathit{backupServer}), T),\\ \mathit{holdsAt}(\mathit{aggreg}(\mathit{Tar}_2, \mathit{belongs}, \mathit{Tar}'), T), \end{array} \right.\\
&\quad \text{policy} \left[\begin{array}{l} \mathit{PolicyConditions},\\ T_1 < T < 2200h, \end{array} \right.\\
&\quad \mathsf{C}_1 \left[\ \mathit{fulfilled}(\mathit{Sub}', \mathit{Tar}_1, \mathit{send}(R'), 2000h, 2200h, T_1, T). \right.
\end{aligned}$$

Finally, the policy derived from C_3 is:

$$\begin{aligned}
&\mathit{obligation}(\mathit{Sub}', \mathit{Tar}_3, \mathit{notify}(\mathit{upload}(R')), T_2, 2200h, T) \leftarrow \qquad (14)\\
&\text{from } \mathsf{C}_3 \left[\begin{array}{l} \mathit{holdsAt}(\mathit{obj}(\mathit{Tar}_3, \mathit{commsOfficer}), T),\\ \mathit{holdsAt}(\mathit{aggreg}(\mathit{Tar}_3, \mathit{belongs}, \mathit{Tar}'), T), \end{array} \right.\\
&\quad \text{policy} \left[\begin{array}{l} \mathit{PolicyConditions},\\ 2000h < T_1 < 2200h,\\ T_1 < T_2 < 2200h, \end{array} \right.\\
&\quad \mathsf{C}_1 \left[\ \mathit{fulfilled}(\mathit{Sub}', \mathit{Tar}_1, \mathit{send}(R'), 2000h, 2200h, T_1, T), \right.\\
&\quad \mathsf{C}_2 \left[\ \mathit{fulfilled}(\mathit{Sub}', \mathit{Tar}_2, \mathit{backup}(R'), T_1, 2200h, T_2, T). \right.
\end{aligned}$$

Some description of what has happened here is appropriate. We focus on the third derived policy. The action is determined by the conditioned action

$$\begin{aligned}
&(\mathit{Sub}, \mathit{Tar}_3, \mathit{notify}(\mathit{upload}(R))) :\\
&\quad \mathit{obj}(\mathit{Tar}_3, \mathit{commsOfficer})\\
&\quad \mathit{ass}(\mathit{Tar}_3, \mathit{belongs}, \mathit{Tar})
\end{aligned}$$

from the rule (11). The first set of conditions in the derived policy's body is inherited from the specific conditioned action in the bottom of the action-refinement rule: these define the nature of the target of the derived policy (it is this target, and the action in the head of the policy, that are altered from the high-level policy). Next, there are conditions come from the high-level policy: these define the subject which must perform the lower level *notify*$(\cdot)$ action, the kind of report referred to, and the relationship of the subject to platoons and divisions.

There are two additional sets of conditions in (14). These require that, for the obligation policy to hold, the obligations derived from other conditioned actions in the action decomposition rule must have been fulfilled. This is a result of the structure of the refinement rule: one should only notify of an upload, once the report has been sent and then the report has been backed up. In general, given a refinement rule $\mathsf{C} \Rightarrow \mathsf{C}_1$ **then** $\cdots$ **then** C_n, the ordering of actions is crucial. An obligation to do the action of C is an obligation to do the actions of the $\mathsf{C}_1, \ldots, \mathsf{C}_n$ in order, and so C_i must be performed only if C_j, for $j < i$, have been correctly executed. To return to our example policy, the times between which the action that the policy requires must be performed, are T_2 and $2200h$. $2200h$ is the final time from the original high-level policy; T_2 is the time at which the obligation derived from the previous conditioned action C_2 was fulfilled.

The refinement of positive authorization policies (with *permitted* in their heads) is very similar. The only difference concerns the temporal constraints on when the derived policies hold. In an obligation policy, an interval is given, within which the action must be performed; when the policy is refined, we

have interpreted this to mean that the entire sequence of lower-level actions must be performed in the interval, in the correct order. But with authorization policies, in general, a reference is made in the head of the policy to a single time at which the action is authorized. We therefore adopt the convention that it is the *final* action in the sequence that is authorized at this time. The high-level policy can be seen as authorizing the production of certain effects (those which the high-level action achieves), and these effects are only guaranteed to be achieved when the entire sequence of actions as been performed.

For negative authorization policies (with *denied* in their head), a slightly different tactic is taken. The denial of a high-level action is the denial of permission to execute a sequence, but one might interpret this to mean, either the denial of permission to perform *every* action in the sequence, or denial to perform only *some* of these actions, selected according to some appropriate criterion. We take the latter approach, and have chosen in the current work to disallow the *final* action of the sequence, as the one that achieves the sequence's completion. However, there is no reason why this choice of the way in which to refine negative authorization policies should not be made available to the user. Note that the final action of a sequence is denied, in this model, only if the previous actions of the sequence have already been performed; this qualification is achieved by conditions in the body of the derived policy.

Definition 7 Let P be either an obligation or authorization policy, of any of the forms

$$\begin{aligned} &\mathit{obligation}(\mathit{Sub}', \mathit{Tar}', \mathit{Act}', T_s, T_e, T) \leftarrow \mathit{Conds}_P, \mathit{Cons}_P, \\ &[\mathit{permitted}/\mathit{denied}](\mathit{Sub}', \mathit{Tar}', \mathit{Act}', T) \leftarrow \mathit{Conds}_P, \mathit{Cons}_P. \end{aligned}$$

Let R be a refinement rule $\mathsf{C} \Rightarrow \mathsf{C}_1$ **then** $\cdots$ **then** C_n. C has the form $(\mathit{Sub}, \mathit{Tar}, \mathit{Act})$: Conds, and each C_i has the form $(\mathit{Sub}_i, \mathit{Tar}_i, \mathit{Act}_i)$: Conds_i. Assume that R matches P with m.g.u. θ. Recall the meaning of Conds^*_P from Definition 6. Matching ensures that Conds and Conds^*_P are consistent with respect to the UML class definitions. Let $\mathit{holdsAt}(\mathit{Conds})$ be defined as the set $\{\mathit{holdsAt}(F, T) \mid F \in \mathit{Conds}\}$ and let $\mathit{holdsAt}^*(\mathit{Conds})$ is defined to be

$$\{\mathit{holdsAt}(F, T) \mid F \in \mathit{Conds} \text{ and } \mathit{Conds}_P \cup \mathcal{C} \not\models \mathit{holdsAt}(F, T)\}$$

(i) if P is an obligation policy, then the refinement of P w.r.t. R is the set of policies

$$\begin{aligned} [\quad &\mathit{obligation}(\mathit{Sub}_i, \mathit{Tar}_i, \mathit{Act}_i, T_i, T_e, T) \leftarrow \\ &\mathit{Conds}_P,\ \mathit{holdsAt}^*(\mathit{Conds}), \\ &\mathit{Ful}f_1, \ldots, \mathit{Ful}f_{i-1}. \quad]\theta \end{aligned} \tag{15}$$

Where $\mathit{Ful}f_j$, for $1 \leqslant j < i$ is

$$\mathit{fulfilled}(\mathit{Sub}_j, \mathit{Tar}_j, \mathit{Act}_j, T_{j-1}, T_j, T_e, T). \tag{16}$$

In the case where $j = 1$, the condition T_{j-1} is replaced by T_s. The $\mathit{Ful}f_j$ expressions ensure that the previous actions in the series of those required have been correctly performed.

(ii) if P is a positive authorization policy, then the refinement of P w.r.t. R is the set of policies

$$\begin{aligned} [\quad &\mathit{permitted}(\mathit{Sub}_i, \mathit{Tar}_i, \mathit{Act}_i, T) \leftarrow \\ &\mathit{Conds}_P,\ \mathit{holdsAt}^*(\mathit{Conds}), \\ &\mathit{Done}_1, \ldots, \mathit{Done}_{i-1}. \quad]\theta \end{aligned} \tag{17}$$

Done_j, for $1 \leqslant j < i$ is

$$\begin{aligned} &\mathit{do}(\mathit{Sub}_j, \mathit{Tar}_j, \mathit{Act}_j, T_j), \\ &T_{j-1} < T_j < T_j, \\ &\mathit{holdsAt}^*(\mathsf{C}_j). \end{aligned}$$

Where $j = 1$, there is no temporal constraint; where $j = i-1$, the constraint is $T_{i-1} < T$.

(iii) finally, if P is a negative authorization policy, then the refinement of P with respect to R is

$$\begin{aligned} [\quad &\mathit{denied}(\mathit{Sub}_n, \mathit{Tar}_n, \mathit{Act}_n, T_n) \leftarrow \\ &\mathit{Conds}_P, \\ &\mathit{holdsAt}^*(\mathit{Conds}), \\ &\mathit{Done}_1, \ldots, \mathit{Done}_{n-1}. \quad]\theta \end{aligned} \tag{18}$$

The definition of *Done* is the same as for *permitted*. ⌟

This definition formalizes the preceding discussion.

C. Multiple refinement rules

In a given scenario, be true that many different refinement rules match a given policy. This could be the case when there are a number of heterogenous devices, which implement an action specified at a higher level of abstraction in different ways. For example, if a generic *configure* action applies to devices designated as *sensors*, this might be refined to an *audio sensor*'s having its sampling rate set, and to an *image sensor*'s having its resolution set. There would be two different refinement rules, for each type of sensor, but both may match the same policy, and a single policy should be refined using both simultaneously, so the policy may be implemented on the PDPs applying both to audio sensors and image sensors. This ability of a single higher-level policy to be refined in multiple different ways is one of the advantages of our approach.

VI. Operationalization and Distribution

Operationalization in software engineering is the process of assigning specific resources for the performance of goals. In the context of policy refinement, we understand it as the selection of named entities for the execution of policies.

In a centralized refinement model, we would expect the following arrangements. There would be a set of policy refinement rules of the form given by (5), together with a class definition $\mathcal{C}$, and an *instance repository* which stores information about the instances of classes known to exist, and their relationships (instance repositories can be represented as instance definitions, as in Section II). A policy to be refined enters the refinement component, and the refinement rules are iteratively applied to it—more than one refinement rule may, of course, apply to a given policy. When policies have reached the point that no more refinement rules are applicable to them, then their form is tested against a subset of the language which

is known to describe concretely implementable components of the domain: class names of actual rather than abstract devices, and the lowest level of concrete description of the properties of those devices. If the policies are expressed in this "implementable" subset of the policy representation language, then they may be operationalized.

Authorization policies grant or deny the permission to perform an action on a target; their operationalization accordingly means grounding the targets referred to in the heads of rules. If the target is not currently ground, then those literals in the body of the authorization policy which include the variable appearing in the head of the rule as target are collected, together with literals transitively sharing variables with them. These are then passed, as a query, to the database holding the current instance repository. Answers return supply a set of ground values for the target of the policy; it is to those targets that the policy is distributed, with the ground target replacing the target variable in the rule.

In the case of obligation policies, the last stage of refinement transforms the policy into ECA-type policies which (i) will notify subjects of the obligation of its imminence, and (ii) take any necessary action if the obligation is fulfilled or violated. Operationalization here means the selection of the appropriate policy monitors to carry out these notifications and responses.

In a fully distributed refinement scenario, partially-refined policies may be distributed to other refinement-centres, with local knowledge of devices and their refinement rules. Phases of decomposition, operationalization and distribution are then made locally.

VII. Related Work

Policy refinement remains one of the most ambitious goals in policy-based security management because it aims to automate the realization of high-level requirements in executable implementations. Early studies (e.g. [8]) highlighted some of the specific issues and have attempted to address them in a very restricted way. However, there has been renewed interest in this problem in recent years as demonstrated by the panel discussion at IM 2007 and several recent studies [9], [10], [11], [12] which address subsets of the problem such as goal decomposition for refinement or transforming specifications using predefined mappings. [13] considers a planning approach to refining change requests into implementable tasks, but does not consider policies. In [14] a goal is essentially a utility function which can be optimised to select a set of parameters for policies used to configure a sensor network. Our approach is much more general than these rather specialised forms of policy refinement.

VIII. Conclusion

In this paper, we have described the process of action decomposition in a policy refinement framework we are developing. Future work must address the formalization of operationalization and distribution, as well as the full integration of distributed policy refinement with policy analysis.

We are testing the approach for specifying policies relating to inter-organizational collaborations and the use of sensor networks. We intend to develop tools to help non-technical users refine their goals into 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] 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.

[2] 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.

[3] R. Craven, J. Lobo, E. Lupu, J. Ma, A. Russo, and M. Sloman, "Distributed policy scenario," ITA Technical Report, 2010.

[4] R. A. Kowalski and M. J. Sergot, "A logic-based calculus of events," *New Generation Comput.*, vol. 4, no. 1, pp. 67–95, 1986.

[5] C. Brodie, C.-M. Karat, and J. Karat, "An empirical study of natural language parsing of privacy policy rules using the sparcle policy workbench," in *SOUPS*, ser. ACM International Conference Proceeding Series, L. F. Cranor, Ed., vol. 149. ACM, 2006, pp. 8–19.

[6] 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.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] D. Trastour, R. Fink, and F. Liu, "ChangeRefinery: Assisted Refinement of High-Level IT Change Requests," *Policies for Distributed Systems and Networks, IEEE International Symposium on*, vol. 0, pp. 68–75, 2009.

[14] G. Campbell and K. Turner, "Goals and policies for sensor network management," *Sensor Technologies and Applications, International Conference on*, vol. 0, pp. 354–359, 2008.