Towards Practical Privacy Policy Enforcement

William H. Winsborough, Jeffery von Ronne, Omar Chowdhury,
Jianwei Niu, Md. Shamim Ashik
wwinsborough@acm.org, {vonronne, ochowdhu, niu, sashik}@cs.utsa.edu

Technical Report CS-TR-2011-009

Department of Computer Science
The University of Texas at San Antonio
San Antonio, TX 78249

ABSTRACT

Organizations that use private information typically must provide assurances to regulators that their practices ensure that regulations are met. However, to the extent that they rely on electronic information systems for the management of private information, they really have no basis for providing those assurances. This paper proposes a framework for the design and implementation of information systems that provably enforce privacy policies. The privacy policies we aim to enforce are expressed in first-order temporal logic (FOTL). They capture not only safety, but also liveness requirements, which are essential in privacy policy. For a variety of reasons, prior work in runtime monitoring is of limited use in privacy policy enforcement. Among these reasons are the need to support liveness requirements, a desire to ensure through static verification that runtime policy violations do not occur, and above all, a recognition that users of electronic information systems require meaningful explanations when actions they attempt to initiate are denied. The latter is particularly relevant in the context of privacy policy because the (human) subject of information often needs to consent to having their personal information shared. So when a denial occurs, it may be that the user needs to seek permission from the subject to share his/her information. For all these reasons, our approach requires us to draw on and solve problems in diverse areas of computer science. We inventory open problems that must be solved, several of which we solve here.

Keywords: Privacy policy and enforcement, first-order temporal logic, safety and liveness properties, static program analysis

1. INTRODUCTION

Our society is becoming increasingly dependent on computer information systems for the proper management of private data. Medical records, financial data, and personal information collected from Internet users are just a few examples. Organizations are required to keep and share such information in a manner that conforms to specific privacy policies, which are mandated by custom, sound business practice, good citizenship, contract, and, often, by law. Examples of privacy policies that carry the force of law include those resulting from the Health Insurance Portability and Accountability Act (HIPAA) [1], the Gramm-Leach-Bliley Act (GLBA) [3], and the Children's Online Privacy Protection Act (COPPA) [2]. In addition to legal regulations, organizations typically have their own business rules that add further privacy requirements.

Organizations that use private information typically must provide assurances to regulators that their practices ensure regulations are met. However, to the extent that they rely on electronic information systems for the management of private information, they have little or no basis for providing those assurances, other than their trust in system developers having written software that enforces the regulations. The complexity of the various aspects of the problem—the regulations and other privacy requirements, the specification of privacy policy, and the systems that manage information while enforcing privacy policy—make it very difficult to develop systems that meet critical privacy objectives. System designers and developers need to have means by which they can check their work.

Several frameworks have been proposed for specifying and analyzing privacy policies, including the Enterprise Privacy Authorization Language (EPAL) [7], Contextual Integrity (CI) [8], Privacy APIs [25], Ponder [13], and work by Breaux and Anton [11]. To date, most of the work in this area has concentrated on frameworks for expressing privacy policies [7, 8, 13], answering queries requesting rulings as to whether transmitting a certain piece of information is permitted [7, 13], methodologies for converting regulations expressed in legal language into formal system requirements [11], or analyzing privacy policy to determine whether it satisfies various properties [25]. Lacking have been enforcement frameworks comprising techniques and tools that support the development and verification of information systems that adhere to such policies.

There has been a great deal of work in enforcement of security policies, particularly in the area of runtime monitors [30, 23, 17]. As we will see, such techniques from the literature are inadequate for enforcing privacy policy. Specifically, monitoring seeks to separate concerns for basic functionality from security concerns. Shortly below, we will illustrate why such separation does not facilitate, but rather hinders privacy policy enforcement. But first, some context.

The information systems that record and transmit our private data are reactive systems [15]. Their behavior can be characterized in terms of sets of infinite sequence of events and/or states. Such a sequence is called a *trace*. Sets of traces are called *temporal properties* [12]. Every temporal property is given by the intersection of two properties, one of which is called a safety property, the other a

liveness property [5, 6]. *Safety*[1] properties say that some bad thing never happens and *liveness* properties say that some good thing eventually happens. Safety properties are amenable to enforcement through classical runtime monitoring techniques [30], while liveness properties are not.[2] Privacy policies often state that certain events cause obligations to be incurred. This is a form of liveness called a response property. For example, HIPPA requires a clinic to release (most of) a patient's medical records when the patient requests them. Once the patient is authenticated, software components can often fulfill the obligation. A court subpoena causes the clinic to incur a similar obligation, though a lawyer may have to be consulted to vet the subpoena. Assuming the lawyer acts responsibly, fulfilling his vetting obligation, we want a guarantee that the system will provide the patient's records to the court. These examples illustrate the fact that sometimes the software system can fulfill liveness requirements on its own, but it is also often the case that such fulfillment depends on the diligence of human agents.

We use a policy language consisting of a subset of first-order linear temporal logic (FOTL) and focus on message transmission as the event that must be regulated, an approach stimulated by Barth *et al.* [8, 9].[3] FOTL extends first-order predicate logic by adding temporal operators that enable formulas to express requirements on past and future events and states.

As mentioned above, runtime monitoring addresses a problem that resembles ours in some respects. The classic approach [30] enforces safety properties, but is unworkable in reactive systems; when a software system attempts to perform an illegal action, execution is terminated. More recent work gets around this problem by introducing edit automata [10, 24], which enable execution to proceed by either suppressing the illegal action, or performing other actions prior to the requested one so as to make the desired action permissible. However, all monitoring-based approaches share a common characteristic: they separate security concerns from basic functionality. This is very suitable for many applications, such as adding security features to legacy code. However, separating policy enforcement from basic functionality is counter productive in privacy policy enforcement for information systems.

The user of an information system plays a central role in the system's operation. When a user attempts to transmit information that would cause a privacy violation, providing an explanation is a central part of the system's functionality. For instance, in the context of electronic medical records, an administrative assistant may request the system to transmit portions of a patient's record for which prior consent by the patient is required. If consent has not been obtained, the administrative assistant needs this to be explained, enabling him to request consent from the patient. Such an explanation is best managed as a part of the system's basic functionality, rather than as a separate wrapper. Thus, a new approach is needed—one that gives the application programmer constructs needed to integrate privacy management along with basic functionality, while still providing rigorous verification that policy is correctly enforced. In particular, we require a system development environment that provides static guarantees that when the system is running, it will enforce the policy, thus ensuring there is no need for a separate component, such as a monitor, to intervene by modifying the system's behavior on the fly. We call this program development methodology *history-aware programming* (HAP).

Broadly speaking, the goal of this paper is threefold: (1) to lay out a vision for providing a framework for supporting HAP; (2) to identify several open problems raised by it; (3) to solve some of the corner stone problems on which our on-going work in this area builds. The problems we solve here are in the following areas: defining forms of first-order linear temporal logic (FOTL) formulas that are suitable for expressing enforceable privacy policies and system component specifications, respectively; formally specified requirements that a given policy of this form must meet to ensure it can be enforced efficiently; requirements for checking that component specifications are properly refined; programming language feature that support what we call history-aware programming; foundations of a type system and an static program analysis that ensures privacy-policy violations will not occur when the system is running; runtime support for maintaining history and obligations. Remaining open problems are in these areas: a comprehensive treatment of human responsibilities in meeting organizational requirements; a methodology for system decomposition and developing suitable component specifications; sound semi-decision procedures for verifying that efficient-enforceability requirements are met by a given policy and for verifying that specification are properly refined. In the course of the body of the paper we will specify these problems in detail and with precision.

To support HAP, we provide the programmer **a new history-query language construct** that can be used to pose queries, expressed as closed FOTL formulas, against the history of system events. The query is part of what we call a *query-conditional* statement. It selects between executing a then-block or an else-block based on the query result. For example, a query-conditional can be used to determine whether a patient has given consent to release his/her medical records to a certain individual. If the query yields "true," the program transmits the information; otherwise, it explain to the user that consent needs to be obtained. (Another query-conditional could be used to inform the user that consent has been refused.) Far from being a distraction to the programmer, the need for such explanations are central to system functionality.

The runtime support we propose for our conditional-query construct is based on work in dynamic model checking [21]. This approach tracks historical events by using a constraint system to record predicate argument values for which predefined FOTL formulas hold. (See section 2 for an example.) In particular, we use this approach to monitor the FOTL formulas that occur in the conditional-query construct within the system code.

To verify statically that send operations are guarded by necessary history queries, we propose a **static program analysis** that transfers information gathered from program semantics, including upon entering the then-branch of a query-conditional to (and across) send operations within that code block. Because query-conditionals guard entire code blocks, not just a single send, the program can avoid performing program state changes in preparation for an illegal send action. Thus, in a well crafted program, there is no need to roll back or otherwise compensate for such state changes. We will see in section 7 how we use static analysis to ensure that liveness properties are enforced (*i.e.*, obligations are fulfilled), provided that any humans that need to participating are diligent in doing so.

[1] The term "safety" is used here with the standard meaning it has in the software verification community [22]. In the security community, it often has another, quite distinct usage in which it refers to the problem of determining whether administrative actions within an authorization system can cause a certain authorization state or class of states to be reached [16].

[2] Some liveness properties can be enforced by edit automata [24]. However, in addition to the hindrance introduced by separation of concerns, the liveness requirements supported by edit automata are too restrictive for our purposes.

[3] Propositional linear temporal logic (PLTL), well known for its use in model checking, is inadequate for our needs because we want to model systems of unbounded size—an unbounded number of human actors, data objects, messages, *etc.* Moreover, quantification is needed for concise policy expression.

As just discussed, we use static program analysis to ensure statically that when the system is running, it enforces the policy. We use static *policy* analysis to ensure that the privacy policy being enforced can be enforced incrementally by using the enforcement mechanisms we employ. In full FOTL, it is possible to write policies that are impossible to enforce incrementally. Even if we bound the number of individual values over which variables range, the problem is intractable in the general case (arbitrary FOTL formulas). However, it is possible to write policy formulas in FOTL that can be enforced incrementally by examining only the system event history, and by handling obligations separately by using methods we discussed shortly above.

Intuitively, when an action is initiated, a poorly formulated policy makes it difficult to detect that performing the action would prevent the system from being permitted to meet its obligations. For instance, consider the following **promising doctor example**. Suppose a policy states the following requirements. When a doctor sends a message to an individual promising that a patient's medical records will be transmitted to the individual, the clinic incurs an obligation to transmit the records. However, the clinic is authorized to release the records only to certain individuals, such as the patient herself. Clearly the system needs to block the doctor's request to send the promise to anyone not authorized to receive the records. The problem is that when the doctor makes his promise, determining whether there is a possible (infinite) system-event future that satisfies the policy is an intractable problem in the general case, even when variables range over small sets of values. A well formulated version of our policy would explicitly state that the doctor is not authorized to promise to send medical records to any individual not authorized to receive them. Such a policy is satisfied by exactly the same traces, but can be enforced reasonably efficiently.

Loosely stated, **the goal of our *policy* analysis** is to ensure that policies are well formulated in the following sense: if no conflict is evident between the system history and the requirements on the history that are explicitly stated, then there is a possible (infinite) system-event future that satisfies the policy. Barth *et al.* [8] provide initial results in this area. They introduce a notion, which they call *weak compliance*, that can be applied to any PLTL formula to capture the intuitive idea that given a finite history, the last state is consistent with the rest of the history and the history requirements that are explicitly stated in the policy. Weak compliance is defined in terms of an automaton-like structure, called a tableau, that can be constructed from a PLTL formula. This has the drawback that the size of the tableau is exponential in size of the PLTL formula. Our approach places syntactic restrictions, defined in section 4, on the structure of our **policy formulas** that make it trivial to syntactically extract from the policy formula an FOTL formula that expresses the same notion as weak compliance for policy formulas. Defined formally below, if a history satisfies this extracted formula, we say the history is **syntactically weakly compliant** (SWC) with the policy.

Barth goes on to call a history *strongly compliant* if there a possible (infinite) system-event future that satisfies the policy. Intuitively, this notion is inadequate for our purposes because it assumes that all agents, whether part of the system or part of the system's environment, cooperate in making the future event sequence actually occur. This neglects the fact that agents in the environment may not cooperate in realizing the possible future event sequence.

In the current paper, we show how, for any policy formula, to use the first order variant of a recently introduced temporal logic $\mathrm{mp^{-}ATL^{*}}$ [26] (which we call $\mathrm{FOmp^{-}ATL^{*}}$) to state the following. Given any finite system-event history, *no matter what future events are initiated by the environment,* the agents in the system are able to ensure that the history is extended to an infinite trace that satisfies the privacy policy. When we make this intuition precise in section 5, satisfaction of the resulting formula will be called **adversarial strong compliance** (ASC). We then go on to present a formula of $\mathrm{FOmp^{-}ATL^{*}}$ that precisely expresses the following idea. At each step in the system execution, provided SWC has been satisfied at every preceding step, ASC will also hold at that step. When this formula is tautological, we know that as long as we enforce SWC, provided obligations are also fulfilled, the system will comply with the privacy policy. The fact that $\mathrm{FOmp^{-}ATL^{*}}$ is strictly more expressive than FOTL means that the problem of determining whether a general $\mathrm{FOmp^{-}ATL^{*}}$ formula is a tautology is undecidable. (The satisfiable formulas of FOTL are not recursively enumerable [19].) Thus in future work we aim to develop sound semi-decision procedures for the syntactic class of $\mathrm{FOmp^{-}ATL^{*}}$ formulas involved.

Throughout the course of this introduction we have described the problem areas to which this paper contributes. Before we can make our technical contributions more precise, some background must be reviewed.

Road map. Section 2 presents background. Section 3 presents our requirements and approach for enforcing privacy policies. It further details our contributions and some open problems. Section 4 and section 5 present our contributions in the area of policy specification, analysis, and monitoring. Section 6 presents our approach to verifying that together the component specifications ensure that the system as a whole enforces the privacy policy. Section 7 presents our contributions in the areas of program structure, programming languages features, and static program analysis, including verification that software components satisfy their specifications. Section 8 concludes.

2. BACKGROUND

This section summarizes prior work that we use or build on, including temporal logic, CI privacy policies, efficient runtime support for queries against the system history, the Actor model of concurrent computation, and the runtime-principals language constructs.

2.1 Temporal Logic (TL)

TL [27] is concerned with characterizing the behavior of reactive systems in terms of states and/or events. Our privacy policy language is a many sorted, first-order linear temporal logic (FOTL) [14], though in the analysis of policies and formalization of policy compliance we make use of another temporal logic [26]. The latter will be summarized in a later section. Due to space constraints we summarize FOTL here briefly.

Linear temporal logics characterize reactive computations in terms of infinite sequences states called *traces*, which we denote by σ. FOTL generalizes propositional linear temporal logic in the same way that first-order logic generalizes propositional logic, namely, by replacing propositional variables by predicate symbols and by introducing quantification and variables. In the formulation we use, trace elements are states and events are embedded in states. We further discuss the structure of states presently.

Sorts resemble a primitive type system; each variable occurring in a formula is assigned a sort when it is quantified and ranges over values in a unique carrier associated with that sort. Predicate argument positions also have associated sorts with which actual arguments must agree. This association is called the *signature* of the predicate symbol. An FOTL *language* is given by a set of variables, a set of sorts, and a set of predicate symbols over those sorts. In our formulation, events, which induce state transitions, are represented by atomic formulas that hold in the destination state. Our

policy language uses only one event predicate, send, which takes a sending agent, a receiving agent, and a message: $\mathrm{send}(p_1, p_2, m)$.

FOTL formulas include *non-temporal formulas*, which are constructed from atomic formulas, possibly with variables, logical connectives, and quantifiers over variables, just as in standard many-sorted, first-order logic. As in the latter logic, a variable is *free* if it is not within the scope of any quantification of that variable. A formula is *closed* if it contains no free variables.

FOTL formulas can also contain temporal operators, each of which takes either one or two FOTL subformulas as arguments, depending on the operator. The temporal operators we use are standard and have the following intuitive meanings[4]. *Future Operators.* Henceforth: $\Box\phi$ says that ϕ holds in all future states. Eventually: $\Diamond\phi$ says that ϕ holds in some future state. *Past Operators.* Historically: $\boxminus\phi$ says that ϕ held in all previous states. Once: $\diamondminus\phi$ says that ϕ held in some previous state. Since: $\phi_1 \mathcal{S} \phi_2$ says that ϕ_2 held at some point in the past, and since then ϕ_1 has held in every state; it is sufficient to consider the most recent point at which ϕ_2 held.

A *state* is an interpretation, meaning that each state s is a mapping of predicate symbols p to relations. Each position in each tuple in $s(p)$ is occupied by a value from the appropriate carrier, based on the signature of p. We assume that $s(p)$ is finite, which is natural because the systems we wish to model are of finite, though possibly unbounded, size; it is simplifying because it means that the number of states is countable.

A *logical environment* η maps each variable to a value in the carrier that corresponds to the variable's sort. That a formula ϕ is satisfied by σ at an index i under η is denoted by $\sigma, i, \eta \vDash \phi$, and can be defined inductively on the structure of ϕ.[5] One says that σ satisfies ϕ, written $\sigma \vDash \phi$, if and only if for all logical environments η, we have $\sigma, 0, \eta \vDash \phi$.

2.2 Contextual Integrity (CI)

The structure of a CI privacy policy is very similar to that depicted in figure 1. (In fact, the form as depicted is the variant that we use. Space constraints prevent our discussing all the differences.) The sorts are P, T, M, and R (denoting agents, attributes, messages, and roles) and their associated carriers are given by $\mathcal{P}$, $\mathcal{T}$, $\mathcal{M}$, and $\mathcal{R}$. The variables p_1, p_2, and q are of sort P, $\hat{r}, \hat{r}_1$, and $\hat{r}_2$ are constants of sort R, t is a variable of sort T, and m is a variable of sort M. The meta-variables $\vec{y}_1$, $\vec{y}_2$, and $\vec{y}_3$ do not occur in CI. We use them to represent vectors of zero or more additional arguments, drawn from p_1, p_2, and q, and used to support parameterized roles. "Procedure" and "diagnosis" are examples of attributes. The meta-variable $\hat{t}$ stands for an attribute set-valued constant. (In CI $\hat{t}$ is actually an individual-valued constant; they make use of an attribute hierarchy.) The formula meta-variables in the norms correspond to syntactic categories introduced below in section 4. The syntactic restrictions thus expressed are part of our policy language; CI also has some syntactic restrictions, though generally speaking, they are not as constraining.

A communication action is denoted by $\mathrm{send}(p, q, m)$, in which p is the sender, q is the receiver and m is the message being sent. Each message contains a set of agent, attribute pairs, $content(m) \subseteq \mathcal{P}\times\mathcal{T}$. A *knowledge state* κ is a subset of $\mathcal{P}\times\mathcal{P}\times\mathcal{T}$. If $(p, q, t) \in \kappa$, this means p knows the value of attribute t of agent q. For example, Alice knows Bob's height. A transition between knowledge states occurs when a message is transmitted from a sender to a recipient, and the attributes contained in the message become known to the recipient.

[4]Our policy language does not use $\bigcirc$ (next) or $\ominus$ (previous) for technical reasons that will be discussed in a later section.

[5]We use "rigid" quantification.

The transition caused by the send event $\mathrm{send}(p, q, m)$ is denoted by $\kappa \xrightarrow{(p,q,m)} \kappa[q \mapsto \mathrm{cl}_{\preceq}(\kappa(q) \cup content(m))]$. One can view the knowledge state as a function that takes an agent q and returns that agent's knowledge state, $\kappa(q) = \{(q', t) \mid (q, q', t) \in \kappa\}$. The function $\mathrm{cl}_{\preceq}$ takes a knowledge state and returns the set of agent, attribute pairs computable from it. Principals are able to send only information that they possess, according to κ.

All the negative norms and at least one positive norm must be satisfied for transmission of message m to be permissible. The meaning of the predicates are as one would expect; $\mathrm{contains}(m, q, t)$ holds if message m contains attribute t of subject q.

Barth *et al.* present representative norms extracted from regulations introduced by HIPAA, COPPA, and GLBA. We discuss some of these that are drawn from HIPAA and COPPA. Norms from the HIPAA privacy rule include those given in formulas (4) and (5) of figure 2. Positive norm (4) gives permission for individuals to receive information about themselves. As with all positive norms, this permission is contingent upon all negative norms being satisfied. Negative norm (5) says that an individual may receive psychotherapy notes about themselves only if permission has been given by the psychiatrist. The negative norm shown in figure 3 is from COPPA and says that if a child sends a web site protected information, then if the web site later receives a request from the parent of that child, the web site must transmit the messages received from the child, as well as the web site's privacy policy. This norm does not conform to our syntactic form for policy formulas. However, it is easily rewritten to do so, as shown in figure 4.

2.3 Past-only FOTL Query Mechanism

Krukow *et al.* [21] present a dynamic programming approach to determine whether at each stage in a trace, which is being generated by a running system, a past-only FOTL formula is satisfied. This is the mechanism we plan to use to answer queries against the history. They call this *dynamic model checking*. They support first order formulas with quantifiers. The technique involves inductively calculating, at each state in the trace, a representation of environments under which each subformula is satisfied. The representation they use is a system of constraints on logical environments. By using this mechanism for each formula appearing in a query conditional, these past-only queries can be answered. The cost of maintaining the mechanism with each step depends linearly on the size of the formula being monitored and polynomially on the number of carrier elements that have been seen in the trace being monitored. The exponent of the polynomial is given by the product of the arity of action predicates and the maximum number of free variables occurring in a subformula of the formula being monitored.

Krukow *et al.* [21] make use of and extend work by Ruşu and Havelund [29] that uses a dynamic programming approach to determine whether, at each stage in a trace that is being generated by a running system, an FOTL formula that uses past temporal operators only is satisfied. They call this *dynamic model checking*. One of their extensions is support for first order formulas with quantifiers. The insight that is the basis of the dynamic programming approach is that the satisfaction of a past-only formula at a given index in the trace can be inductively computed quite efficiently from the current state, the satisfaction at the given index of each of its proper subformulas, and the satisfaction at the previous index of each of its subformulas. For example, if in the previous step $p(x)$ had held in all prior steps for $x = c$, and in the current step, $p(x)$ also holds for $x = c$, the monitor can conclude that in the current step, $p(x)$ has held in all prior steps for $x = c$. More formally, we have $(\ominus\boxminus p(x) \wedge p(x)); \{x = c\} \equiv \boxminus p(x); \{x = c\}$. For the previous stage and the current stage they allocate an array with one

$$\Theta : \Box\forall p_1, p_2, q : P.\forall m : M.\forall t : T.\text{send}(p_1, p_2, m) \wedge \text{contains}(m, q, t) \rightarrow \bigvee_{\varphi^+ \in \text{norms}^+} \varphi^+ \wedge \bigwedge_{\varphi^- \in \text{norms}^-} \varphi^- \quad (1)$$

$$\text{positive norm} : \text{inrole}(p_1, \vec{y_1}, \hat{r_1}) \wedge \text{inrole}(p_2, \vec{y_2}, \hat{r_2}) \wedge \text{inrole}(q, \vec{y_3}, \hat{r}) \wedge (t \in \hat{t}) \wedge \psi \wedge \beta \quad (2)$$

$$\text{negative norm} : \text{inrole}(p_1, \vec{y_1}, \hat{r_1}) \wedge \text{inrole}(p_2, \vec{y_2}, \hat{r_2}) \wedge \text{inrole}(q, \vec{y_3}, \hat{r}) \wedge (t \in \hat{t}) \wedge \psi \rightarrow \chi \quad (3)$$

Figure 1: General form of knowledge-transmission policy

$$\text{inrole}(p_1, \textit{covered-entity}) \wedge \text{inrole}(p_2, \textit{individual}) \wedge (q = p_2) \wedge (t \in \textit{phi}) \quad (4)$$

$$\text{inrole}(p_1, \textit{covered-entity}) \wedge \text{inrole}(p_2, \textit{individual}) \wedge (q = p_2) \wedge (t \in \textit{psychotherapy-notes}) \rightarrow \Diamond\exists p : \mathcal{P}.\text{inrole}(p, p_2, \textit{psychiatrist}) \wedge \text{send}(p, p_1, \langle p_2, \textit{release-psych-notes}\rangle) \quad (5)$$

Figure 2: Some norms based on HIPAA regulations

$$\text{inrole}(p_1, \textit{child}) \wedge \text{inrole}(p_2, \textit{web-site}) \wedge q = p_1 \wedge t \in \textit{protected-information} \rightarrow \Box\forall p : P.\text{inrole}(p, p_1, \textit{parent}) \wedge \text{send}(p, p_2, \langle q, \textit{request-information}\rangle) \rightarrow \Diamond\text{send}(p_2, p, \textit{privacy-notice}) \wedge \Diamond\text{send}(p_2, p, m) \quad (6)$$

Figure 3: A norm based on COPPA regulations

$$\text{inrole}(p_1, q, \textit{parent}) \wedge \text{inrole}(p_2, \textit{web-site}) \wedge \text{inrole}(q, \textit{child}) \wedge t \in \textit{request-information} \rightarrow \forall m' : M.(\Diamond(\text{send}(q, p_2, m') \wedge \text{contains}(m', q, \textit{protected-information})) \rightarrow \Diamond\text{send}(p_2, p_1, \textit{privacy-notice}) \wedge \Diamond\text{send}(p_2, p_1, m')) \quad (7)$$

Figure 4: The rewritten COPAA regulations based norm of figure 3

bit for each node in the abstract syntax tree of the formula, representing whether or not the subformula is satisfied. The extension replaces each bit by a constraint that identifies logical environments for which the formula, which now can have free variables in it, is satisfied. The approach provides us a lot of what we need for handling safety properties. However, it shares with security automata the drawback of providing no direct assistance with liveness properties, such as the response properties that are central to privacy policies.

2.4 Actors

The Actor Model is a theoretical framework for describing concurrent computations that has been also used as the basis for the design of (the concurrency support of) several programming languages, including industrial strength languages such as Erlang, Scala.

In the actor model [18, 4], a software system is considered to be a collection of concurrently operating actors that communicate through asynchronous message passing. Each actor has a mailbox through which it receives messages. Based on its state/behavior, an actor reacts to the messages it receives one at a time—but not necessarily in the order they arrive—by performing some action. The action an actor takes in response to the receipt of a message can involve sending a finite number of messages to other actors it knows about, creating a finite number of new actors, and/or changing its state/behavior so that it will take different actions in response to future messages.

The actor model itself does not specify how the actor's behavior is defined. In this paper, we will be defining a language, which like Scala, is class-based and imperative. Thus, each actor's behavior is defined by an actor class which consists of action-methods. Each action-method contains imperative code that will execute when the actor receives a certain type of message.

Agha *et al.* [4] and Talcott [31] give semantics for actor components in terms of the evolution of configurations of actors, which we will make use of as a basis for specification refinement. A *configuration of actors* is a group of actors with an interface for interaction with their environment. An interface consists of a set of *receptionists,* that specifies which actors of the system are visible to the environment, and a set of *externals* that specifies the names of actors in the environment known to actors in the system. The literature defines how high-level actor components can be created as compositions of low-level components with the lowest-level of components being defined directly in terms of actors. One of the important contributions of this actor-based conception of components is that it treats the creation and initialization of new actors as first class operations, and gives full treatment to the evolution of component interfaces over time as new receptionists are exposed to the environment.

2.5 Runtime Principals

In their work on Runtime Principals, Tse and Zdancewic [32] introduce a language whose type system enforces information flow policies for confidentiality and integrity that depend on the principals that dynamically interact with the system. This is accomplished through the use of variables that contain runtime principals. Runtime principals are judged to have singleton types that are constructed using principals or type-level principal variables that statically represent principals. These same principals and type-level principal variables can also be used to construct labels that are used to construct types for data variables. These labels express who is allowed to read or write certain variables. We will make use of a variation of this construct in our language, HAP.

Their language also allows type-level principal variables to be bound by an 'acts for' relation, which is used to express delegation. The principal Alice acts for Bob means that Bob has delegated his authority to Alice, so that Alice can read or write everything that Bob can read or write. The type rules maintain an environment representing a static approximation of the 'acts for' relation that is used in the type rules to reason about what operations are allowed. Programs are allowed to condition code on a dynamically query of whether the 'acts for' relationship holds between two runtime principals at a particular program point, and in the environment of the body of that query. Thus, the type system might discover, that within the body of a query, that a type-level principal variable α acts for Alice, and it could then allow α to read variables that it knows Alice can read.

The type signatures of functions can be made polymorphic with respect to (has type parameters that take) these type-level principal variables. This allows the creation that can be applied to variables containing different runtime principals by specializing the function for the appropriate runtime principals. These type parameters can be bound so that the function can only be instantiated for type-level principal variables that 'act for' some principal. Thus, one

can write a function that takes a runtime principal α constrained so that α 'act for' Alice, takes some data that Alice can read, and returns the same data typed so that it can be read by α.

The types of data structures can make use type-level principal variables that are existentially quantified. This allows one to create a list of pairs where each pair contains the runtime principal represented by α and some data readable by that same α without the identity of α being known by the type system. Note that, this allows an unbounded number of principal values to be contained in a value whose type only has a single (existentially quantified) type-level principal variable.

3. REQUIREMENTS, APPROACH, CONTRIBUTIONS, AND OPEN PROBLEMS

We shall provide several techniques to ensure incremental enforcement is correct and efficient.

First, in section 4, we restrict the syntactic form of policies so that the future obligations (liveness properties) are easy to recognize, reason about, and to elide (see section 5) from the policy. Once obligations are removed, the resulting formula states a requirement that is "past only" in a sense that will be made precise later. Essentially this means that the resulting formulas express requirements only on legal histories, and are easily converted into queries of the kind our conditional-query construct supports.

We also disallow use of the temporal operator $\ominus$. The formula $\ominus\phi$ says that ϕ held in the previous step. This expressive power is not needed in privacy policy—the other past operators capture the requirements that are needed. Moreover, the operator makes incremental enforcement unnecessarily difficult: a requirement that my second to last action was not a certain send needs to be enforced before I actually perform the send.

Second, as was illustrated in the introduction by the promising doctor example, we need to formalize the property of policy formulas that SWC entails ASC. We do this in section 5.

Third, in on-going work, we are developing reasonably efficient sound semi-decision procedures for determining that SWC entails ASC holds. Among the approaches we are exploring to this problem we are investigating the use of *small model theorems* [28, 33]. The idea here is to show that for a certain class of formulas, if the formula is satisfiable, then it is satisfiable over a carrier bounded by a certain size. Such theorems justify taking the approach suggested (without justification) by Barth *et al.* [8] of converting a first-order formula over carriers of small size into a propositional formula. Such a step is essential to enable us to apply any kind of (static) model checking technique. In particular, it enables us to soundly convert the problem of satisfiability of $\mathrm{FOmp}^{-}\mathrm{ATL}^{*}$ formulas to satisfiability of $\mathrm{mp}^{-}\mathrm{ATL}^{*}$ formulas. This conversion is exponential in the number of free variables occurring in any given subformula, with the base of the exponential being given by the size of the carriers. It seems very likely that by placing quantifiers in a manner that makes their scope as small as possible, the number of free variables occurring in any given formula can be kept acceptably small. (Recall that this analysis is done statically, not a runtime.) However, additional techniques will be needed, as in general satisfiability of $\mathrm{mp}^{-}\mathrm{ATL}^{*}$ formulas is double exponential-time complete. Decomposing policies in a manner akin to program slicing may help to reduce substantially the size of formulas that need to be considered in any given problem instance. Finally, as mentioned in the introduction, it is not essential to be able to recognize every satisfiable (or tautological) formula. So abstractions introduced to simplify the problem further may introduce certain sound forms of approximation.

Fourth, users must participate in fulfilling some obligations. When this is the case, the user will be informed and, as necessary, reminded, and after an appropriate amount of time has elapsed without the user fulfilling his obligation, the user's actor may notify the actor associated with a person in a position of authority.

Fifth, the information system consists of a collection concurrently executing actors that communicate through asynchronous messages. Each actor represents one principal and is an instance of an actor class that defines the actor's behavior. An actor class can only be instantiated for actors belonging of some particular role. The actions of that actor are considered to be actions of the principal it represents. An actor only gains knowledge through the messages it has sent and or received.

Sixth, we hierarchically decompose the policy into specifications of abstract components, which each consist of one or more actors. These specifications collectively enforce the policy. The actors that are consider to make up components can be programs (instances of actor classes) or 'human actors'. Specifications are also associated with actor classes, and responsibilities assigned to humans can be treated as their 'specification.' Each higher-level specification should be shown to be entailed by lower-level specifications.

Seventh, each actor has a history mechanism, which records information about the communications that that actor has participated in. The mechanism considers messages to be received at the time they are dequeued and the appropriate action-method is executed. The information recorded by the history mechanism is used to answer queries embedded in an actor's code about that actor's history. The local nature of the history mechanism is essential to the scalability of the system design. While there may be a very large number of users of the information system, and the system as a whole may communicate with a very large number of external entities, the number of individuals a single principal will communicate with will be much smaller. Furthermore, the number of events that each actor history manager needs to record will grow relatively slowly.

Eight, since knowledge is local, specifications must be written in such a way as to ensure that an actor representing a principal becomes aware of actions that it needs to know to determine the permissibility of its actions. Since policy norms refers only to send operations to which the principal denoted by p_1 in figure 1 is a party (*i.e.*, a communication participant), one strategy to accomplish this to have only a single actor for at least some of the principals.

Ninth, at the lowest level of refinement, specifications are associated with actor classes against which the code for individual 'action-methods' (that executes in response to messages that have been received) is verified. These require that by the end of each action-method, any obligation incurred as a result of the receive event that triggered the method's invocation has been handled. Handling the obligation may mean fulfilling it directly, but it can also mean sending a message to another actor. That other actor may or may not be associated with a user. In the former case, the recipient actor takes some further action toward the obligation's eventual fulfillment. In the latter, the obligation requires a user's participation to be fulfilled.

Tenth, we use type rules and static program analysis to verify that at runtime actor instances will comply with the specifications of the actor class defining its behavior.

4. POLICY SPECIFICATION LANGUAGE

This section specifies defines the syntactic form of our FOTL policies. The syntactic categories we use are shown in figure 5. As is customary, we overload the names of the syntactic categories and also use them as metavariables ranging over formulas in their respective category. Thus we are now able to see some of the restrictions placed on norms depicted in figure 1. Notice in particular

$$
\begin{array}{rl}
\text{(Atomic Formulas) } \gamma ::= & \mathrm{send}(p_1, p_2, m) | \mathrm{inrole}(p, \vec{y}, r) \\
& | \mathrm{contains}(m, q, t) | (t \in t') \\
& | e_1 = e_2 | \mathit{true} \\
\text{(Non-temporal Formulas) } \mu ::= & \gamma | \mu \wedge \mu | \mu \vee \mu | \exists x{:}\tau.\, \mu | \forall x{:}\tau.\, \mu \\
\text{(Pure Past Formulas) } \psi ::= & \mu | \psi \wedge \psi | \neg \psi | \psi \,\mathcal{S}\, \psi | \exists x{:}\tau.\, \psi \\
\text{(Obligation Formulas) } \beta ::= & \Diamond \mu | \beta \wedge \beta \\
\text{(Mixed Formulas) } \chi ::= & \beta | \psi | \psi \wedge \beta | \psi \rightarrow \beta | \forall x{:}\tau.\, \chi
\end{array}
$$

Figure 5: FOTL policy formulas.

the limited way in which future temporal operators are used. Aside from the $\Box$ at the outer-most level, the only future subformulas are of the form given by β and $\Diamond$ can be applied only to non-temporal formulas. This is the key to our being able to define SWC as cleanly as we do in section 5. The fact that the negative norm shown in figure 3 can be expressed in our form, as shown in figure 4, illustrates why we believe this restriction is acceptable. An analogous situation arises when a member of medical staff sends a new entry to the medical record repository: when the patient subsequently asks for her records, the clinic must eventually provide them. Here again, we can reframe the requirement in terms of the time point at which the patient requests her records.

The equality in γ warrants explanation. The meta-variables e_1 and e_2 range over agents and messages. The messages in question can be message variables, constants, or tuples of constants and variables, such as $\langle q, \mathit{release\text{-}psych\text{-}notes} \rangle$.

In any syntactic context, we call messages that contain subject attributes *regulated messages* and those that contain no attributes *speech acts*. The latter are not explicitly discussed by Barth *et al.* [8], although they are used in the HIPPA norm depicted in figure 2. It is through the transmission of speech acts that requests are made, permission is granted, *etc.*

We allow roles to be parameterized, as illustrated in figures 2, 3, and 4. For this we extend CI's inrole predicate to a family of predicates of differing arities (much as is done in paper by Barth *et al.* [9] that builds on CI). We have $\mathrm{inrole}(p, \vec{y}, r)$ if $(p, \vec{y}, r) \in \mathrm{roleAssignment} \subseteq \mathcal{P} \times \mathcal{P}^* \times \mathcal{R}$. For example, if $(\mathit{Alice}, \mathit{Bob}, \mathit{psychiatrist}) \in \mathrm{roleAssignment}$, then Alice is Bob's psychiatrist and if $\mathrm{inrole}(\mathit{Bob}, \mathit{Fred}, \mathit{Carol}, \mathit{child})$ holds, then Bob is the child of Fred and Carol. We make the simplifying assumption that principals are statically assigned to roles.

The FOTL that we allow in the positive norms are of form $\psi \wedge \beta$ whereas Barth *et al.* allow arbitrary FOTL formulas (past and future). In case of negative norms, they do not allow any temporal operators in the antecedent of the norms whereas, we allow pure past temporal formulas of form ψ in the antecedent. However, they allow arbitrary FOTL formulas in the consequent of the negative norms. On the other hand, the FOTL formulas that we allow in the consequent of the negative norms are of form χ.

5. POLICY ANALYSIS

This section begins by showing that the syntactic restrictions we place on policy formulas make incremental enforcement straightforward, under the assumption that SWC implies ASC. The section begins by formalizing what it means for incremental enforcement to be straightforward. This notion is called incremental monitorability. It then shows that policy formulas are incrementally monitorable. It then goes on to introduce $\mathrm{FOmp^{-}ATL^{*}}$, to define ASC, and, given a policy formula, to express in an $\mathrm{FOmp^{-}ATL^{*}}$ formula the property that SWC implies ASC for this formula.

DEFINITION 1 (INCREMENTALLY MONITORABLE). *An FOTL formula ϕ is* incrementally monitorable *if for any given finite trace σ and for any logical environment η, $\sigma, |\sigma| - 1, \eta \vDash \boxminus \phi$ implies there exists an infinite trace $\hat{\sigma}$ such that $\sigma \cdot \hat{\sigma} \vDash \Box \phi$.*

$\mathit{weak}(\Theta)$ denotes the formula derived from Θ by replacing each subformula of the form $\Diamond \mu$ with true and removing the outermost $\Box$ (see figure 1). The formula obtained expresses the safety component of Θ. We can define SWC using $\mathit{weak}(\Theta)$ as follows.

DEFINITION 2. *Assume a fixed privacy policy Θ. Given a finite history σ and a state s, s is* syntactically weakly compliant (SWC) with respect to σ *if for all η_0, we have $\sigma \cdot s, |\sigma|, \eta_0 \vDash \boxminus \mathit{weak}(\Theta)$.*

In the above definition, although σ is finite and we have formally defined $\vDash$ only in terms of infinite traces (or paths), the usage of $\vDash$ here is well defined because, as $\mathit{weak}(\Theta)$ is a pure past formula, $\sigma \cdot s, |\sigma|, \eta_0 \vDash \boxminus \mathit{weak}(\Theta)$ depends only on the states in σ.

THEOREM 3. *Any closed, pure past FOTL formula ψ without the temporal operator $\ominus$ is incrementally monitorable.*

PROOF. The proof uses the intuition that it can be shown by induction on the structure of ψ that the truth value assumed by each subformula of ψ is identical in two adjacent states have identical labels. The basis is trivial. In the step, the logical connectives and quantifier cases are straightforward. Recall that the only temporal operator is "since". The formula $\psi_1 \,\mathcal{S}\, \psi_2$ is true when ψ_2 is true in the current state. Otherwise it is false when ψ_1 is false in the current state. In the remaining case, in both of the two adjacent states under consideration it takes on the same value as it did in the state immediately prior to them. $\square$

As privacy policies of our form (Θ) do not have $\ominus$ temporal operator, we have the following.

COROLLARY 4. *The formula $\mathit{weak}(\Theta)$ is incrementally monitorable.*

The temporal logic FOmp^{-}ATL*. This section shows how we can use (the first-order variant of) a recently introduced temporal logic, namely $\mathrm{mp^{-}ATL^{*}}$ [26], to significantly refine the notion of strong compliance due to Barth *et al.* [8]. Barth's formulation neglected the fact that agents in the environment may be adversaries that do not cooperate in extending the current system history to form an infinite trace that satisfies the policy. By explicitly distinguishing between cooperating agents and potentially non-cooperating agents, we here obtain a stronger notion of compliance that avoids this pitfall. Our formulation requires the cooperating agents to have a strategy that enables them to satisfy the policy, no matter what actions are initiated by non-cooperating agents.

Formulas of $\mathrm{mp^{-}ATL^{*}}$ are interpreted over concurrent game structures (CGS). Due to space constraints, we provide here an intuitive summary and refer the reader to Mogavero *et al.* [26] for a detailed presentation. The logic presented there is propositional; however generalizing it to first-order is straightforward and required for our purposes because we require our policies to be formulas in the language. We call the first order extension of $\mathrm{mp^{-}ATL^{*}}$ logic $\mathrm{FOmp^{-}ATL^{*}}$. A CGS $\mathcal{G}$ resembles a Labeled transition system, except that actions are replaced by global decisions, which intuitively speaking distinguish individual actions according to which agent initiates it. Thus, denoting the set of agents in the system by Ag and individual decisions by AC, a global decision is a member of the function space $\mathrm{Ag} \rightarrow \mathrm{AC}$. Transitions are defined by a function that takes a state and a global decision and returns a new state. A strategy f_A for a set of agents $A \subseteq \mathrm{Ag}$, defines the local decisions that the (cooperating) agents in A make, based on the current system history. $\mathrm{mp^{-}ATL^{*}}$ introduces generalized path quantifier which is also known as strategy quantifier

$$\langle\!\langle \varnothing \rangle\!\rangle \Box[\boxminus weak(\Theta) \longrightarrow \langle\!\langle A \rangle\!\rangle \Theta]$$

Figure 6: $\mathrm{FOmp^{-}ATL^{*}}$ formulation of SWC entails ASC for Θ. We denote this formula by $\textsc{swc} \rightarrow \textsc{asc}(\Theta)$.

(denoted by $\langle\!\langle \rangle\!\rangle$). Intuitively, the formula $\langle\!\langle A \rangle\!\rangle \phi$ says that the agents in A have a strategy f_A such that no matter what local decisions are made by agents not in A, by using f_A, the agents in A can force the current history to be extended to an infinite trace that satisfies ϕ. When all the agents in the system are in A, $\langle\!\langle A \rangle\!\rangle \phi$ says that there exists a path that satisfies ϕ. Similarly, the formula $\langle\!\langle \varnothing \rangle\!\rangle \phi$ says that all possible paths satisfy ϕ.

In order to define adversarial strong compliance in the next section, it is necessary to use the semantics of $\mathrm{mp^{-}ATL^{*}}$, not just a formula of the logic. $\mathrm{mp^{-}ATL^{*}}$ syntax includes state formulas Φ and path formulas Ψ, and path quantifiers and temporal operators can be nested arbitrarily. In $\mathrm{mp^{-}ATL^{*}}$ the semantics of state formulas identifies the current state by the finite sequence of states ρ that lead to the current state. Let us outline the semantics of path-quantified formulas, which are state formulas. The statement $\mathcal{G}, \rho \vDash \langle\!\langle A \rangle\!\rangle \Psi$ holds if and only if there exists a strategy f_A such that for all infinite paths π that result from agents outside of A taking arbitrary decisions and agents in A applying f_A, it holds that $\mathcal{G}, \rho \cdot \pi, 0 \vDash \Psi$. The latter is defined by structural induction much as is done for FOTL formulas with the exception that when a path quantifier is encountered, the subformula is again a state formula. To say that a state formula holds at some index k in $\rho \cdot \pi$ (written, $\mathcal{G}, \rho \cdot \pi, k \vDash \langle\!\langle A \rangle\!\rangle \Psi'$), the prefix of $\rho \cdot \pi$ having length $k+1$ (written, $\rho \cdot \pi_{\leq k}$) takes the place of ρ in the semantics of state formulas given above (written, $\mathcal{G}, \rho \cdot \pi_{\leq k} \vDash \langle\!\langle A \rangle\!\rangle \Psi'$).

Adversarial strong compliance. Given a history ρ' and an action a, Barth's notion says that $\rho = \rho' \cdot a$ is strongly compliant if there exists an infinite π such that $\rho \cdot \pi$ satisfies the policy. We say that ρ is *adversarially strongly compliant* for a given CGS $\mathcal{G}$ and for a given environment η if $\mathcal{G}, \rho, \eta \vDash \langle\!\langle A \rangle\!\rangle \Theta$ holds. Intuitively, this means that the no matter what actions the potentially adversarial environment initiates, the human and software agents that compose the information system can extend the history to an infinite trace that satisfies the privacy policy. In contrast, Barth's notion presumes that the environment cooperates in extending the history.

A $\mathrm{mp^{-}ATL^{*}}$ specification for SWC entails adversarial strong compliance. We denote by $\textsc{swc} \rightarrow \textsc{asc}(\Theta)$ the formula given in figure 6. In essence, the formula says the following for every path starting at the initial state and at each step along that path. If SWC has held at every step up to the current one, then no matter what decisions the other agents make thereafter, the agents in A can force an infinite sequence of global decisions that yields a trace satisfying the privacy policy.

THEOREM 5. *Given a privacy policy Θ, a CGS $\mathcal{G}$, an environment η, a history ρ' and an action a, if $\mathcal{G}, \eta \vDash \textsc{swc} \rightarrow \textsc{asc}(\Theta)$ and $\rho, \eta \vDash \boxminus weak(\Theta)$ where $\rho = \rho' \cdot a$, then $\mathcal{G}, \rho, \eta \vDash \langle\!\langle A \rangle\!\rangle \Theta$*

THEOREM 6. *For a given privacy policy Θ if $\textsc{swc} \rightarrow \textsc{asc}(\Theta)$ holds for all CGS $\mathcal{G}$ and all environment η, then whether an action a is adversarially strongly compliant can be incrementally monitored.*

6. FROM POLICY TO SPECIFICATION

We aim to provide a methodology for not only formally specifying and reasoning about privacy policies in temporal logic but also for developing distributed information systems that can be verified to comply with those policies. This section is concerned with how to refine privacy policies into specifications describing the externally visible behavior of components made up of actors, and how to verify the correctness of such refinements.

6.1 Levels of Abstraction

Our approach to the refinement of specifications is hierarchical and is loosely based on the notion of actor components found in the literature [4, 31]. At each level of abstraction, there will be a specification for each actor component that describes assumptions about the messages its actors may receive from actors in its environment and provides guarantees about the messages its actors may or must send to actors in the environment. Thus, although the methodology of specification refinement is hierarchical, the semantics of the specification as well as the implementation are defined in terms of communicating actors. During top-down development of the specifications, the policy analysis techniques discussed previously in Section 5 can be used to ensure that the specifications being developed during the refinement are feasible.

In order to relate specifications at these different levels of abstraction it is necessary to establish a mapping from entities at one level to entities at the next. For example, legal requirements, such as HIPAA, and organizational privacy policies constrain the behavior and place obligations on people and organizations that qualify as covered entities, so it is necessary to establish a mapping from the actions taken by the individual actors that make up the system to the activities and entities regulated by the privacy policy. To simplify this mapping, we require that all actors within a component always act on behalf of a single principal at the policy level.

Additionally, since the main focus of this paper is elsewhere, we assume that information is correctly labeled with the attributes it contains. In practice, much of this data will have been initially entered into the system by humans and must be manually tagged with the appropriate attributes (cf. Barth et al. [9]).

6.2 Actor specifications

Actor specifications are the foundation on which higher-level specifications (which have a similar structure) are established. Each actor class has an assume-guarantee specification that proscribes how instances of that actor class may interact with its environment (i.e., the messages it may or must send to other actors). In this section, we discuss how these specifications are structured.

Specifications include predicates that do not occur in policies. Specifically, because our programming language treats send and receive as asynchronous events, our specification language must include a receive predicate, $\mathrm{receive}$; and because our specifications will deal with messages sent by actors (of sort A associated with carrier $\mathcal{A}$) that act on behalf of the agents mentioned in the policy, the specification language includes the predicate, $\mathrm{actsOnBehalfOf}$, to associate actors with the agents regulated by the high-level policy. Furthermore, actors are aware of only the actions in which they participate. So we additionally wish to constrain certain actors; sends and receives are initiated only by the local actor, which we denote by the metavariable, $self$. Thus, in the context of actor specifications, the analogue of atomic formulas γ is the syntactic category of *local atomic formulas* γ', defined by

$$\gamma' ::= \mathrm{send}(self, a, m) | \mathrm{contains}(m, q, t) | \mathrm{receive}(self, a, m) | \mathrm{actsOnBehalfOf}(a, p) | \mathrm{inrole}(p, \vec{y}, r) | (t \in t') | true | false$$

The other local versions of the syntactic categories of our *specification FOTL language*, *local non-temporal formulas* μ', *local pure past formulas* ψ', *obligation formulas* β', and *local mixed formulas* χ' are defined in terms of γ' just as are the policy FOTL language shown in Figure 5, except that γ' is used in place of γ.

An *actor specification* has the form

$$\left(\Box \bigwedge_{\psi' \in \text{A-Assum}} \psi'\right) \rightarrow \left(\Box \bigwedge_{\psi' \in \text{A-Safety}} \psi' \wedge \bigwedge_{\chi' \in \text{A-Liveness}} \chi'\right)$$

in which, for the given actor, A-Assum is a set of assumption clauses, A-Safety is the set of safety clauses and A-Liveness is the set of liveness clauses. Assumption clauses $\psi' \in$ A-Assum are restricted to take the form

$$\forall a{:}A.\, \forall m{:}M.\, \text{receive}(\mathit{self}, a, m) \rightarrow \psi'(a, m)$$

safety clauses $\psi' \in$ A-Safety are restricted to take the form

$$\forall a{:}A.\, \forall m{:}M.\, \text{send}(\mathit{self}, a, m) \rightarrow \psi'(a, m)$$

and liveness clauses $\chi' \in$ A-Liveness are restricted to take the form

$$\forall \vec{x}\,.\, \psi'(\vec{x}) \rightarrow \Diamond \exists a{:}A.\, \exists m{:}M.\, \mu'(a, m, \vec{x}) \wedge \text{send}(\mathit{self}, a, m)$$

Here we assume that $\forall \vec{x}$ includes specification of the appropriate sorts.

The requirements on the syntactic forms of assumption and safety clauses (an implication with a receive by *self* or a send by *self*, respectively, as the antecedent) serve to syntactically isolate a specific action, so that the clause as a whole describes under what circumstances that action is permitted.

Thus an assumption clause in a specification for an electronic medical record archive, might say it will only ever receive requests for a patient's records from Actors acting on behalf of that patient, their healthcare providers, or their medical insurance companies. In contrast, a safety clause in a specification for an electronic medical record archive, might say something like, in order for the medical record archive to send a medical record of some patient to any actor that does not act on behalf of the healthcare provider or that patient, it must have previously been the case that someone acting on behalf of the patient has sent authorization to the archive to release records to that individual.

Furthermore, this restriction also serves to ensure that the safety clause is incrementally monitorable and, moreover, that violations of the safety clause can be associated with a specific send action. More technically, if ψ' is a safety clause, $\Box\psi'$ will only fail to be satisfied by an infinite trace σ if and only if there exists a finite prefix trace σ' of σ ending with a send event, and the formula ψ' is not satisfied by σ' at that send event ($\sigma' = \sigma'' \cdot \text{send}(\mathit{self}, a, m)$ and $\sigma', |\sigma'| - 1, [] \not\models \psi'$). Consequently, a safety clause ψ' can be complied with by ensuring that $\text{send}(\mathit{self}, a, m)$ events only occurs when the history σ is such that $\sigma, |\sigma| - 1, [] \models \psi'$.[6]

In liveness clauses, the syntactic form ensures that an actor is obligated to send messages under circumstances defined in terms of events it is a party to (messages it sends or receives). Like assumption and safety clauses, liveness clauses are restricted to take the form of an implication. In contrast, however, the implication of a liveness clause has an antecedent that contains a local pure past formula ($\psi'(\vec{x})$), whereas the consequent is restricted to requiring the actor to send some action in the future as constrained by a local non-temporal formula ($\mu'(a, m, \vec{x})$). In this way, liveness clauses describe conditions which incur an obligation of the actor to some send a message in the future. The universal quantification of $\vec{x}$ allows a single condition in a single clause to incur obligations to send multiple messages. As an example, considering again the specification of a medical record archive, a liveness clause quantified over a patient, doctor, and medical record, might thus state that, if the patient requests their medical records and the doctor previously sent the record to the archive, the archive is obligated to eventually send the patient that medical record. Consequently, if the doctor had sent multiple records to the archive, the archive would be obligated to send all of them to the patient.

[6]We use [] to denote the empty environment.

6.3 Demonstrating Correct Refinement Specification

Component specifications have a similar structures to actor specifications, except that its atomic formula describe communications between actors inside and outside of the component instead of messages to and from some particular actor.

In principal, component specification are shown to be correctly implemented by showing that it is entailed by the composition of specifications of the subcomponents or actors that make up the component. Without loss of generality, we will focus on the subcomponent case and assume that there is only one assumption, guarantee, and liveness clause per component/subcomponent. If we have an assumption A_C for a component, an assumption A_i for each subcomponent i, and a safety guarantee S for the component and S_i for each subcomponent i, and a liveness guarantee L_C for the component and L_i for each subcomponent i, it is sufficient to show the validity of the following refinement formulas:

$$(\forall a_1{:}A.\, \forall a_2{:}A.\, \forall m{:}M.\, \text{receive}(a_2, a_1, m) \rightarrow \Diamond\text{send}(a_1, a_2, m)) \rightarrow (\Box A_C) \rightarrow \bigwedge_i (\Box S_i) \rightarrow \bigwedge_i (\Box A_i)$$

$$(\forall a_1{:}A.\, \forall a_2{:}A.\, \forall m{:}M.\, \text{receive}(a_2, a_1, m) \rightarrow \Diamond\text{send}(a_1, a_2, m)) \rightarrow (\Box A_C) \rightarrow \bigwedge_i (\Box S_i) \rightarrow S_C$$

$$(\forall a_1{:}A.\, \forall a_2{:}A.\, \forall m{:}M.\, \text{send}(a_1, a_2, m) \rightarrow \Diamond\text{receive}(a_2, a_1, m)) \rightarrow (\Box A_C) \rightarrow \bigwedge_i (\Box L_i) \rightarrow L_C$$

The ability to use the validity of these formulas to establish the correctness of each refinement step relies on the restrictions that we place on subcomponent assumptions and safety guarantees. These ensure that they are incrementally monitorable, and furthermore, that only one can be false at any particular position in the trace. Consequently, $(\Box S_i) \rightarrow (\Box A_i)$ is equivalent to $\Box((\ominus\boxminus S_i) \rightarrow (\boxminus A_i))$, and $(\Box A_i) \rightarrow (\Box S_i)$ is equivalent to $\Box((\ominus\boxminus A_i) \rightarrow (\boxminus S_i))$.[7] Additionally, induction on the length of trace prefixes can be used to show

$$\Box((\ominus\boxminus S_i) \rightarrow (\boxminus A_i)) \wedge ((\ominus\boxminus A_i) \rightarrow (\boxminus S_i)) \rightarrow (\boxminus A_i \wedge S_i)$$

(In Jonsson and Yih-Kuen [20], a similarly restriction of assumptions to the form $\ominus\boxminus\psi$, where ψ is a pure past temporal formula, is used in order to ensure induction can be applied to show the guarantees resulting from composition.) Furthermore, since $\boxminus A_i \wedge S_i$ is incrementally monitorable, $\Box\boxminus A_i \wedge S_i$ is equivalent to $(\Box A_i) \wedge (\Box S_i)$.

Standard model checking techniques can be applied to establish the validity of these refinement formulas after employing abstraction or a small model theorem to handle the first-order nature of the formulas.

[7]The symbol $\ominus$ stands for 'weak yesterday'. $\ominus\phi$ is satisfied at the first position of a trace or if $\ominus\phi$ is satisfied.

7. PROGRAMMING LANGUAGE DESCRIPTION AND STATIC ANALYSIS

This section presents our approach and some of the technical details of programming language, static analysis, and runtime support for HAP. The basic goals for these elements have been outlined in the introduction. We begin by presenting a combination of techniques for each of these elements that clearly produce sound static and dynamic guarantees of policy enforcement. We then discuss on-going work to extend the class of programs and policies for which such guarantees can be made.

As discussed in the introduction, we introduce a novel language construct, the query-conditional, the conditional expression of which is a pure-past FOTL formula. This is the foundation of the program's determination whether send events it would initiate are legal, or when not, in the case of user initiated action, providing denial explanation. These queries are supported by a *history mechanism* in the runtime environment that uses a constraint system to record argument values for which the pure-past query formulas hold. This history mechanism is distributed across actors, with each actor recording only the send and receive events in which it has participated. Each actor maintains its own record of these communication events.

Policy and actor specifications are concerned with the subjects and attributes contained in regulated messages, but not with attribute values. Consequently, for regulated messages, the history mechanism need not record the entire message content (e.g., specific attribute values), but only needs to update its constraints based on which attributes are being transmitted. While program variables can contain attribute values, we rely on the types of program variables and messages to identify what attributes are contained. For instance, if an x-ray technician requests that the system sends a patient's x-ray to his doctor for diagnosis, the static type of the data structure holding the x-ray will indicate that it is that patient's x-ray. This information, which is encoded in the type of the x-ray data, rather than the x-ray data itself, will be passed to the actor's history mechanism, so that the mechanism can update its constraints to reflect the send. In addition to labeling structures containing protected information with the attributes they contains and the principals that protected information is about, types are also used to statically relate principals variables with the related roles of the principals they contain.

This same type information is also supplied to a static program analysis tool. This static program analysis tool verifies each actor independently against its specification. It verifies that each send in an actor complies with the safety clauses of that actor's specification, and that the obligations described by the liveness clauses of an actor's local specification are eventually satisfied.

7.1 A Language for History-Aware Programming

Actor Structure

The actors comprising the information system are instances of actor classes whose definition is written in the History-Aware Programming Language (HAPL). Each actor class consists of a list of local state variables and a list of action-methods. Each action-method matches some message signature and its body contains imperative code that should be executed whenever a message of that signature is received by the actor.

A simple actor class reflecting this structure is shown in Figure 7. It is written in HAPL$^-$, an untyped variant of HAPL. (The type system will be discussed after we introduce HAPL's basic structure.)

```
actor Forwarder {
  var forwardee = SinkActor()

  action (sender) setForwardee(a) {
    forwardee = a
  }
  action (requester) getForwardee() {
    send requester theForwardee(forwardee)
  }
  action (sender) stuff(m) {
    send (forwardee) stuff(m)
  }
}
```

Figure 7: A Simple HAPL$^-$ Actor Class

Instances of this `Forwarder` actor class contain a local state variable `forwardee` and responds to the messages `setForwardee`, `getForwardee`, and `message`. If a message whose signature matches one of the corresponding actor-classes methods is received by an instance of the actor class, the body of that action-method will be executed. (Messages that do not match any action-method signature are discarded.) When executing the body of an action method, formal parameters (e.g., `a`, `m`) are bound to the components of the contents of the message causing the action-method to be executed. The variable `self` is bound to the actor instance that received the message. The body of `setForwardee` simply updates the variable `forwardee` based on the parameter. The actor responds to the `getForwardee` messages by sending the contents of `forwardee` to `requester` The actor responds to the `stuff` message by sending a message to the `forwardee` that is identical to the one it received.

Local Events and Temporal Queries

Each actor's runtime execution generates a local history. An actor's local history consists of events corresponding to the actor receiving a message m from a ($\mathrm{receive}(self, a, m)$, which occurs whenever the execution of an action-method begins), the actor sending a message m to a ($\mathrm{send}(self, a, m)$, which corresponds to the execution of a send statement within an action-method), and the actor concluding its response to a message ($action\text{-}termination$, which corresponds to the termination of an actor-method). For technical reasons, the history also includes $actor\text{-}idle$ events. These are generated non-deterministically when an actor-method is not executing. Since each actor has only a single thread of execution, action-method executions are not concurrent, and a given actor responds to at most one message at a time. Thus, receive and $action\text{-}termination$ events are paired.

In writing actors whose behavior always complies with specifications written in temporal logic, it is often useful to be able to condition the actor's behavior on past events corresponding to clauses in the specification. This allows, for example, one to write code that will send privacy policies to the users only if it has not sent a policy to them before. This is supported by allowing the actors definition to contain temporal queries consisting of local pure past formulas. The queries can then be used to guard blocks of statements, such that a guarded block only gets executed if the local history satisfies the temporal query at its current position. This could be employed in our `Forwarder` example to only forward one messages from actors that have previously sent `Forwarder` a `getForwardee` message, by changing the `stuff` action-method to:

```
action (sender) stuff(m) {
  if (◇receive(self, sender, getForwardee())) {
```

```
    send (forwardee) stuff(m)
  }
}
```

These queries can be supported efficiently using the dynamic programming approach from Krukow et al. [21] that was described in Section 2.

Types, Principals, and Protected Information

Static verification of programs against privacy policies requires reasoning about principals, their roles and protected attributes, and so HAPL introduces types that can be used to statically represent principals, roles, attributes and their relationships. In addition to being used within type system to verify they are being used consistently, this type information is exposed to the static program analysis, so that it has additional static information about the principals involved in various operations.

This additional information is incorporated into the type system by introducing type-level variables for principals (that were inspired by Tse and Zdancewic's Runtime Principals [32]). In our context, a type-level principal variable provides a static representation for an element of $\mathcal{P}$ that is unchanged for the duration of that type-level principal variables scope. These type-level principal variables do not have a runtime representation, but rather are used to parameterize the types for program-level principal values, actor values, and structures containing protected information. Additionally, actor classes and actor methods may be made polymorphic over principals by having them take type-level principal variable parameters, which can be bound to have specific roles.

As an example, consider the following action-method, which receives the message testResult containing the address of an actor `docAddr` which is authorized to receive information for the doctor and test result of some patient, and which then sends the test result to the doctor:

```
action[α, β, γ; inrole(β, patient) ∧ inrole(γ, β, doctor)]
    (sender:Actor[α])
    testResult(docAddr:Actor[γ], result:Attribute[β,TestResult])
{
        send (docAddr) myPatientsResult(result)
}
```

This action-method is parameterized by the type-level principal variables α, β, and γ, which are constrained such that β is a patient and γ is β's doctors. These type-level principal variables are then used to construct types for the messages sender and the message's contents. The type, `Actor[`α`]`, on `sender` allows the sender to be any actor that acts on behalf of the principal represented by α, which is unconstrained. The type `Actor[`γ`]` on `docAddr` requires that `docAddr` be a reference to an actor that acts on behalf of the principal γ, who was previously constrained to be a doctor of β. The type of `Attribute[`β`,TestResult]` on result denotes that `result` contains values of the attribute TestResult about β, who we knew previously to be a patient. Only the values that have Attribute types are allowed to contain protected information, and they are only allowed to contain the attribute(s) provided to the Attribute type constructor.

Another important consideration for expressivity, is that we allow data structures to be packed into values with existentially quantified types, which are also used in Tse and Zdancewic's Runtime Principal work [32]. This is necessary to allow multiple related values to be stored in a data structure of unbounded size and still have their relationships preserved (e.g., a list of 3-tuples consisting of a patient, his doctor, and his medical chart).

7.2 Static Program Analysis

We first transform each of the specification's liveness clauses into a *termination safety clause* that requires that, when an action-method terminates, any obligations incurred since the start of the execution of that action-method (including any obligations incurred by the receive event that marks the beginning of that action-method's execution). The conjunction of this with the safety clauses forms a *verification formula* that must be shown to hold whenever the assumption clauses hold.

For a liveness clause

$$\forall \vec{x}\,.\,\psi'(\vec{x}) \rightarrow \Diamond \exists a{:}A.\,\exists m{:}M.\,\mu'(a,m,\vec{x}) \wedge \mathrm{send}(\mathit{self},a,m)$$

we can create a termination-safety clause of the form

$$\begin{aligned} &\mathit{action\text{-}termination} \\ &\quad \rightarrow \forall \vec{x}\,.\,\neg\psi'(\vec{x}) \\ &\qquad\quad \mathcal{S}\,(\exists a{:}A.\,\exists m{:}M.\,\mu'(r,m,\vec{x}) \wedge \mathit{send}(\mathit{self},a,m)) \\ &\qquad\qquad \vee\,(\exists a{:}A.\,\forall m{:}M.\,\mathrm{receive}(\mathit{self},a,m) \wedge \neg\psi'(\vec{x})) \end{aligned}$$

Intuitively, this termination-safety clause requires there to not have been a triggering event since either the corresponding obligation was fulfilled or there was a receive that was not itself a triggering event. When combined with the proposition that every execution of an action method terminates ($(\exists a : A.\,\forall m : M.\,\mathrm{receive}(\mathit{self},a,m)) \rightarrow \Diamond\, \mathit{action\text{-}termination}$) and that receive / termination pairs are not nested, this entails the original liveness formula. This is because, since ψ' is a local pure past formula and can only include send and receive events, the triggering condition ψ' can only ever be satisfied at some position in a trace if it was satisfied at the previous position, if the current position is a receive event, or the current position is a send event (which only occurs between a receive and its $\mathit{action\text{-}termination}$). Therefore, at any position that satisfies the triggering event, there will eventually be a position (before some $\mathit{action\text{-}termination}$) at which the corresponding obligation is fulfilled. This transformation allows us to convert a liveness requirement of the actor into a condition that can be verified independently for on each actor-method.

If we let A-TermSafe be the set of temporal logic clauses obtained by transforming each of the clauses in A-Liveness into its corresponding termination-safety clause, then we can define an actor's *verification formula* as follows:

$$(\boxminus \bigwedge_{\psi' \in \text{A-Assum}} \psi') \rightarrow (\bigwedge_{\psi' \in \text{A-Safety}} \psi' \wedge \bigwedge_{\psi'' \in \text{A-TermSafe}} \psi'')$$

The syntactic form of the verification formula is such that, if it is satisfied at every position non-$\mathit{actor\text{-}idle}$ position in the trace, the actor's specification will also be satisfied by that trace. (This is because, each clause, is incrementally monitorable and is always satisfied at non-$\mathit{actor\text{-}idle}$ positions.)

Static Analysis of Actor Code.

We use static program analysis to ensure that the verification formula for each action-method is satisfied by reaching local event histories. Given an actor, we say that a local event history of that actor *reaches* a program location in an action-method if at some step in some system execution, the actor's thread reaches that program location with that local history.

Our analysis is an iterative dataflow analysis, which given an action-method, computes, for each program location, information about the histories that reach that location. This information takes the form of 3-valued truth assignments over formulas that are derived from the verification formula for the given action-method and

from the temporal queries appearing in the action-method. Each formula that is assigned "true" is satisfied by every reaching history. Each that is assigned "false" is not satisfied by any reaching history. Finally if the formula is neither known to be satisfied by every reaching history nor is it known to be satisfied by no reaching history, then the formula is assigned "unknown."

In the dataflow analysis, we cannot interpret logical variables as ranging directly over carrier elements. Instead, we must associate them with static program constructs, such as the types of program variables that at runtime, take on values in $\mathcal{P}$. However, the semantics of those program constructs must then be grounded in terms of carrier elements. These two steps of interpretation are necessary to establish statically the relationship between specifications and programs. However, it requires a certain amount of machinery. In our policies, we have no logical variables of sort R. The treatment of logical variables of sorts A, T, and M are relatively straightforward. However, treating logical variables of sort P is rather intricate, so we focus on them here.

To explain which formulas are "derived," we use the following notions. Within the context of dataflow analysis, we use a different set of symbols than are used in the semantics of FOTL. They are syntactic program constructs, namely type-level principal variables, as well as a new collection of anonymous principal variables. We call these symbols *symbolic principal variables* (SPV) and we call a function from logical variables to SPVs a *static logical environment* (SLE). We call a formula ϕ' a *static instance* of a formula ϕ if ϕ' is obtained from ϕ by substituting each free variable in ϕ according to some SLE. The formulas whose 3-valued truth values we wish to track are instances of subformulas of the verification formula or a temporal query formula. We call these subformulas *formulas of interest*.

The dataflow analysis computation simulates the effect that executing the program has on the truth value of formulas of interest by using SLEs. For instance, suppose the information system belongs to a medical clinic. Also suppose that an action-method poses a temporal query asking whether the principal carrier element held in a variable whose type is parameterized by type-level principal variable α has sent the clinic permission to release his medical records to a principal whose type is parameterized by β. On entry to the then-branch of the query-conditional, the static analysis assigns the value "true" to the FOTL formula that says the principal denoted by α has sent permission to release his medical records to the principal denoted by β. So if there is a send operation initiated by the clinic that transmits α's medical records to β, presumably the action-method specification states that doing so is permissible, and the send operation is validated by the static analysis.

In addition to setting a particular subformula to true or false, the query transfer functions must also closure operation to determine what other subformulas can be determined to be true or false. This is done using a generalization of a three-value version of the procedure used by Krukow *et al.* [21], which propagates information from subformulas to formulas. Since our equations can learn new facts (e.g., the condition at temporal queries and assumption clauses at method entry) that were previously true but not known to be true, it is necessary also to propagate information from formulas to subformulas.

Additionally, as with the procedure of Krukow *et al.*, the transfer functions for send, $\mathrm{receive}$, and $\mathit{action\text{-}termination}$ events need to 'roll time forward' and adjust the formulas of interest accordingly.

7.3 Current Limitations

There are several limitations of our current work that we wish to address in the future. In our current work, we assume that roles are fixed; it will clearly be necessary to treat role change in order to support things the acquisition of new patients. Additionally, we assume that information that the static type system provides to the static analysis and dynamic queries about roles and attributes is complete. A more realistic model would allow the static system to represent over- and under-approximations of the roles certain individuals might have. Another limitation is that we restrict each actor/component to stand for exactly one principle; in the future, we hope to look at the possibility of supporting more flexible mappings between actor actions at the implementation level and the actions of principles at the policy level.

In the future, we also plan to add support programmer-defined invariants that can include local program state along with temporal events. These can be supported by extending the programming analysis to include predicates for values being in data structures, and by incorporated the invariants into the verification formula.

8. CONCLUSION

In this paper we have outlined a methodological framework for designing, developing, and verifying information systems that provably enforce privacy policies. Several novel, open technical problems arise in this context. We have inventoried several of these and solved many of them here. These technical contributions will act as corner stones on which we are actively building in on-going research. The paper is the first to report on this long-term research objective. Completion of this objective will take a number of years, and the number of open problems that must be solved will require several papers and articles to report.

9. REFERENCES

[1] Health resources and services administration, 1996. Health Insurance Portability and Accountability Act, Public Law 104-191.

[2] Federal trade commission, How to comply with the children's online privacy protection rule, 1999. Public Law.

[3] Senate banking committee, Gramm-Leach-Bliley Act, 1999. Public Law 106-102.

[4] G. A. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(01):1–72, 1997.

[5] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.

[6] B. Alpern and F. B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:117–126, 1987.

[7] P. Ashley, S. Hada, G. Karjoth, C. Powers, and M. Schunter. Enterprise privacy authorization language (EPAL 1.2), 2003.

[8] A. Barth, A. Datta, J. C. Mitchell, and H. Nissenbaum. Privacy and contextual integrity: Framework and applications. In *IEEE Symposium on Security and Privacy*, pages 184–198, 2006.

[9] A. Barth, J. Mitchell, A. Datta, and S. Sundaram. Privacy and utility in business processes. In *20th IEEE CSFS*, pages 279–294, 2007.

[10] L. Bauer, J. Ligatti, and D. Walker. More enforceable security policies. In *Foundations of Computer Security*, 2002.

[11] T. Breaux and A. Antón. Analyzing regulatory rules for privacy and security requirements. *IEEE Trans. Softw. Eng.*, 34(1):5–20, 2008.

[12] E. Y. Chang, Z. Manna, and A. Pnueli. Characterization of temporal property classes. In *ICALP*, pages 474–486, 1992.

[13] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The ponder policy specification language. In *POLICY*, pages 18–38, 2001.

[14] E. A. Emerson. Handbook of theoretical computer science. chapter Temporal and modal logic, pages 995–1072. 1990.

[15] D. Harel and A. Pnueli. On the development of reactive systems. In *Logics and Models of Concurrent Systems*, pages 477–498, 1985.

[16] M. Harrison, W. Ruzzo, and J. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8):461–471, 1976.

[17] K. Havelund and G. Roşu. Efficient monitoring of safety properties. *Int. J. Softw. Tools Technol. Transf.*, 6:158–173, 2004.

[18] C. Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8(3):323 – 364, 1977.

[19] I. Hodkinson, F. Wolter, and M. Zakharyaschev. Decidable fragments of first-order temporal logics. *Annals of Pure and Applied Logic*, 106(1-3):85–134, 2000.

[20] B. Jonsson and T. Yih-Kuen. Assumption/guarantee specifications in linear-time temporal logic. *Theoretical Computer Science*, 167(1-2):47 – 72, 1996.

[21] K. Krukow, M. Nielsen, and V. Sassone. A logical framework for history-based access control and reputation systems. *J. Comput. Secur.*, 16(1):63–101, 2008.

[22] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.

[23] M. Leucker and C. Schallhart. A brief account of runtime verification. *Journal of Logic and Algebraic Programming*, 78(5):293–303, 2009.

[24] J. Ligatti, L. Bauer, and D. Walker. Run-time enforcement of nonsafety policies. *ACM Transactions on Information and System Security*, 12(3), January 2009.

[25] M. J. May, C. A. Gunter, and I. Lee. Privacy apis: Access control techniques to analyze and verify legal privacy policies. In *CSFW*, pages 85–97, 2006.

[26] F. Mogavero, A. Murano, and M. Y. Vardi. Relentful strategic reasoning in alternating-time temporal logic. In *international conference on Logic for programming, artificial intelligence, and reasoning*, pages 371–386, 2010.

[27] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, volume 526, pages 46–67, 1977.

[28] A. Pnueli, Y. Rodeh, O. Strichmann, and M. Siegel. The small model property: how small can it be? *Inf. Comput.*

[29] G. Roşu and K. Havelund. Synthesizing dynamic programming algorithms from linear temporal logic formulae. Technical report, 2001.

[30] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3:2000, 2000.

[31] C. L. Talcott. Composable semantic models for actor theories. *Higher-Order and Symbolic Computation*, 11(3):281–343, 1998.

[32] S. Tse and S. Zdancewic. Run-time principals in information-flow type systems. In *IEEE Symposium on Security and Privacy*, pages 179–193, 2004.

[33] L. D. Zuck and A. Pnueli. Model checking and abstraction to the aid of parameterized systems (a survey). *Computer Languages, Systems & Structures*, pages 139–169, 2004.