Toward a Formal Characterization of Policy Specification & Analysis

Arosha Bandara*, Seraphin Calo†, Jorge Lobo†, Emil Lupu‡, Alessandra Russo‡, Morris Sloman‡
*The Open University, Milton Keynes, MK7 6AA, UK - Email: a.k.bandara@open.ac.uk
†IBM Research, IBM T J Watson Research Center, NY, USA - Email: {scalo, jlobo}@us.ibm.com
‡Imperial College London, London SW7 2AZ, UK - Email: {e.c.lupu, a.russo, m.sloman}@imperial.ac.uk

***Abstract*—Policy-based management of the security of a military communications network can simplify the configuration process, while increasing security and availability. An effective policy-based approach requires analysis of policies for inconsistencies, and for desired security properties. It also must provide for the refinement of high-level security goals into concrete policies. This paper defines a language based on first-order logic formulae containing explicit time arguments which is expressive enough for specifying a range of authorization and obligation security policies, while supporting the formalisms and automated tools needed for analysis and refinement. Both system behavior and the semantics of the policies themselves are defined in terms of execution traces, to enable reasoning about algorithmic solutions to policy analysis. The paper also proposes some analysis tools based on the use of logical abduction.**

I. Introduction

Secure, reliable and adaptable communications is needed to support dynamic mission-based coalitions of partners from different military and non-military organizations. If the wrong information is communicated to the wrong person/device, it could cost the lives of the personnel involved in the mission. Likewise, if the right information is not communicated and shared with the right people, it could also lead to loss of lives. Policy-based security management should enable military personnel to specify security requirements in terms of simple, intuitive goals which are translated into the concrete system settings in such a way that the system behaves in a consistent and desirable way. The objective is to minimize the technical expertise required by military personnel, and to automate policy management as far as possible. This is dependent on being able to specify and analyze policies to ensure that they prescribe correct and desirable behavior. For example, inconsistencies should not arise because the available communication devices cannot support the specified policies.

We assume that military personnel specify goals using a structured natural language aimed at non-technical people. Goals are automatically translated into a formal, logic-based abstract language for refinement and analysis. Our past experience has indicated that logic languages, while good for reasoning, are not amenable to efficient implementation, particularly on small hand-held devices. Thus abstract policies must be translated into concrete implementable policies described in languages such as Ponder[1] or XACML[2].

This paper focuses on the intermediate abstract policy language that requires a powerful logic-based formalism with available reasoning tools, based on which techniques for aiding policy analysis and refinement can be developed. Our previous work in this area used a formalization of policies based on the Event Calculus [3], but there was no a priori consideration of the complexity and computability properties. As a result, it was necessary to reverse engineer various restrictions into the language in order to ensure tractability of the analysis and refinement procedures. The more foundational approach of [4] and [5] uses formalisms for policy with well defined complexity results, based on static policy models where access control decisions do not depend on temporal properties of the system. Additionally, their work does not support obligation policies, which are often required to implement security mechanisms. These limitations restrict the type of policies that can be expressed, and preclude the use of their approach for practical policy-based security management applications.

The work presented in this paper addresses these shortcomings by defining a language expressive enough to deal with a range of security policy requirements, and tractable to facilitate automation of the analysis process. Our technical approach is based on developing a foundational treatment of the behavior of policies, and using this as a basis to reason about algorithmic solutions to policy analysis. We will also extend this to deal with refinement as future work. System behavior is defined in terms of execution traces of inputs, states and outputs of the system at each logical point in time. We use the input, state and output symbols in these traces to define first-order logic formulae containing explicit time arguments to specify authorization and obligation policies. The semantics of these formulae are defined in terms of the system execution traces. Our formulae contain explicit time arguments, rather than standard temporal logic, in order to keep our language in the realm of logic programs. This allows us to use abductive reasoning to analyze policy specifications, as well as being more expressive than using temporal logic operators [6].

Research was sponsored by the U.S. Army Research Laboratory and the U.K. Ministry of Defence and was accomplished under Agreement Number W911NF-06-3-0001. The views and conclusions contained in this document are those of the author(s) 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. The U.S. and U.K. Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

The paper is organized as follows: Section 2 defines basic terms and concepts used in our formal policy language; Section 3 gives the syntax and semantics of this language followed by examples to demonstrate its capabilities and limitations in section 4; Section 5 discusses the analysis capabilities of our language together with some directions for future work; Section 6 presents some related work on formalizing security policies; and, finally section 7 presents our conclusions and summarizes our plans for developing this approach further.

II. Definitions

A system is said to comply with a policy if it acts according to the behavior specified by the policy. A simple characterization of the system is thus needed to determine whether it complies with the policy.

We will assume that at any time a system can be in one of a given set of *states*, with system behavior defined in terms of how the system transitions through states over time. Systems interact with the world by receiving *inputs* which may result in state changes and the generation of *outputs*. Inputs come from external entities and cannot be controlled by the system. Thus, we characterize the behavior of a system in terms of its state transitions and outputs, in reaction to its inputs.

Formally, we have three sets $\mathcal{I}, \mathcal{S}, \mathcal{O}$ of *Inputs*, *States* and *Outputs*, and define the set of traces $\mathcal{T}$ with respect to $\mathcal{I}, \mathcal{S}, \mathcal{O}$ as the set of finite or infinite sequences of triples $(i_0, s_0, o_0), (i_1, s_1, o_1), \ldots$, where $i_k \in \mathcal{I}, s_k \in \mathcal{S}, o_k \in \mathcal{O}$ for any $k \in N$. A *system* Sys is defined (characterized) by a subset of $\mathcal{T}$, $Sys \subseteq \mathcal{T}$, the set of all its possible behaviors.

Example 1: As an example, consider a secure communications system which accepts as inputs requests to transmit the location information of specific military personnel. Outputs of the system would be transmissions of the requested location information or denials if the policy for the current state of the system does not permit the user to perform the request.

We assume a typed first order logic language $\mathcal{L}$ with typed variables, constant and function symbols, and with a set of typed predicate symbols partitioned into three sets I (*input predicate symbols*), S (*state predicate symbols*) and O (*output predicate symbols*). Function and predicate symbols have an assigned *arity* which is a non-negative integer. A *term* is recursively defined as a variable, a constant, or $f(t_1, \ldots, t_n)$ if f is a function symbol of arity n and each t_k is a term of the appropriate type. $p(t_1, \ldots, t_n)$ is an *atomic formula* or *atom* if p is a predicate symbol of arity $n \geq 0$ and each t_k is a term of the appropriate type. Formulae can be formed using the standard logical connectors, $\neg$, $\wedge$, $\vee$, ... and the quantifiers $\forall$ and $\exists$. An *input formula* only has input predicate symbols and similarly for *State* and *output* formulae . A *ground* term does not have variable occurrences and a (atomic) formula has no variable occurrences.

For a typed first order language $\mathcal{L}$ we define an input i in $\mathcal{I}_\mathcal{L}$ to be any subset of the atomic ground input formulas in $\mathcal{L}$; a state s in $\mathcal{S}_\mathcal{L}$ to be any subset of the atomic ground state formulas in $\mathcal{L}$ and an output o in $\mathcal{O}_\mathcal{L}$ to be any subset of the atomic ground input formulas in $\mathcal{L}$. When this is clear from the context we will drop any reference to $\mathcal{L}$.

Example 2: Extending example 1, we define an input predicate symbol: req; a single state predicate symbol: $dataProperty$, and two output predicate symbols: do and $deny$. We have three constant symbols: $alice$, bob, $charlie$ and $david$; and three function symbols: tx, $securityLevel$ and $location$. A few elements in $\mathcal{I}_\mathcal{L}$, $\mathcal{S}_\mathcal{L}$, $\mathcal{O}_\mathcal{L}$:

$$
\begin{array}{l}
i_0 = \{req(alice, tx(bob, location(charlie), securityLevel(high)))\}, \\
s_0 = \{dataProperty(location(charlie), securityLevel(high))\}, \\
o_0 = \{do(alice, tx(bob, location(charlie), securityLevel(high)))\}, \\
i_1 = \{req(bob, tx(david, location(charlie), securityLevel(low)))\}, \\
s_1 = \{dataProperty(location(charlie), securityLevel(low))\} \\
o_1 = \{deny(bob, tx(david, location(charlie), securityLevel(low)))\}
\end{array}
$$

A possible trace of our sample system could be $(i_0, s_0, o_0), (i_1, s_1, o_1)$.

Since the purpose of a policy is to characterize the compliant behaviors of a system we will formally define policies as restrictions over the possible traces $\mathcal{T}$. Hence, a policy P defines a subset of acceptable traces $mod(P)$ that model the policy: $mod(P) \subseteq \mathcal{T}$.

Example 3: In our example if we have a policy where Bob is not allowed to send Alice's location information to Charlie, then there cannot be a trace with a triple (i, s, o) in which an atom of the form $do(bob, tx(charlie, location(alice), securityLevel(X))$ is in o, with X either equal to $high$ or low.

We can now formally define policy compliance.

Definition 1: A system Sys complies with a policy P iff $Sys \subseteq mod(P)$. We will denote this by $Sys \models P$.

In practice, we start with a system whose behavior is modified by policies which may change over time. The *domain description* is a core set of system constraints, that are invariants on the system behavior regardless of changes due to policies. For example, if the communications system supports a super user who is allowed to transmit any data item in any situation, policies cannot force changes to this core constraint. The domain description also defines the set of operations that the system supports and a policy cannot introduce new operations, or dictate the input behavior of the system. In other words, although the domain description may say something about input behavior, policies cannot modify it. In essence a domain description defines systems that are correct.

Definition 2: Given a set of traces $\mathcal{T}$, a domain description D is a subset of the power set of $\mathcal{T}$: $D \subseteq 2^\mathcal{T}$. We say that a system Sys behaves according to the domain D iff $Sys \in D$.

Example 4: An input/ouput automata (e.g., a Mealy machine) can be used to describe the domain of a system. Figure 1 shows a automaton representing the domain of the communication system described previously. The transitions in this state chart show that traces in D can contain only triples $\{(\{req(\ldots, tx(\ldots))\}, \{dataProperty(\ldots)\}, \{do(\ldots)\}),$ $(\{req(\ldots, tx(\ldots)\}), \{dataProperty(\ldots)\}, \{deny(\ldots)\})\}$ where each trace can be obtained by instantiating the variables using the values from the finite sets - in this example, the set of users, locations and security levels.

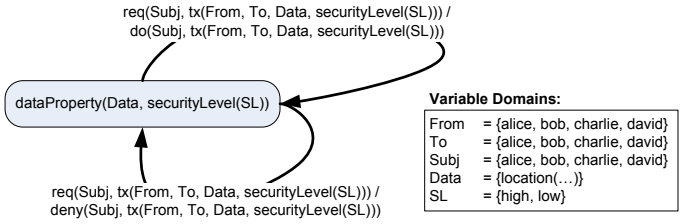

Fig. 1. Finite state automaton representing the domain description of the example communications system

Domain descriptions dictate valid policies, i.e., if there is no system from the domain description that complies with the policy then the policy cannot be implemented.

Definition 3: Given a domain description D, a policy P is consistent in D if there is $Sys \in D$ such that $Sys \models P$; otherwise P is inconsistent.

In our communications system example it is easy to see that if we want a system to comply with a policy in which *Alice* can send $contactInfo(\dots)$ data then there is no behavioral trace of the system that can implement this policy since $contactInfo(\dots)$ is not a type of data specified in the domain description. We can also define policy conflicts in terms of inconsistencies.

Definition 4: Given a domain description D, and a collection of policies $P_1, \dots, P_n$ such that each P_i is consistent with D, the policies are in conflict if there is no system $Sys \in D$ such that $Sys \models P_i$ for every $1 \leq i \leq n$.

As described above, policies constrain the possible traces of the system in the same way as the domain descriptions. Therefore, we can represent policies in a similar fashion in terms of restrictions on the allowable traces in D.

Example 5: For the communications system example, consider a policy that states that "Alice is allowed to transmit location data that has a low security level to Bob if she uses a high security level for the transmission, but Bob is not allowed to transmit location data to anyone else". This natural language policy could be enforced using the following concrete policies in a system implementing a language like Ponder2 [1]:

Policy 1:
permit $Alice \rightarrow$
$Transmit(Alice, Bob, location(X), secLevel(high))$
when $secLevel(location(X), low)$
Policy 2:
prohibit $Bob \rightarrow Transmit(Bob, *, location(*), *)$

In this case, the first policy only allows Alice to transmit high security location data (e.g. $do(Transmit(Alice, Bob, location(X) \dots))$) in the output of a trace only if the security level of the transmission method is high. The second policy prohibits the output trace from indicating any transmissions of location data from Bob.

In the next section we introduce a language to formally specify security policies, i.e., a language to define sets of acceptable traces. Policies will be described using two types of statements: *authorizations* and *obligations*.

III. Language specification

Before going into the details of the syntax and semantics of the language it may be helpful to explain the notation used. Throughout the paper, constants, functions, and predicate symbols begin with a lower-case letter, and variables begin with an upper-case letter. The symbol T will generally refer to a variable of type *Time*, the symbol $\vec{X}$ will be used to denote a tuple of variables of (possibly different) types, not including *Time*, the symbol $\vec{x}$ to denote a tuple of ground terms of types different from *Time*, and the symbol σ will denote a trace. The symbols $\top$ and $\bot$ are used to denote respectively the constant Boolean values *True* and *False*. We assume the type *Time* to be given by the set N of natural numbers. We will, therefore, use $0, 1, 2, 3, \dots, k, \dots$, for $k \geq 0$ to denote constants of type *Time*, standard functions $+$, $-$, $\times$, to construct terms of type *Time* and relations $=, \neq, \leq, <, \geq, >$ to define relations over *Time*. Greek-letter meta-variables are used to refer in general to terms and expressions in the language.

A. Authorizations

There is a general consensus that authorizations typically identify four different entities: a subject to whom the authorization is being granted, an action (possibly with some parameters) that defines the right being granted (i.e. the subject is authorized to execute the action), a target where the action will be executed and an optional condition that must be verified before granting the right to the subject.

Example 6: Alice can delete classified data files from her device if she sends a notification to the supplier of the data 10 minutes in advance and the supplier does not respond to the notification asking Alice to retain the file.

There are three operations or actions mentioned in the policy: $notify$, $delete$ and $retain$. We appeal to the reader's intuition for the attributes of the predicates and operations that will be used through the examples. The subject of the authorization is $alice$. The target is Alice's $device$ where the file resides. To specify authorization policies we will make use of the following domain independent predicates:

1) $req(Subject, Target, Action, Time)$
2) $do(Subject, Target, Action, Time)$
3) $deny(Subject, Target, Action, Time)$
4) $permitted(Subject, Target, Action, Time)$
5) $denied(Subject, Target, Action, Time)$

req is an input predicate symbol, do and $deny$ are output predicate symbols, and $permitted$ and $denied$ are state predicate symbols. Intuitively the time argument in all the predicates can be interpreted as the point in a trace where the predicate is being evaluated. The example makes use of another group of subjects: $Suppliers$ of data. In general subjects can also be targets as in the case of $Suppliers$ which are targets of $notify$. The following is a domain dependent predicate needed for the specification:

1) $filedesc(Supplier, Name, Type, Time)$

The policy can be (partially) described by the following rule:

$$do(alice, S, notify(delete(F)), T_0) \wedge filedesc(S, F, class, T_0) \wedge \\ \textbf{not}\ req(S, F, retain(F), T_1) \wedge T_0 \leq T_1 \leq T_2 \wedge T_2 - T_0 \geq 10mins \\ \rightarrow permitted(alice, device, delete(F), T_2) \tag{III.1}$$

The **not** in the formula has a special (logic programming-like) meaning: for a given sub, tar, act and t and trace σ, if $req(sub, tar, act)$ is not i_t where (i_t, s_t, o_t) is the t^{th} tuple of σ then **not** $req(sub, tar, act, t)$ will hold in σ. The trace σ complies with the policy if for all possible values of s, o, f, t_0, t_1 and t_2 of the variables S, O, F, T_0, T_1 and T_2 respectively, such that $do(alice, s, notify(o, delete(f)) \in o_{t_0}$, $filedesc(s, f, calss) \in S_{t_0}$, $t_0 \leq t_1 \leq t_2$, $t_2 - t_0 \geq 10mins$ and $req(s, f, retain(f), t_1)$ holds in σ then $permitted(alice, device, delete(f)) \in s_{t_2}$. A system Sys complies with this policy if every trace in Sys complies with the policy.[1]

For availability we need to make sure that if the request to execute an action appears in the trace of a system and the subject making the request is permitted to execute the action the action is executed. We achieve that by adding the following *domain independent* rule to our policies, which we call the *basic availability policy rule*:

$$req(Sub, Tar, Act, T) \wedge permitted(Sub, Tar, Act, T) \\ \rightarrow do(Sub, Tar, Act, T)$$

Before we define basic policies we need to introduce the concept of time constraints. An expression C of the form $\tau_1\ op\ \tau_2$ where τ_i is either a constant or variable of type *Time* or an arithmetic expression built using $+$, $-$, *Time* constants and *Time* variables, and op is one of the operators $=, \neq, <, \leq, >, \geq$, is referred to as a *time constraint*.

Definition 5: A basic *positive authorization policy rule* is a formula of the form:

$$L_1 \wedge L_n \wedge C_1 \wedge C_m \rightarrow permitted(S, R, A, T)$$

where all the following properties hold:

1) $n \geq 0$ and $m \geq 0$.
2) Each L_i is either an atom or an atom preceded by **not** extended with an extra *Time* argument.
3) No predicate in the L_is can be $permitted$.
4) Each C_i is a time constraint.
5) All variables appearing in the time constraints must also appear in a non-time constraint atom in the rule.
6) For the extra argument T_i of each L_i it must hold that $T_i \leq T$ either implicitly because they are part of the constraints or because $C_1 \wedge \cdots \wedge C_m$ implies the relation.
7) If the predicate in L_i is do the relation must be $T_i < T$.
8) S R, A, T are terms of type Subject, Target, Action and Time respectively.

A *basic policy* Π is a set of positive authorization policy rules together with the basic availability policy rule.

[1]Depending on the granularity of *Time* the expression $10mins$ will be replaced by the right constant.

The constraints in the time arguments are imposed to ensure that the permission to execute an action in a particular state of a trace does not depend on "future" properties of the trace. Condition (3) makes policies hierarchical logic programs which can be evaluated in linear time with respect to the size of the set of policy rules. We will partially lift this condition later in the paper.

Given a policy rule P, a ground instance of P is a policy rule in which all the variables in P are simultaneously replaced by ground terms of the appropriate type from $\mathcal{L}$. Let $ground(P)$ be the set of all ground instances of P, and for a policy Π, let $ground(\Pi)$ be the union of all $gound(P)$ for all $P \in \Pi$. To formally define the set of traces specified by a policy we will first define when our time extended predicates are modelled by a trace.

Definition 6: Let $\mathcal{L}$ be a sorted first-order language whose sorts include the set of natural numbers N as *Time* sort. Let Σ be the first-order relational signature of $\mathcal{L}$, given by $\Sigma = I \cup O \cup S$. A ground atom $p(\vec{x}, k)$ is a *time extended* ground atom in $\mathcal{L}$ if $p(\vec{x})$ is a ground atom in $\mathcal{L}$ and k is a constant from *Time*. Let $\sigma = (i_0, s_0, o_0), (i_1, s_1, o_1), \ldots$ be a trace in $\mathcal{T}$ and let ϕ be a time extended ground atom in $\mathcal{L}$. The satisfiability of ϕ at position $k \geq 0$ of the trace σ is defined as follows:

- $\sigma, k \models p(\vec{x}, k)$ iff $p(\vec{x}) \in i_k$, where p is an input predicate symbol
- $\sigma, k \models p(\vec{x}, k)$ iff $p(\vec{x}) \in s_k$, where p is a state predicate symbol
- $\sigma, k \models p(\vec{x}, \vec{t}, k)$ iff $p(\vec{x}, \vec{t}) \in s_k$, where p is a state predicate symbol
- $\sigma, k \models p(\vec{x}, k)$ iff $p(\vec{x}) \in o_k$, where p is an output predicate symbol

If $\phi = p_1(\vec{x}_1, k_1) \wedge \cdots \wedge p_n(\vec{x}_n, k_n)$, then $\sigma \models \phi$ iff $\sigma, k_i \models p_i(\vec{x}_i, k_i)$, for every i, $1 \leq i \leq n$.

Definition 7: Let $C = i\ op\ j$ be a ground time constraint where op is one of the operators $=, <, \leq, >, \geq$, and let σ be a trace in $\mathcal{T}$. C is said to be satisfied in σ, denoted as $\sigma \models C$, if and only if i and j are positions in σ such that $i\ op\ j$. If either i or j are arithmetic expressions they are evaluated with the usual meaning.

Formally, the set of traces specified by a basic policy is characterized by a special class of traces called *supported* traces, which are defined as follows.

Definition 8: Let Π be a policy. A trace $\sigma = (i_0, s_0, o_0), (i_1, s_1, o_1), \ldots$ is called *supported by* Π if for each $k \geq 0$, for every ground atomic literal $A \in s_k$ or $A \in o_k$ that can be formed with predicate symbols from the set $\{do, permitted\}$ there exists ϕ such that $\phi \rightarrow A \in ground(\Pi)$ and $\sigma \models \phi$.

The traces in $mod(\Pi)$ for a set $\Pi = P_1, \ldots, P_n$ of policy rules will be supported traces σ satisfying each policy rule P_i. If we consider the policy rule (III.1) with the basic availability policy rule, a trace containing a tuple (i_k, s_k, o_k) such as $do(alice, supplier, notify(delete(file))) \in o_k$ is only supported if

$$permitted(alice, supplier, notify(delete(file))) \in s_k$$

since only the basic availability policy rule can support

$$do(alice, supplier, notify(delete(file)))$$

Definition 9: Let Π be a policy and σ be a supported trace by Π. The trace σ *satisfies* Π, denoted $\sigma \models \Pi$, if and only if for every $\phi \rightarrow A \in ground(\Pi)$ such that $\sigma \models \phi$, then $\sigma \models A$. Let $mod(\Pi) = \{\sigma | \sigma \models \Pi\}$.

Because of its direct correspondence to hierarchical logic programs we can implement monitors that can evaluate policies in linear time with respect to the size of the policy.

Note that the policy in our example does not specify how to treat a request for a $notify$. The definition of a supported trace for the policy does not let any subject do a $notify$. A possibility would be to permit all actions not explicitly described by basic policy rules. Before introducing such a rule we must close the permits for $delete$, i.e. indicate which $delete$ operations are denied, using the following rule:

$$\begin{array}{l} filedesc(Sub, Sup, F, class, T) \wedge \\ \textbf{not } permitted(Sub, system, delete(F), T) \\ \quad \rightarrow denied(Sub, system, delete(F), T) \end{array}$$

We are leaving open $delete$ actions that might be applied to non-classified data. We can now close the permits for all the actions by replacing the basic availability rule with the new domain independent availability rule:

$$\begin{array}{l} req(Sub, Tar, Act, T) \wedge \textbf{not } denied(Sub, Tar, Act, T) \\ \quad \rightarrow do(Sub, Tar, Act, T) \end{array}$$

If we care about denying actions, e.g. for auditing purposes, we can also add the following domain independent rule:

$$\begin{array}{l} req(Sub, Tar, Act, T) \wedge denied(Sub, Tar, Act, T) \\ \quad \rightarrow deny(Sub, Tar, Act, T) \end{array}$$

We refer to this rule as the *negative availability* rule and refer to the new availability rule as the *positive availability* rule. We are not limited to using $denied$ to close permits. We can directly describe prohibitions using $denied$ rules:

$$location(warzone, T) \rightarrow denied(Sub, Tar, notify(A), T)$$

We have informally introduced negative authorization policies. In formal terms, negative authorizations are defined as follows.

Definition 10: A basic *negative authorization policy rule* is a formula of the form:

$$L_1 \wedge L_n \wedge C_1 \wedge C_m \rightarrow denied(S, R, A, T)$$

where all the following properties hold:

1) $n \geq 0$ and $m \geq 0$.
2) Each L_i is either an atom or an atom preceded by **not** extended with an extra *Time* argument.
3) No predicate in the L_is can be $denied$ but it could be $permitted$.
4) Each C_i is a time constraint.
5) All variables appearing in the time constraints must also appear in a non-time constraint atom in the rule.
6) For the extra argument T_i of each L_i it must hold that $T_i \leq T$ either implicitly because they are part of the constraints or because $C_1 \wedge \cdots \wedge C_m$ implies the relation.
7) If the predicate in L_i is do the relation must be $T_i < T$.
8) S R, A, T are terms of type Subject, Target, Action and Time respectively.

A *basic authorization policy* Π is extended to be a set of positive and negative authorization policy rules together with the positive availability rule.

To cover the semantics of negative authorizations we only need to add to the set $\{do, permitted\}$ of supported traces the predicate symbol $denied$, and if the negative availability rule is also part of the policy the predicate symbol $deny$.

With the possibility of negative and positive authorizations we can reach states in supported traces in which both $permitted(sub, tar, act)$ and $denied(sub, tar, act)$ apply for a given subject sub, target tar and action act. The positive availability rule gives priority to $denied$ (i.e. $denied$ overrides $permitted$). To give priority to $permitted$ we just need to reintroduce the basic availability rule and add $notpermitted(Sub, Tar, Act, T)$ to the condition of the negative availability rule. Furthermore, we can have a more granular control over these priorities. We could, for example, have different default rules for different actions. We just need to specify the given action in the availability rules.

There are two ways in which we can allow $permitted$ and $denied$ in the conditions of positive authorizations and $denied$ in the conditions of negative authorizations: (1) Given a literal L_i with such a predicate, similar to do, we make sure that the time argument T_i in $L_i$obeys the constraint $T_i < T$; and (2) We can find a total order over the actions such that the action argument in both L_i and in the $permitted$ or the $denied$ after "$\rightarrow$" is not a variable. Furthermore, the action in L_i must precede the action in the $permitted/denied$. Similar order constraints can be imposed on subjects and targets. These are easy properties to check and the computational complexity of policy evaluation remains linear w.r.t. to the policy set size.

B. Obligations

There are many classes of obligations. In this paper we will limit our treatment to obligations acquired by a subject to execute an action. The subject could be an entity external to the system such as when a user is allowed to execute an action with the condition that she accepts the obligation to execute another action later on. Obligations can also be imposed in parts of the system, such that the system itself takes the responsibility of executing the action. In the former case the system cannot enforce the obligation, it can only monitor whether the obligations have been fulfilled.

Example 7: A node must provide a second identification within 5 minutes of establishing a connection to the wireless server; otherwise the server will drop the connection.

This example covers two obligations, one by the node making the connection, the second by the server which we will consider part of the system, that needs to drop the connection if the node does not fulfill its obligations.

In addition to the previously identified domain independent predicates for describing authorizations, we will introduce the following predicates for obligations:

1) $obl(Subject, Target, Action, T_1, T_2, Time)$. The subject is the entity acquiring the obligation to invoke the action on the target. T_1 and T_2 are the limits of the interval within which the obligation must be fulfilled.
2) *fulfilled*$(Subject, Target, Action, Time)$
3) *violated*$(Subject, Target, Action, Time)$

We can write the two obligations in the example as follows:

$$node(U,T) \wedge do(U, server, connect(U, server), T) \\ \rightarrow obl(U, server, submit2ID(U, server), T, T + 5min, T)$$

$$violated(U, server, submit2ID(U, server), T) \\ \rightarrow obl(server, server, disconnect(U, server), T, T + 1, T)$$

The first rule assigns obligations to nodes. Here a node is any entity that can request a connection to the server. The node is identified by the domain description predicate $node$. This predicate can be generated by the system by predefined authentication mechanism. The second rule assigns an obligation to the server and the server needs to invoke an action on itself to fulfill the obligation. We define two policy independent rules for *fulfilled* and *violated*:

$$obl(Subject, Target, Action, T_1, T_2, T) \wedge \\ do(Subject, Target, Action, T) \wedge T_1 \leq T \leq T_2 \\ \rightarrow fulfilled(Subject, Target, Action, T) \quad \text{(III.2)}$$

$$obl(Subject, Target, Action, T_1, T_2, T) \wedge T > T_2 \\ \rightarrow violated(Subject, Target, Action, T) \quad \text{(III.3)}$$

Note that for an obligation to be violated it needs to exist after the upper limit of the interval T_2. This allows us to drop obligations before they are fulfilled if needed. We will add a completion rule for obligations stating that obligations persist unless they are revoked, fulfilled or violated:

$$obl(Subject, Target, Action, T_1, T_2, T) \wedge T \leq T_2 \wedge \\ \textbf{not } revoke(Subject, Target, Action, T) \wedge \\ \textbf{not } fulfilled(Subject, Target, Action, T) \wedge \\ \textbf{not } violated(Subject, Target, Action, T) \\ \rightarrow obl(Subject, Target, Action, T_1, T_2, T + 1) \quad \text{(III.4)}$$

Definition 11: An *obligation policy rule* is a formula of the form:

$$L_1 \wedge L_n \wedge C_1 \wedge C_m \rightarrow obl(S, R, A, T_l, T_h, T)$$

where all the following properties hold:

1) $T_l \leq T_h$
2) $n \geq 0$ and $m \geq 0$.
3) Each L_i is either an atom or an atom preceded by **not** extended with an extra *Time* argument.
4) Each C_i is a time constraint.
5) All variables appearing in the time constraints must also appear in a non-time constraint atom in the rule.
6) For the extra argument T_i of each L_i it must hold that $T_i \leq T$ either implicitly because they are part of the constraints or because $C_1 \wedge \cdots \wedge C_m$ implies the relation.
7) If the predicate in L_i is do the relation must be $T_i < T$.
8) If the predicate in L_i is obl the relation between the time arguments must be $T_i < T$; otherwise there is a total order over the actions (resp. subjects/targets) such that both the action A (resp. subject S/target T) in the obl predicate in the right hand side of the implication and the action (resp. subject/target) argument in L_i are not variables and the action (resp. subject/target) in L_i precedes the action in the right hand side of the implication in the order.
9) S R, A, T are terms of type Subject, Target, Action and Time respectively.

An *obligation policy* is a finite set of obligation rules together with (III.2), (III.3) and (III.4).

We extend our definition of supported traces to cover obligation policies as follows.

Definition 12: Let Π be a policy. A trace $\sigma = (i_0, s_0, o_0), (i_1, s_1, o_1), \ldots$ is called *supported by* Π if for each $k \geq 0$, for every ground atomic literal $A \in s_k$ or $A \in o_k$ that can be formed with predicate symbols from the set $\{do, permitted, denied, obl, revoke, fulfilled, violated\}$ there exists ϕ such that $\phi \rightarrow A \in ground(\Pi)$ and $\sigma \models \phi$.

The rest of the definitions remain the same and the evaluation of policies because of the hierarchical structure of the rules is still linear with respect to the size of the set of policies.

Before moving to the examples we would like to mention that although we have limited the constraints to time, extending the constraints other domains (e.g. $Sec\_level_1 < Sec\_level_2$) is simple. We will work out the details in extensions of our work.

IV. Example Policies

In this section we present a number of policies that relate to a scenario involving a coalition search and rescue mission (*sar*) inspired by the Holistan vignettes [7]. The mission is led by a US special operations team (*us_spec*) and involves two partners, a UK medical team (*uk_medic*) and a team from the Holistan National Guard (*hng*). The mission commander (*mc*) is the leader of the *us_spec* team and will be responsible for generating orders and transmitting them to the other partners. Some of these orders, e.g., ***o1:****"Move to intercept enemy at grid location G"* will be classified as *secret* whereas others will be *unclassified*. This classification also applies to intelligence reports available to mission partners.

The communications system for the mission uses an adhoc network to transmit messages. The system is integrated with a policy based security management framework that ensures that the security policies of the mission are satisfied.

For this scenario, we have selected policies that satisfy a range of security requirements, from simple access control to separation of duty. For each policy, we present a natural language and formal definition before discussing the formal language features being used:

Example 8: Mission personnel who are permitted to read mission orders are allowed to read mission intelligence within 12 hours of the mission

This policy is an authorization rule that depends on another permission and domain specific predicates $mission(M,T)$, $orders(O,M,T)$, $intel(I,M,T)$, and $startTime(M,ST,T)$. We would express this rule in our formalism as follows:

$$\begin{array}{l}\forall O,M : mission(M,T_1) \wedge orders(O,M,T_1) \wedge \\ intel(I,M,T_2) \wedge permitted(Subject,O,read,T_1) \wedge \\ startTime(M,T_3,T_1) \wedge T_1 < T_2 < T_3 \wedge \\ (T_3 - T_2) < 12hrs \rightarrow \\ permitted(Subject,I,read,T_2)\end{array} \quad \text{(IV.1)}$$

Despite having $permitted$ in both the head and body of the rule, the above policy is acceptable in our formalism because the $permitted$ predicate in the body specifies a target that is disjoint from the target used in the head predicate.

Example 9: If a UK medical team member is obliged to transmit secret orders to the hng team, he must inform the mission commander of this action within 10 minutes.
This policy is an example of the need for obligation policies as part of a security policy definition because the commander needs to be kept informed of information being disclosed in order to evaluate the risk to the mission. The formal definition of this policy is as follows:

$$\begin{array}{l}\forall T,O,M : mission(M,T) \wedge orders(O,M,T) \wedge \\ obl(uk\_team,hng,transmit(O),T_1,T_2,T) \wedge \\ classify(O,secret) \wedge T_1 < T_2 < T \wedge (T_3 - T) < 10mins \rightarrow \\ obl(uk\_team,mc,transmit(sent(hng,O)),T,T_3,T)\end{array} \quad \text{(IV.2)}$$

This is another example of a policy that requires the same predicate symbol, in this case *obl*, in both the head and body of the rule. Once again this can be safely expressed in our language because at least one of the Subject, Target, and Action sets in the two *obl* predicates are disjoint.

Example 10: The mission commander is not allowed to both authorize and command a mission.
This is an example of a policy that ensures that particular functions are not performed by the same subject, i.e., the policy enforces a separation of function constraint. The formal definition of this policy is as follows:

$$\begin{array}{l}\forall T,M : mission(M,T) \wedge permitted(mc,M,authorise,T) \rightarrow \\ \quad denied(mc,M,command,T) \\ \forall T,M : mission(M,T) \wedge permitted(mc,M,command,T) \rightarrow \\ \quad denied(mc,M,authorise,T)\end{array} \quad \text{(IV.3)}$$

Example 11: The mission commander is not allowed to command a mission that he has authorized.
This is an example of a separation of duty policy where a subject is prevented from performing a particular action if he has performed some other conflicting action previously. This can be formally defined in our language as follows:

$$\begin{array}{l}\forall T,M : mission(M,T) \wedge \\ do(mc,M,authorise,T) \wedge T_1 > T \rightarrow \\ denied(mc,M,command,T_1).\end{array} \quad \text{(IV.4)}$$

Example 12: If the mission commander is a colonel, he is allowed to both authorize and command missions but he is not permitted to both authorize and command missions in the same sector of the city.
This is an example of a *Chinese Wall* policy where a subject is prohibited from performing an action on a particular target if he has already performed some other conflicting action on another target. The formal definition of this policy would be as follows:

$$\begin{array}{l}\forall T,M_1,\exists S : mission(M_1,T) \wedge mission(M_2,T) \wedge \\ \quad sector(M_1,S,T) \wedge \\ \quad \forall M_2 : sector(M_2,S,T) \wedge role(colonel,Subj,T) \wedge \\ \quad\quad \textbf{not} permitted(Subj,M_2,authorize,T) \rightarrow \\ \quad\quad permitted(Subj,M_2,command,T) \\ \forall T,M_1,\exists S : mission(M_1,T) \wedge mission(M_2,T) \wedge \\ \quad sector(M_1,S,T) \wedge \\ \quad \forall M_2 : sector(M_2,S,T) \wedge role(colonel,Subj,T) \wedge \\ \quad\quad \textbf{not} permitted(Subj,M_2,command,t) \rightarrow \\ \quad\quad permitted(Subj,M_2,authorize,t)\end{array} \quad \text{(IV.5)}$$

In this case, the need for universal quantifiers in the body of the rule means that these rules are not covered by the semantics of the language currently defined. Introducing such quantifiers while maintaining the computational complexity properties of the language is a challenge to be addressed in our future work.

V. Policy Analysis

Using our formalism, policies are always consistent, as it is not possible to construct a supported trace in $D \cap mod(\Pi)$ that satisfies both $permitted(sub,tar,act,t)$ and $\textbf{not}\ \ permitted(sub,tar,act,t)$, or $denied(sub,tar,act,t)$ and $\textbf{not}\ \ denied(sub,tar,act,t)$, for a given subject sub, target tar, action act and time t. Similarly for obligation policies. However, other forms of analysis and conflict detection, different from classical inconsistency, can be performed on a given set Π of authorization and obligation rules, such as *coverage* analysis, *modality* and *application specific* conflicts. *Coverage* refers to having polices that cover all cases of interest, e.g. Alice has the appropriate rights at the appropriate time. *Modality* conflicts can be of different types. They can be between authorization rules, when Π accepts, at least one, supported trace that satisfies both $permitted(sub,tar,act,t)$ and $denied(sub,tar,act,t)$ at the same time point t, for the same subject sub, target tar and action act; or they can occur between authorization and obligation rules. A conflict between authorizations and obligations may occur when the policy Π accepts a supported trace that satisfies $obl(sub,tar,act,t_1,t_2,t)$, at some time point t, $t_1 \leq t \leq t_2$, and at the same time satisfies $denied(sub,tar,act,t)$. *Application specific* conflicts, are, instead, conflicts that arise because the management actions being performed are incompatible with each other, as for instance, it is the case of *conflict of duty* (known also as the requirement to ensure separation of duties). Analysing a given set Π of policies for any of these types of conflicts can be translated into checking whether Π verifies series of properties, each indicating that Π free of a particular conflict. For instance, a property P can be $\forall T : \textbf{not}\ (permitted(s,t,a,T) \wedge denied(s,t,a,T))$, which states that at every time point, a given subject s should never be both permitted and denied to perform an action a on a

target t. The analysis task that we are concerned with, in this paper, is therefore to discover whether a given set Π of policies satisfies a given series of conflict-free defining properties P_i, and if not why not. Thus, for each property P_i we need to evaluate whether

$$D \cap mod(\Pi) \models P_i \tag{V.1}$$

and generate appropriate diagnostic information if not. We make use of an *abductive* approach [8], called *abduction in refutation mode* [9], whereby the analysis task in (V.1) is translated into an equivalent problem of showing that it is not possible to consistently extend Π with assertions that a particular set Δ of inputs may actually occur in such a way that $D \cap mod(\Pi \cup \Delta) \models \neg P_i$. We solve this latter task but attempting to compute such a Δ using a complete abductive proof procedure [10]. The completeness of this procedure is guaranteed by the hierarchical structure of our policy rules. If the procedure finds such a Δ then the assertions in Δ acts as an example of modality conflict and therefore counter-example to the given property P_i. The counter-examples that our approach generates describe inputs occurring in a sub-class of traces that must themselves satisfy the given property P_i. This is ensured by considering P_i as an integrity constraint on the form of possible traces, which prunes the set of possible counter-examples. A detailed description of the particular abductive proof procedure used here can be found in [10].

The properties that we are able to analyze are properties of the form $\forall T.P(T)$ where P is a formula with only one time variable T. For instance, consider a *coverage* analysis of the policy in III.1. Let's verify that Alice is able to delete a file. The property to check is $\forall T : permitted(alice, device, delete(file), T_2))$. The abductive proof procedure would compute the input set $\Delta = \{req(alice, bob, notify(file), T_0)$, **not** $req(bob, file, retain(file), T_1)\}$ for time points T_0, T_1 and T_2 such that $T_0 \leq T_1 \leq T_2$ and $T_2 - T_0 \geq 10min$, provided that it is not possible to prove $req(bob, file, retain(file), T)$ for any $T_0 \leq T \leq T_2$.

Analyzing a set Π of authorization policies for *modality* conflicts is defined as showing that the following property holds for any given subject s, target t and action a

$$D \cap mod(\Pi) \models \forall T : Time \\ \textbf{not } (permitted(s, t, a, T) \wedge denied(s, t, a, T))$$

This property is treated as a safety property and the procedure reduces to just find two arbitrary time points - a time t_1 before a violation of the above property arises and a subsequent time t_2 when the conflict occurs. The procedure checks whether it is possible to identify a set Δ of input formulae of the form **(not)** $req(s, t, a, t_1)$ such that

$$D \cap mod(\Pi \wedge \Delta) \models permitted(s, t, a, t_2) \wedge denied(s, t, a, t_2)$$

If such a computation, which in the case of hierarchical rules always terminates, fails then the authorization policy Π can be assumed to have no modality conflict (i.e. the conflict has been refuted). On the other hand, if the abductive proof procedure computes such a Δ, Π is said to *imply a modality conflict*. It is then necessary to show that it is indeed possible to construct a trace $\sigma \in D$ that satisfies $\Pi \wedge \Delta$. Such trace will then satisfy $permitted(s, t, a, t_2) \wedge denied(s, t, a, t_2)$ and provide an example of modality conflict. To illustrate this type of analysis with a simple example, consider the two rules

$$req(Sub, Tar, notify(A), T) \rightarrow \\ permitted(Sub, Tar, notify(A), T + 1) \\ location(warzone, T) \rightarrow denied(Sub, Tar, notify(A), T)$$

The abductive procedure would compute the input $\{req(sub, tar, notify(a), t)\}$. This, together with Π and D, would give $permitted(sub, tar, notify(a), t) \wedge denied(sub, tar, notify(a), t)$, provided that D satisfies $location(warzone, t)$. Any system trace supported by Π that satisfies $req(sub, tar, notify(a), t)$ would then be an example of modality conflict.

Similar technique can be applied for the other modality conflicts involving obligations. In particular we can verify the following property for a given subject s, target t and action a:

$$D \cap mod(\Pi) \models \forall T : \textbf{not } (obl(s, t, a, t_1, t_2, T) \wedge \\ t_1 \leq T \leq t_2 \wedge denied(s, t, a, T))$$

which shows that the obligation of subject s to execute action a cannot be fulfilled because the subject does not have the right to execute a at the appropriate times.

Abductive reasoning can also be used to compute traces that lead to *application* specific conflicts. Taking Example IV.4 we can verify if the separation of function constraint specified by this policy is violated by checking the following property for a given subject s. In this example the target is any $mission(M)$ and the actions are $authorise$ and $command$:

$$D \cap mod(\Pi) \models \forall T, M \\ \textbf{not } (permitted(s, M, authorise, T) \wedge mission(M, T) \wedge \\ permitted(s, M, command, T))$$

Finally, using our formalism we can check if a given set of policies satisfies a desired security property. For example, to check that only mission commanders have permission to read mission orders we try to look for violations by trying to find a subject p and a time t and Δ that to verify the following:

$$D \cap mod(\Pi \wedge \Delta) \models \\ \textbf{not } role(p, commander, t) \wedge \\ permitted(p, missionOrders, read, t)$$

In summary, the ability to check these types of property addresses a range of analysis needs and the use of abductive reasoning for this purpose has the added advantage of providing the sequence of input events that lead to a property violation.

VI. Related Work

Amongst the many alternative approaches to policy specification, there are a number of proposals for formal, logic-based notations. In particular logic-based languages have proved attractive for the specification of security policies but they can

be difficult to use and are not always directly translatable into efficient implementations. The work presented in this paper bears some similarities to our previous work on formalizing policies using Event Calculus [3]. Both approaches advocate modelling the temporal properties of the managed system and they both support authorization and obligation policies. However, because of its emphasis on developing tool support for policy analysis and refinement, our prior work did not focus on defining the computational complexity properties of the formalism at the outset. In contrast, this paper develops a formalism for policy-based security management with well defined semantics and computational complexity properties.

The Lithium language of Halpern and Weissman [4] has taken a more foundational approach by developing formalisms for policy that have well defined complexity results. However, these results are based on static policy models where access control decisions do not depend on temporal properties of the system. Additionally, there is no support for obligation policies which are often required to implement security mechanisms.

Irwin et al. propose a formalism for obligation policies together with analysis techniques [11]. In this paper, we have adapted the syntax of these obligation policies to produce a more general language that allows more complex policy rules to be expressed. However, the hierarchical structure of the rules in our language ensure that it is still computationally tractable, and that it is capable of supporting analyses such as the *strong accountability* checking presented in [11].

Other formal languages take advantage of the computational efficiencies obtained by using subsets of first order logic, such as stratified logic. Barker proposed a language that supports specification of access control policies using stratified clause-form logic, with emphasis on RBAC policies [12]. However, this work does not address conflict detection in policy specifications. The Authorization Specification Language (ASL) of Jajodia et al. [5], [13] is another example of a language based on stratified clause-form logic that also offers techniques for detecting modality conflicts and some application specific conflicts in authorization policy specifications. However, this language does not model temporal properties of the managed system, nor does it have support for obligation policies, both of which are provided in our language. Further, the analysis capabilities of ASL do not allow static analysis of policy specifications that use constraints, assuming instead that conflicts will be detected at runtime.

VII. Conclusions & Future Work

In this paper we have presented our initial work on a formal characterization of policy analysis, together with a formal language for specifying both authorization and obligation policies. Our technical approach is based on a foundational treatment of the behavior of policies, and on using this as a basis to reason about algorithmic solutions to policy analysis. This is an improvement on existing work in the field which either ignored the temporal properties of policy managed systems or ignored the need for obligation policies to describe realistic policy-based security mechanisms. Our formalization is based on a characterization of system behavior in terms of execution traces that define the inputs, states and outputs of the system at each logical point in time. We have also defined a formal language to specify authorization and obligation policies using first-order logic formulae containing explicit time arguments and defined the semantics of this language in terms of the system execution traces. Subject to the restrictions given in Section III, policies expressed in our formalism can be evaluated in linear time with respect to the size of the policy. Additionally, by choosing first-order logic with explicit time arguments (rather than temporal logic operators) we can apply abductive reasoning tools to perform a range of analysis tasks.

The main focus of our future work is to extend our formalism to deal with policy refinement. Our initial work on this suggests that the execution trace formalization can be used to characterize refinement in the following manner:

Definition 13: For a domain description D, a policy P_R is a refinement of a policy P in D iff for every $Sys \in D$ that $Sys \models P_R$, then $Sys \models P$, i.e. $mod(P_R) \subseteq mod(P)$.

In addition to working on policy refinement, we plan to extend our language to allow constraints on arbitrary variables in the antecedent of policy rules; allow mixed quantifiers in the rules (as required in Example 12 above); support a choice between policy rules (e.g., allow action 'authorize' or 'command' but not both); support aggregation of variables (e.g., allow "if more than 3 members of mission are present"); and, constraints over domains different than time.

References

[1] G. Rusello, C. Dong, and N. Dulay, "Authorisation and conflict resolution for hierarchical domains," in *Proc. of Int. Workshop on Policies for Distributed Systems and Networks*, June 2007.

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

[3] A. K. Bandara, "A formal approach to analysis and refinement of policies," Ph.D. dissertation, Imperial College London, UK, July 2005.

[4] J. Y. Halpern and V. Weissman, "Using first-order logic to reason about policies," in *Proc. of 16th IEEE Computer Security Foundations Workshop*, 2003, p. 187.

[5] S. Jajodia, P. Samarati, V. Subrahmanian, and E. Bertino, "A unified framework for enforcing multiple access control policies," in *Proc. of the ACM Int. SIGMOD Conf. on Management of Data*, May 1997.

[6] D. Toman and D. Niwinski, "First-order queries over temporal databases inexpressible in temporal logic," in *Proc. of the 5th Int. Conf. on Extending Database Technology (EDBT)*, vol. 1057, 1996, pp. 307–324.

[7] D. Roberts, G. Lock, and D. Verma, "Holistan a futuristic coalition scenario for international coalition operations," in *Proc. of Knowledge Systems for Coalition Operations*, May 2007.

[8] B. Van Nuffelen, "Abductive constraint logic programming: implementation and applications," Ph.D. dissertation, K.U.Leuven, Belgium, 2004.

[9] A. Russo, R. Miller, and J. Kramer, "An abductive approach for analysing event-based requirements specifications," in *Proc. of 18th Int. Conf. on Logic Programming*, vol. 2401, Aug 2002, pp. 22–37.

[10] A. Kakas and P. Mancarella, "Generalised stable models: A semantics for abduction," in *Proc. of European Conf. in AI*, 1990, pp. 385–391.

[11] K. Irwin, T. Yu, and W. H. Winsborough, "On the modeling and analysis of obligations," in *Proc. of the 13th ACM Conf. on Computer and communications security*, 2006, pp. 134–143.

[12] S. Barker, "Security policy specification in logic," in *Proc. of Int. Conf. on AI*, June 2000, pp. 143–148.

[13] S. Jajodia, P. Samarati, and V. Subrahmanian, "A logical language for expressing authorizations," in *Proc. of the IEEE Symposium on Security and Privacy*, 1997, p. 31.