A service-oriented privacy model for smart home environments

by

Ryan Michael Babbitt

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
Johnny Wong, Major Professor
Simanta Mitra
Yong Guan

Iowa State University

Ames, Iowa

2006

TABLE OF CONTENTS

1. INTRODUCTION

Smart home technology is an application of ubiquitous computing that equips living environments with different types of sensors, actuators, and appliances under computer control to improve the quality of life for inhabitants. Services such as health and behavior monitoring, personalized customization of home operation, control and automation of the environment, and assistance with physical or mental tasks enable inhabitants to live safer, more comfortable, and more independent lives. Many commercial and research efforts are investigating the vast potential that smart homes and related products provide to assist the activities of daily living. However, the resulting efforts frequently suffer from two main limitations that hinder their widespread use. First, resulting products are usually proprietary, offering closed services that are tailored to specific applications and cannot be easily reused or extended by other services in the smart home. Second, the invasiveness of the technology and use of personal information may allow the privacy of the inhabitants to be violated.

We have previously addressed the privacy issue by calling for a privacy policy-based framework [1][2] to control the collection, storage, use and dissemination of personal information in smart home environments. This framework supports several high level goals, including promoting inhabitant awareness of the abilities of devices/services contained in the smart home space, using privacy policies that express the contextual nature of privacy, providing mechanisms and tool support for the authoring, deployment, enforcement, and auditing of privacy policies, as well as creating and verifying policy models to detect conflicts and incorrect specification of privacy policies. In this thesis, we focus on the modeling and verification of policies by proposing a combination of the service-oriented computing and privacy policy paradigms to create a preliminary privacy model for smart homes. We then offer an example scenario and discuss how we employ model checking techniques to verify various aspects of our proposed policy model. The major contributions of this work are four-fold:

- We extend the notion of personal privacy to include the control of how household objects are used by smart home services.
- We introduce the use of service-oriented computing to bind resources to the policy space.
- We define a novel service-oriented privacy policy model that authorizes both the flow of personally sensitive data and the control of environment objects based on inhabitant preferences and various system contexts.
- We introduce the use of model checking techniques to verify correctness properties of privacy policy models and their enforcement.

The rest of the paper is organized as follows: section 2 gives background information about smart homes, information privacy, policy-based management in distributed systems, and model checking, section 3 presents existing privacy analyses and policy models, section 4 presents our novel privacy model, section 5 illustrates with an example scenario how model checking can be used to verify our privacy model, and section 6

concludes with discussion and future work. Appendix A contains model implementation files, and Appendix B contains property specification files.

2. BACKGROUND

This section offers background information about smart homes, service oriented architectures, information privacy, policy-based management concepts in distributed systems, and model checking.

2.1 Smart Homes

As mentioned previously, smart homes and related products are meant to improve the quality of life for inhabitants by incorporating different types of computer controlled sensors, actuators, and appliances in a regular living environment and coordinating these devices in ways beneficial to inhabitants. Sensors in the home read information about objects and individuals in the smart home space, and actuators affect changes to objects in the smart home. If the services that use information from sensors are remotely managed or monitored, then there is the possibility for uninvited or unknown leakage of personal information to these external systems. These consequences are more severe if data are exchanged among these third party systems or released to the public, such as happened with five million customer travel records in the JetBlue Airways incident of 2002 [4]. On the other hand, if the services that invoke the actuators are remotely managed, there is a possibility for external entities to manipulate devices and services in a manner that may disturb the inhabitant or place the home in an undesired state. Any of these situations presents possible privacy vulnerabilities in the smart home.

There are other challenges to managing privacy in smart homes as well. Since many different academic and industry projects are developing smart home products and applications, various hardware and software technologies are being used, making smart home environments quite heterogeneous, not only in terms of the types of devices and services present, but also in terms of the platforms used to support them. Also, as devices and services evolve and new ones come to market, new components will be added to the home, and old ones will be removed or reconfigured. Thus, smart homes will thus also be dynamic in nature. In order to achieve scalable, extensible, and interoperable privacy solutions under these conditions, some sort of open software framework built on existing and emerging standards must be employed [3]. Several smart home efforts [5][6][7][8] attempt to address these problems in various ways. The Gator Tech Smart House [5], in particular, is an open, interoperable infrastructure that uses the service-oriented computing paradigm to manage devices and services. In our opinion this framework, though still at the conceptual stages of development, represents the best existing approach to manage the complexity inherent to smart homes. We therefore use it as a reference point for our privacy policy model by focusing on the type of service-oriented environment that the Gator Tech Smart House uses.

2.2 Service-Oriented Computing

The service oriented paradigm focuses on encapsulating computational resources into service objects, which are “self-describing, open components that support rapid, low-cost composition of distributed applications.” [9] The service descriptions are used to encode the capabilities and characteristics of the services

so the basic operations of publication, discovery, selection, binding, and invocation can occur in a loosely-coupled fashion. Service-oriented architectures (SOA) are frameworks that facilitate the life cycle management, composition, and change management of services including monitoring, coordination, and conformance of services [10]. Some of these frameworks are centralized, like OSGi [11], and others are decentralized, like the web services suite (SOAP [12], WSDL [13], and UDDI [14]), UPnP [15], and Jini [16]. The service-oriented paradigm, particularly web services, has been widely adopted to achieve interoperability in enterprises and, as evidenced by the Gator Tech Smart House, is also showing promise for managing the complexity of smart home environments. We investigate the use of service-oriented computing to bind smart home device and service resources to the privacy policy space by utilizing service descriptions that include terms pertinent to privacy.

2.3 Information Privacy

When attempting to understand privacy, it must be noted that it is an inherently complex concept involving relationships between society, technology, the individual, and the law [17]. As new technologies emerge to collect, process, and disseminate (personal) information, laws are created to regulate or standardize intended and unintended uses of the technology to limit harmful effects on societies and individuals. This, in turn, influences which technologies are developed, and the cycle continues in a similar feedback loop, becoming even more complex when considering the differing attitudes about privacy across cultural or legal boundaries [18].

2.3.1 Privacy and Social Science

Though privacy can be a somewhat vague notion, Margulis [19] cites two definitions of privacy that have made a significant impact in western nations in the late 20th century. These theories are the contributions of social scientists Alan Westin and Irwin Altman. We now briefly highlight these theories to show that our personal policy-based approach aligns with contemporary social science views of privacy.

Westin's defines privacy as "the right of individuals, groups, or organizations to determine when, how, and to what extent information about them is communicated to others" which is realized by freeing oneself from observation by others, promoting close relationships, avoiding identification or surveillance, and limiting disclosure to others [19][20]. To Westin, privacy is a dynamic process of "voluntary and temporary withdrawal of a person from general society" that enables personal autonomy, release from social pressures, self evaluation, and protected communication [19][20]. Similarly, Altman defines privacy as "the selective control of access to the self" [19][21]. To him, privacy is comprised of five dimensions: the change of interpersonal information boundaries over time, the disparity between desired and actual levels of privacy, the existence of a privacy spectrum that ranges from social isolation (when actual privacy is greater than desired privacy) to crowding (when desired privacy is greater than actual privacy), dependence upon inputs from others and outputs to others, and a hierarchical nature of privacy at individual and group levels [19].

Though Westin focuses more on the end results and internal motivations of privacy and Altman focuses on the environmental and external motivations of privacy, together these theories involve processes of controlling information communicated *to* others and information received *from* others. This motivates the use of personal privacy policies in the smart home to manage both personal information flow (i.e. access to self) and control the effects outside entities can have on the smart home (i.e. boundaries to keep out external influences).

2.3.2 Privacy and Regulation

In recent decades, regulatory bodies, particularly in Western nations, have sought to investigate and codify the principles that should regulate systems that collect and use personal information. The results are aggregately known as the Fair Information Practices (FIP) and are widely accepted as a basis for any privacy aware or enabling technology. We have previously categorized various FIP representations from the Organization for Economic Cooperation and Development (OECD) [22], the U.S. Federal Trade Commission (FTC) [23], and the Canadian Standards Association (CSA) [24] into four general themes: data practice specification, personal consents, enforcement/safeguards, and accountability/audit [1]. In the following discussion "data owner" refers to the person about whom data is collected, "data collector" refers to the entity that collects and stores the data, and "data user" refers to any entity that accesses data from the collector.

According to the FIP, data practice specification requires data collectors and data users be transparent about their procedures using personal data, including such information as who is making the request, what data is needed, who can access the data (including the data owner), for what purposes the data will be used, and how long the data will be stored, as well as the security precautions taken during collection and storage. Also, the FIP deem that changes made to a policy affecting information previously collected should be made apparent to the original data owners as well. In short, a data practice specification should allow a data owner to have a high level comprehensive knowledge about what happens to personal data so that a well-informed decision can be made whether or not to allow its collection. To make this decision efficiently, individuals create a list of privacy preferences regarding their personal data and related data practices. After this list is created, a representative software agent can give consent on behalf of the data owner by evaluating the data practice specification associated with an access request against those preferences. If they match, the information exchange is allowed and results in an agreement between data owners and data collectors/users about acceptable use and management of personal data. Enforcing these agreements then requires upholding the data practices presented when consent was obtained from the data owner during collection. This includes restricting access to appropriately authorized individuals, restricting information use to the purposes for which it was collected, fulfilling any obligations incurred during this process, and ensuring the proper security and accuracy of data during its transportation, storage, and use. Lastly, data collectors and users should be held accountable for their specification and enforcement of data practices. This is done in four ways: 1) giving data owners the chance to verify the accuracy of data practice policies, 2) allowing data owners to verify the quality and accuracy of personal information and correct it if need be, 3) auditing data collectors and users for compliant enforcement of

data practices, and 4) providing mechanisms for data owners to challenge compliance to data practice specifications and get complaints resolved.

2.4 Policy-Based Management

Policy-based management techniques have been useful for a number of distributed system domains including network resource management [25] and security [26]. As such, we now review common policy-related terms, types of policies, and issue of conflicts in policies. Definitions and concepts are taken in part from [27][28][29]. A *policy* is a collection of rules that govern the operation of a system. These rules are created by one or more *policy authorities* and contain associations of *rulings,* or possible access decisions, to sets of *subjects,* entities the policy applies to, *targets,* the resources which are being governed, and *actions,* the operations taken by subjects on targets. Additionally, rule applicability to a request may be limited by requiring some set of *constraints* or *conditions* about states or attributes of the system to be true before using a rule, and *obligations* may be mandated to be carried out by the subject before, during, or after the access is granted. Lastly, policy terms are frequently organized into hierarchies in order to reduce the complexity in specifying rules. These hierarchies usually form a directory-style tree in which parent elements generalize and classify their children elements according to similar function, purpose, properties, or management. Using hierarchies reduces the complexity of specifying rules by allowing terms to be specified at higher levels of granularity, but may introduce an overhead as rulings may need to be distributed up or down the hierarchy as abstract rules are "unfolded" into concrete ones.

One major concern in policy-based systems is that of preventing, detecting, and resolving conflicts within a policy or between separate policies. There are two main types of conflicts: *modality conflicts* and *application specific conflicts*. Modality conflicts occur when opposite rulings (i.e. allow and deny or obligate and refrain) are allowed for the same request which occurs because of overlapping terms (subject, objects, and actions). The conflicts may be a result of errors in policy specification and thus produce ambiguous or undefined behavior during evaluation. Application specific conflicts, on the other hand, result from incomplete knowledge of the system being managed (i.e. unknown or unplanned dependencies between objects, subjects, or actions out of the scope of the policy model). Standard examples of application specific contexts include conflicts of duties, resource/priority conflicts, multiple policy managers, self management, and conflicts of interests [30]. We discuss later how to use model checking to detect various types of conflicts in our policy model.

2.5 Model Checking

Model checking is a discipline within the field of formal methods used to verify the absence of logical errors in a system [32]. First, a logical model of the system behavior is created. The specification of this model may be symbolic, mimicking high level programming language constructs or state based, viewing a system as a set of states and transitions between them. In either case, the specification is converted to a state graph of the

system that may have an infinite number of paths or states. Next, some sort of property about the system behavior is specified and checked against the model's execution. There are two basic types of properties: *safety* properties, which specify that an unwanted state is never reached, and *liveness* properties, which specify that eventually a desired state is reached. When the model is executed, its state graph is traversed and compared with the property being checked. If the property is ever violated, the verification fails, possible resulting in an error trace that can be used to refine either the model or the property. The failure of a property may result from errors in the logic of the physical system but can also be attributed to incorrect specification or abstraction of the model or the property. Reducing the complexity of the model by including only the necessary system behavior decreases the size of the model, expedites verification, and can help this debugging process. In this work, we use model checking to verify the absence of modality conflicts and the correctness of policy evaluation. This process is described in section 5.

3. RELATED WORK

Some work has been done in recent years to analyze privacy in ubiquitous environments and create privacy policy models for various aspects of the data life cycle. Two policy model paradigms that have been well-studied are data collection policies, specifically for the Internet, and data usage policies, specifically in enterprise systems. We now discuss this related work.

3.1 Privacy Analyses in Ubiquitous Computing

Privacy theories and FIP have been incorporated by researchers in many variations into design goals for privacy management. Bellotti and Sellen [33] proposed an analysis centered on feedback and control for collaborative media spaces. Langheinrich [34] analyzed privacy based on such implications of pervasive computing as ubiquity, invisibility, sensing, and memory amplification. His analysis augmented the FIPs with principles of anonymity and pseudonymity of personally identifiable information and device operation based on the proximity of data owners and users to devices. Jiang [35] suggested a privacy model that seeks to minimize asymmetric information flow between data owners and data collectors/users over physical, social, activity-based boundaries. Palen and Dourish [36] take a similar approach based on Altman's privacy theory by viewing privacy as ongoing boundary negotiations between the public and private and the self and others over time. Hong et al. [37] suggest a privacy risk analysis approach to handling privacy that has social, organizational, and technical focuses. Price et al. [38] propose a high level privacy framework that allows for various types of uncertainty to be introduced to data and is composed of a four layer analysis: the regulatory regime users are in, the type of service they require, the type of data being disclosed, and their personal privacy policy.

In contrast to these analyses, we have previously outlined an approach to privacy management based on controlling data at all states in its lifecycle and emphasizing privacy related internal, external, and environmental contexts [2]. Data context concerns the subject matter of data, its accuracy, its precision, and its freshness. Inhabitant context refers to various states or properties of the inhabitant such as location, activity, financial situation, health, and personal preferences. Environment context denotes the states of the smart home that may influence privacy such as time, safety and security concerns, and various operational details of devices and services. Lastly, request context assumes most of the content of the Fair Information Practices such as who is using data, what is done with it, how long it is kept, etc. In [2] we suggest using separate policies for major states in the data lifecycle, concentrating on data collection, storage, and access. In [1], we raise issues associated with privacy policies and requirements for privacy in smart home systems and give a case study illustrating many of these issues.

In addition to these analyses, various privacy models have been suggested in the literature, particularly in the areas of data collection and data usage. We now discuss the most prominent example of data collection privacy, W3C's Platform for Privacy Preferences, and several examples of data access models and their applicability to smart homes.

3.2 Privacy Policies for Data Collection

The Platform for Privacy Preferences (P3P) [39] is a W3C standard for enabling protection of consumer information over the Internet. A P3P privacy policy is an XML-based collection of statements containing a list of data types being collected, the purpose for their collection, the length of data retention, a list of parties that have access to the data, and a high-level description of the consequences of this data collection. To be useful though, a P3P policy must be used in conjunction with a privacy preference specification language such as APPEL [40] or XPref [41]. At the client side, a P3P-enabled browser, configured with user preferences, requests a webpage and examines any P3P privacy policies associated with it to decide if personal information should be transferred via the webpage. The main contribution of P3P is its emphasis on creating machine-readable policies that can be retrieved and viewed when a data request occurs. However, P3P suffers from commonly known problems of being domain specific to the Internet, lacking clear definitions of terms, not defining formal policy semantics, and not being linked to an enforcement mechanism. These problems result in not being able to specify acceptable behavior, rejecting good policies, and accepting incorrect policies [41][42].

Despite these shortcomings, P3P has been suggested for ubiquitous systems by Langheinrich [43] with the introduction of a Privacy Awareness System in which the data practices of devices and services are collected and evaluated against user preferences. Though this work addresses some key issues of privacy in ubiquitous environments such as the energy conservation of devices and different policy announcement techniques, the system still suffers from the major drawbacks of P3P and associated preferences languages. Toward this end, Ackerman [44] identifies the need to augment P3P with context awareness for social and organizational environments, support for more varied data types, and assistance mechanisms to aid users in mediating the complexity of privacy preferences. Hong et al. [45] contribute to this goal by extending P3P with three new data elements: time, location, and data owner, and two new data categories: application groups and user groups. These works are major improvements for adopting P3P into ubiquitous and smart home technologies, but more research needs to be done to create a formal semantics for P3P, extend its syntax, and link it to enforcement mechanisms to achieve successful privacy management in smart home environments.

3.3 Enforcing Privacy during Data Usage

P3P can only be used to describe the data practices of websites. It does not guarantee that these practices are followed or that policies are enforced. Therefore, in addition to P3P, some sort of enforcement mechanism must be used. Several such privacy models already exist [46][47][48][49][50][51][56], but they are only logical specifications that neglect to address the binding of resources to the policy space, often lack proper support for contextual needs of smart home privacy, and have not been formally verified in any way.

The InfoPriv model [46][47] is an early privacy model for analyzing information flow within an organization. It is a graphical model in which vertices represent data users as well as data objects, and edges represent some type of information flow or information flow constraint. Graph analysis can then be used to

determine whether information is allowed or prohibited between vertices and to isolate and resolve conflicts. The model has the advantage of being very simple to understand, but does not include multiple information flows, purpose, or contexts and is restricted to an intra-organization analysis.

A formal task-based privacy model is proposed in [48] that focuses on the FIP notions of purpose binding between collection and access and restricting data processing to necessary tasks. In this model, data objects and system tasks are both associated with purposes, and a user accesses data by way of a system subject (process) that has a set of authorized tasks. An access is allowed if it is necessary to the user's current task, if the user is authorized to perform the current task, and the data object's purpose and the task's purpose agree. This model yields a promising formalization of purpose binding, but it does not include provisions for contextual factors during access, and its notions of tasks and subjects are vague. The Privacy-Aware Role-Based Access Control model [49] extends the task-based model in [48] including purpose hierarchies, obligations, and conditions in its model, however, it still lacks a robust notion of environment and occupant contexts.

The Purpose-Based Role Access Control model [50] organizes users, purposes, and data types into hierarchies. Users may assume conditional roles based on attributes specified during role assignment, and data types/objects are associated with a both a list of allowed purposes and a list of prohibited purposes. These sets of purposes are then inherited in various ways along the data hierarchy. In order to solve conflicts caused by this inheritance of purpose over data types, the authors suggest separating purposes into *strong purposes*, which cannot be overridden by data subtypes, and weak purposes, which can be overridden by data subtypes. Though the concept of conditional roles is useful, purpose overriding does not appear to be a very intuitive solution to conflict management. Also, this access control model does not include notions of contextual rules or control of environment objects.

IBM's Enterprise Privacy Authorization Language (EPAL) [51] and the Enterprise Platform for Privacy Preferences (E-P3P) [52] are high level languages for the internal enforcement of a company's privacy policies. Each enterprise defines its own data users, data types, actions, purposes, conditions, and obligations, with users, data types, and purposes arranged in hierarchies. Subsets of these policies have been shown to support operations such as combination, union, intersection, refinement, and equivalence [53][54], but the model lacks some flexibility in request specification [55]. The advantages of this model are the hierarchical grouping of elements, well-defined processing/access logic, and a supporting policy algebra. However, this model also does not include a robust notion of context, and its self-definition of terms is a limiting factor for enterprise cooperation.

The InfoSpace model in [56][57] that approaches privacy from a social science perspective. Authors Jiang et al. introduce a principle called minimum asymmetric information flow, which seeks to balance the information that flows between data owners and data collectors/users. Central to this model are repositories of personal data called "information spaces" which are physical or logical collection of data objects delineated by location, social, or activity boundaries. In this model, all data objects are also described according to their

accuracy, confidence, and precision (collectively called data sensitivity) and are assigned authorized spaces by their owner(s). The access policy is called a privacy tag, and as data objects are moved from space to space, their privacy tags move with them, allowing a trusted infrastructure to detect and correct unauthorized boundary crossings. This model portrays a very advanced notion of context, including data sensitivity, location, activity, and social settings, but does not clearly include any notion of purpose bindings. Also, it is not clear how to integrate information spaces beyond a single organizational unit.

4. SMART HOME PRIVACY POLICY MODEL

The privacy models discussed in the previous section regulate access to resources by using abstract policy concepts such as objects, purposes, users, roles, conditions, and obligations. This level of abstraction, though useful for reasoning about policies, makes instantiating and verifying these policies difficult in real world situations and applications. Also, with the exception of the InfoSpace model, these models do not incorporate robust notions of contextual information or address the control of actuator effects on environment objects and devices. Lastly, none of these models explain how to bind physical resources to the abstract policy space, which is necessary to implement the policy models in a real system. To address these issues, we propose a privacy model based on the service-oriented computing paradigm, where services are software artifacts that control zero or more devices, interact with each other via invocations, and may be remotely monitored or operated. As previously noted, some approaches [48][49] do model software operations by using the concepts of subjects and tasks, where subject is a software entity that performs an operation, and a task is the goal of the actual operation performed. However, these approaches do not explain any procedure to establish the mapping between software executables and the subject and task sets. We now show how our service-oriented privacy model addresses these limitations. We first discuss several environment assumptions we have made and the high-level architectural components necessary to support our policy evaluation process. Then, we present the privacy policies in our model and their evaluation and discuss possible conflicts that may occur in the policy specifications.

4.1 Assumptions

We make several assumptions to reduce the scope of our privacy model. First, we base our model on a centralized service-oriented architecture, which we define as one in which service invocation, authorization, and execution are controlled by a central authority. Modeling a centralized SOA reduces the complexity of our privacy policy model by allowing a single trusted central set of components to make all service invocation decisions instead of having separate evaluation points and needing to distribute and coordinate policies among them. Though centralization is common to privacy policy literature discussed earlier, there are obvious scalability and reliability risks as more devices are managed, more data is produced, more services are used, and larger policies are created. Second, we assume a fault-free, closed, secure environment in which there are no covert channels. That is, all devices and services are known to the system, and all service descriptions have been authenticated and verified for accuracy. Though this may be an unrealistic assumption, it currently affords us the ability to trust all services and their invocations which greatly simplifies our model. Third, we assume that service invocation request, evaluation, and execution, happen as (near) instantaneous time periods. Having instantaneous service invocation and evaluation means that the context under which a request enters the system is very likely to be the same context in which the evaluation and service execution occur. Therefore, we know that a context-based rule applies the same way throughout a service execution. On the other hand, continuous

operations (like media streaming for example) occur over periods of time in which the system context may change. This makes contextually authorized accesses difficult to evaluate because there is no single system context that can be decisively associated with an operation or evaluation. It may be possible, however, to approximate continuous operations as a set of repeated discrete invocations (i.e. a sequence of image requests versus streaming), but this means that each individual request needs to be evaluated, which would increase system overhead. Some sort of caching-style optimization schemes may reduce this overhead. We plan to address the extension of our model to non-constant service executions, as well as the porting of our privacy model to a decentralized environment and addition of security considerations in future work.

4.2 Privacy Policy Model Architecture

To introduce our privacy management system, we briefly present a high level architecture that helps conceptualize our policy model. The components are described in terms of the features they provide that are used by the privacy management system. A summary of these components can be found in figure 1.

Service Registry	Policy Store	Context Manager	Privacy Manager
-stores service descriptions and binding information -returns this information to the Privacy Manager when requested	-stores the privacy policies in the system -returns a set of rules applicable to a given request when queried by the Privacy Manager	-retrieves context data from appropriate services and supplies it to the Privacy Manager	-receives service invocation requests -coordinates other components to evaluate a request -returns the access decision

Figure 1 Major components of our policy architecture

4.2.1 Privacy Manager

In assuming a centralized SOA, we acknowledge the existence of a core component or set of core components that intercept and authorize service invocations. We refer to this set collectively as the Privacy Manager. In our architecture, the Privacy Manager is the coordinator of the evaluation process. It receives service invocation requests, calls other components, and makes the final decision whether the access will occur. The specific request evaluation algorithm we use will be described in greater detail in section 4.5.

4.2.2 Service Registry

The Service Registry is a standard component of any SOA that stores the description and binding information for installed services. This information is published in the Service Registry so applications and other service objects can query for services against some criteria, bind to them, and invoke them. We need such a component to store the publicly available methods of a smart home service. When one service wants to invoke

an operation of another service, the request is passed to the Privacy Manager. The Privacy Manager may then retrieve any supporting information related to service descriptions from the Registry. The specific structure of service descriptions in our model is discussed in section 4.3.5.

4.2.3 Policy Store

Once policies are created, they must be stored in the system to evaluate future service invocations. The Policy Store component is responsible for this. Though this component would be in charge of keeping current versions of policies as well as past versions for policy traceability, for our current work, we assume that once a privacy policy is created, it is static and does not change. We plan to address the modeling of changing polices in future work. When the Privacy Manager needs the sections of a policy that pertain to a service request, it queries the Policy Store for that information. Because the system is assumed to be fault-free and secure, we trust the results retrieved.

4.2.4 Context Manager

The final architectural piece, the Context Manager bridges the gap between the dynamic nature of system execution and the policy evaluation. When run-time information is needed from the system to evaluate a context, it is the Context Manager's responsibility to retrieve this information from the appropriate component/service. In this work, we assume that the system allows the Context Manager total and reliable access to data and that the time cost of gathering of this data is negligible.

4.3 Privacy Policy Model Terms

Before describing the policies in our model, we describe and define the terms that are present in the policies. We first discuss the two main resources over which privacy rules are written: data items and environment objects. Then we discuss data users (i.e. individuals, organizations, or enterprises) and their access purpose. After that, we describe how these entities are represented in service descriptions, and finally, we define the policies in our privacy model. The dependencies between basic model entities and services are shown in figure 2.

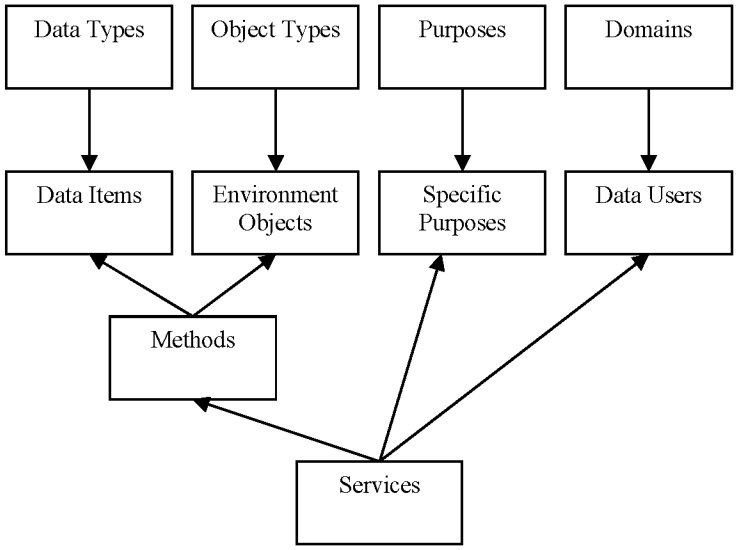

Figure 2 Dependencies of terms and services.

4.3.1 Data Items and Data Types

The first basic resource controlled by our privacy system is data. Sensors observe information about the home environment or inhabitants, and corresponding services allow this information to be recorded by the system and accessed by other services. A *data item* is the physical quantity being measured. Data items have *values* that are produced by the sensors. These values must fall inside a numerical or symbolic *domain* and may vary in accuracy and precision. Data items also have *types* that are an abstract classification of the data item. Having access to data values give software systems and their corresponding users knowledge about the state of the home or inhabitant. Though this information is needed to customize the operation of the services in the home, its access by certain users could violate an inhabitant's privacy wishes. Therefore, a data item or a data type is *privacy sensitive* if, for any value of that data type under any circumstance, the inhabitant wishes to restrict its possession by any party. Examples of data types that individuals usually consider privacy sensitive are identity/contact information, location, activity, health, financial status, and personal preferences. Existing literature [39][48][51][49][50] arranges data types into hierarchies in which parent data elements generalize their children. Data items are then mapped to a leaf node of the hierarchy. We follow this approach.

Definition 1: A *Data Type Hierarchy* is a 4-tuple $<T, T', T_H, T'_H>$, where

1. T is the set data items possible to be sensed by any smart home.
2. T' is the set of all abstract data types that organize data items in T.
3. T_H: $T \rightarrow T'$ is a function that associates a data item to its type.
4. T'_H: $T' \rightarrow T'$ is a function that maps a child data type to one parent data type.

4.3.2 Environment Objects and Object Types

The second basic resource used by smart homes, which is not addressed in current privacy policy literature, are the environment objects controlled by the house, like doors, windows, appliances, etc. Actuators control these objects, and services control the actuators. Thus, other services (and their corresponding users) can act upon objects in the smart home space by invoking those services that control objects and exert some measure of external control over the internals of the smart home. This loss of control of the home may result in situations that violate inhabitant privacy, either by increasing the sensitivity of data collected or by exploiting a conflict of interest for illicit gain. For example, invoking service that turns a light on next to a video camera increases the clarity of the image and the resulting sensitivity of the video, and invoking a service to open a window in the winter profits the electrical company because the heater must run more. Thus, a given environment object is *privacy sensitive* if, under any circumstance, the inhabitant wishes to restrict its control by any party.

In terms of restricting control of environment objects, there are several options of granularity to choose from. First, we could attempt to model the states of an object and create privacy policies based on these states. However, this option would require policy authors to familiarize themselves with all the states of all objects before writing knowledgeable policies. This would be too much work for inhabitants. Second, we could model the notion of *any* control of an object. Then, only the fact that an environment object transitions to *some* state needs to be known, the actual state space of the object would be irrelevant. This would make specification simpler and requires less knowledge of objects but would give less accurate control of the home. We opt to model the second option for the ease of specification and define an Object Type Hierarchy as follows:

Definition 2: An *Object Type Hierarchy* is a 4-tuple $<O, O', O_H, O'_H>$, where

1. O is the set of all environment objects available to be controlled by any smart home.
2. O' is the set of all the object types that categorize object in O.
3. O_H: $O \rightarrow O'$ is a function that maps an environment object to its type.
4. O'_H: $O' \rightarrow O'$ is a function that maps a child object type to one parent object type.

4.3.3 Data Users and Domains

Services may be deployed and used inside the smart home on behalf of various organizations, enterprises, or individuals. For example, a location tracking system in the house may be monitored by the police department (an organization), a private security company (an enterprise), a caregiver, or a family member (individuals). We refer an entity that receives data from a smart home service as a *data user*. We let each data user be logically classified into a *domain*. Existing work [48][49][50][51] models data users with hierarchies, so we follow this convention as well. However, our scope of a data user is larger than that of existing works. Whereas existing works focus only on modeling data users as roles *inside* an enterprise, we choose to model the enterprise *itself* as a single data user, along with other organizations and important individuals, because these

groups are the entities which will create and access services. Thus, our notion of data users is more inclusive than that of other works. We define a domain hierarchy as follows:

Definition 3: A *Domain Hierarchy* is a four-tuple $<D, D', D_H, D'_H>$

1. D is the set of all data users of any smart home.
2. D' is the set of all the domains of the data users.
3. $D_H: D \rightarrow D'$ is a function that maps a data user to its domain.
4. $D'_H: D' \rightarrow D'$ is a function that maps a child domain to one parent domain.

Note that the set of data users of any particular smart home consists of two sets of users. Organizations and enterprises will commonly deploy and use services in multiple smart homes on the basis of a professional relationship with the inhabitant, usually defined by compensation for some service. However, most users that interact with the smart home share some sort of personal relationship with an inhabitant, such as being a family member or a trusted friend. We note that the a single hierarchy can be maintained for the first set of users but that each inhabitant must maintain the portion of the data user hierarchy corresponding to the second set. We plan to further address the management of domain and other hierarchies in future work.

4.3.4 Purposes

As mentioned in Fair Information Practices, purpose refers to the reason or goal of a data access and purpose binding refers to ensuring the purpose of the use of data agrees with the purpose of its collection. Current privacy literature [39][43][48][49][50][51] only applies purpose binding to data; however, since we also desire to regulate accesses to environment objects we must apply a purpose binding to them as well. Since we are using a service-oriented environment and services are placed in smart homes to address some particular need, the purposes of data items and environment objects are defined relative to the goals of service that uses/controls them. Some lower level services may have a single guiding purpose, for example the sensing of a data item, but some higher level services may have multiple purposes, such as the sensing and processing of a data item and the automated control of an environment object. Unfortunately, there is no standard mapping for purpose to a software service. We do note, though, that this purpose-service mapping is dependent upon how *modular* a service is. A cohesive, well-defined service will have fewer purposes than a non-modular service. Also, it is unclear how a method of a service helps accomplish a purpose. Do all methods contribute to a single main service purpose, or do methods themselves correspond to different purposes? In our current model, we assume that services are created modularly with a small number of purposes and that all methods of a service fulfill the service purposes. We plan to investigate purpose relationships between services and methods in future work, and, like existing approaches, model purposes with a hierarchy.

Definition 4: A *Purpose Hierarchy* is a 4-tuple $<P, P', P_H, P'_H>$, where

1. P is the set of specific purposes of smart home services.
2. P' is the set of general purposes of smart home services.
3. P_H: $P \rightarrow P'$ is a function that maps a specific purpose to a general purpose.
4. P'_H: $P' \rightarrow P'$ is a function that maps a child general purpose to a parent general purpose.

4.3.5 Services

Services are the software artifacts placed in the smart home on behalf of some data user(s) to address some sort of need. When they are installed in the smart home, descriptions of their publicly available methods are stored in the Service Registry so they can be searched for and invoked by other services. Since each method may access some set of data items and some set of environment objects, it is service/method invocations that cause data flow among data users and manipulations of environment objects to occur. In order to govern service invocations, we must know what service's purpose is, what data is given as input to the method invoked, what data is returned as output, and what environment objects are accessed by this method. Therefore, we assume that service descriptions are annotated in the following manner.

Definition 5: A *method description m* is a 4-tuple $<mid, T_{IN}, T_{OUT}, O_{CON}>$, where

1. *mid* is an unique identifier, relative to a service, for the method.
2. $T_{IN} \subseteq T$ is the set of input data items to the method.
3. $T_{OUT} \subseteq T$ is the set of data items that are returned by the method.
4. $O_{CON} \subseteq O$ is set of the environment objects that are controlled by the method.

Additionally, we require every service to have at least one data user, and assume that once the service is installed, the data users and purposes for that service remain static.

Definition 6: A *service description s* is a 4-tuple $<sid, D_{BND}, P_{BND}, M>$, where

1. *sid* is a globally unique identifier for this service.
2. $D_{BND} \subseteq D$ is the set of data users bound to this service.
3. $P_{BND} \subseteq P$ is the set of specific purposes bound to the service.
4. M is a set of method descriptions of the service.

Notation 1: We may also want to refer to a component of a service or a method of that service. Given a service description *s* and a method description *m*:

1. *domains(s)* denotes the set of data users of service *s*.
2. *methods(s)* denotes the set of methods of service *s*.
3. *purposes(s)* denotes the set of purposes bound to service *s*.
4. *data_in(s, m)* denotes the set of input data items to method *m* of a service *s*.

5. *data_out (s, m)* denotes the set of output data items of method *m* of a service *s*.
6. *objects(s, m)* denotes the set of environment objects controlled by method *m* of service *s*.

4.4. Privacy Policy Model Definition

In the previous section we explained how services are described in our privacy model. In this section, we investigate how to construct privacy policies that control the flow of data and control the usage of environment objects with respect to the contextual conditions inherent to each. In our model, we use authorization policies that take rulings from the set {*allow, deny*} and have a default ruling from the same set that is used if no rule applies to a request. We first address how the presence of services influences the availability of policy terms and the contexts available for privacy rules. We then describe two privacy policies, a data privacy policy and an object privacy policy, that together specify inhabitant privacy in a smart home. Figure 3 shows the dependencies of terms in policies.

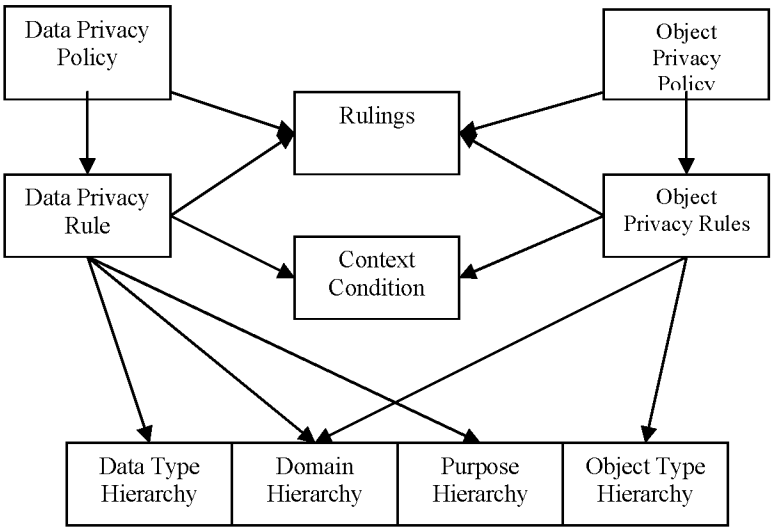

Figure 3 Dependencies of terms and policies.

4.4.1 Availability

Before discussing policy structure, we want to note that the services inside the smart home dictate what data types are in use in the smart home as well as what environment objects are controllable in the home. As more services are added, more data types become available to be used by the system and more environment objects can be controlled. Conversely, when a service is removed, it is possible that a data type or object type may no longer be available to the smart home. This means that the services inside the smart home dictate what policies can be created, and what policy rules are valid for a given smart home. We call this service-term

coupling *availability* and relate several definitions to it. Let S be a set of service descriptions of all services installed in a smart home.

Definition 7: We define the set of *available domains* in the smart home $D_{AV} = \bigcup_{s \in S} domains(s)$.

Definition 8: We define the set of *available purposes* in the smart home $P_{AV} = \bigcup_{s \in S} purposes(s)$.

Definition 9: We define the set of *available data items* in the smart home

$T_{AV} = \bigcup_{s \in S} \bigcup_{m \in methods(s)} data\_out(s,m)$.

Definition 10: We define the set of *available objects* in the smart home

$O_{AV} = \bigcup_{s \in S} \bigcup_{m \in methods(s)} objects(s,m)$.

4.4.2 Contextual Conditions

Both data and object privacy policies contain contextual conditions, which are mechanisms to make rules apply to certain situations. Context conditions are expressed as a finite propositional logic formula of clauses containing conjunctions and disjunctions of predicates over variables representing available data items. The fact that these clauses are constructed over data types has three important implications. First, the system requires there to be a sensing service for the variable in question. Second, the system and inhabitant needs to know the possible values of these quantities so that meaningful conditions can be made. And, lastly, the system needs reliable access to these variables to be able to determine if the context holds at the time of an access request, which is acceptable since the Privacy Manager is a trusted component.

Definition 11: A *contextual clause*, Φ, is defined as a predicate <*x op literal*> where

1. *x* is a variable over some data item in the set of available data items in T_{AV},
2. *op* is a binary operator in the set $\{=, \neq, <, >, \leq, \geq\}$, and
3. *literal* is some constant in the numerical domain of *t*.

Definition 12: A *contextual condition*, $\bar{c}$, a finite propositional formula over contextual clauses, is defined as $\bar{c}$:= Φ | $\bar{c}_1$ AND $\bar{c}_2$ | $\bar{c}_1$ OR $\bar{c}_2$, where $\bar{c}_1$ and $\bar{c}_2$ are also contextual conditions

4.4.3 Data Privacy Rules and Policy

A data privacy policy is a collection of rules that govern the use of privacy sensitive data by allowing inhabitants to authorize who, how, and under what conditions their data is used. The information necessary for this decision to be specified and enforced is given in service descriptions. However, enumerating over a list of

services and creating a set of rules for each individual service would be a very tedious task. Therefore, we specify data privacy policies over the domain, data, and purpose hierarchies. A default ruling is applied to a data request if no policy rules have been explicitly created.

Definition 13: A *data privacy policy*, P_{DATA}, is a 5-tuple $<D_{AV}, P_{AV}, T_{AV}, R_D, dr>$ where

1. D_{AV} is the set of all available domains in the smart home.
2. P_{AV} is the set of all available purposes in the smart home.
3. T_{AV} is the set of all available data types in the smart home.
4. R_D is a non-empty set of data privacy rules.
5. *dr* is a default ruling from the set *{allow, deny}*.

Definition 14: A *data privacy rule* is a 5-tuple $<r, d, t, p, \overline{c}>$ where

1. *r* is a ruling from the set *{allow, deny}*.
2. *d* is an available domain in D_{AV}.
3. *t* is an available data item in T_{AV}.
4. *p* is an available purpose in P_{AV}.
5. $\overline{c}$ is a contextual condition over data items in T_{AV}.

The meaning of a rule $<r, d, t, p, \overline{c}>$ is "ruling *r* is applied when services on behalf of domain *d* attempt to access data type *t* for purpose *p* and the context $\overline{c}$ evaluates to true." A small example of a data privacy rule is "Deny video data to all domains for monitoring purposes when I am in the bathroom". Thus, if any service requests for monitoring video data when the occupant's location is in the bathroom, the request is denied. The aggregate meaning of the rules in a data privacy policy depends on the evaluation algorithm used. We discuss ours in Section 4.4.

4.4.4 Object Privacy Rules and Policy

Similarly to a data privacy policy, an environment object usage policy is a collection of rules is also an authorization policy governing how, why, and under what conditions domains use objects. Rules are written over domain, purpose, and object hierarchies, and a default ruling is also applied to a request if no other object policy rule can be.

Definition 15: We define an *environment object privacy policy*, P_{OBJ}, as a 5-tuiple $<D_{AV}, P_{AV}, O_{AV}, R_O, dr>$ where,

1. D_{AV} is the set of all available domains in the smart home.
2. P_{AV} is the set of all available purposes in the smart home.
3. O_{AV} is the set of all available object types in the smart home.

4. R_O is a non-empty set of data privacy rules.
5. *dr* is a default ruling from the set *{allow, deny}*.

Definition 16: An *object privacy rule* is a 5-tuple $<r, d, p, o, \overline{c}>$ where

1. *r* is a ruling from the set *{ allow, deny}*.
2. *d* is an available domain in D_{AV}.
3. *p* is an available purpose in P_{AV}.
4. *o* is an available object type in O_{AV}.
5. $\overline{c}$ is a contextual condition over data items in T_{AV}.

The meaning of an environment object rule $<r, d, o, p, \overline{c}>$ is "apply ruling *r* when services from domain *d* access object *o* with purpose *p* and the context $\overline{c}$ holds." An example of an object privacy rule is "allow my family members to call me (use my phone) for communication after 8 p.m.". Again, the meaning of an object privacy policy depends on the evaluation algorithm used. Ours is discussed in the next section.

4.5 Request Evaluation

A service request comes in the form of one service method invoking another service method. We call the service that initiated the request the *source service*, the method that initiated the request the *source method*, the service that received the request the *target service*, and the method to be invoked as the *target method*. We similarly define *source* and *target data users, source* and *target purposes,* etc. After receiving an invocation request, the authorization process is divided into four phases: decomposing the service request to atomic requests, determining the various privacy rules that apply to each atomic request, evaluating the contextual conditions of applicable rules, and making a global evaluation decision based on these rules. The overview of this process is shown in figure 4.

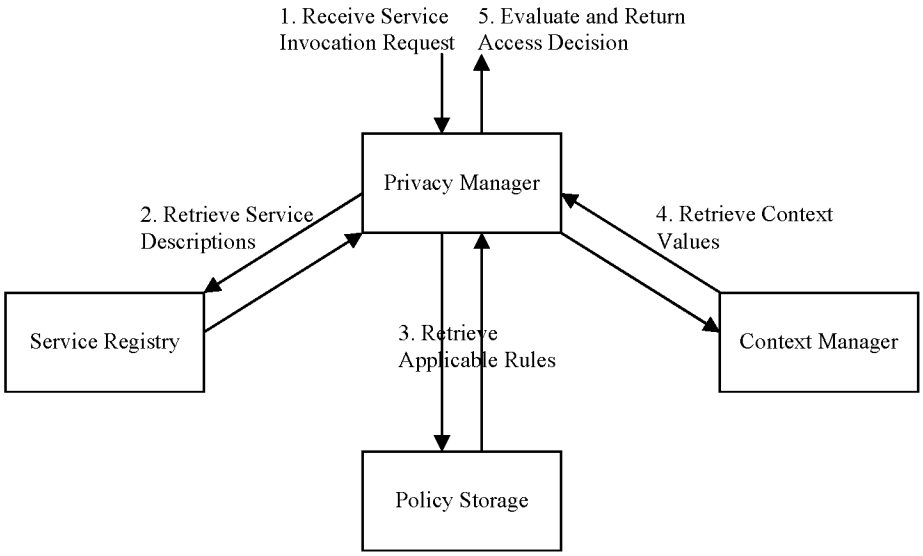

Figure 4 Request evaluation process

4.5.1 Service Request Decomposition

When a service request is received by the Privacy Manager, it contains the source service and method and target service and method.

Definition 17: We define a *service invocation request* as a four-tuple $<sid_1, mid_1, sid_2, mid_2>$ where

1. sid_1 is the source service id
2. mid_1 is the source method id.
3. sid_2 is the target service id.
4. mid_2 is the target method id.

In keeping with our goals of controlling the data flow and restricting object usage, we have four separate checks to make for a given request:

- Check #1 (Data-to-Target): Are the data types in the target method arguments allowed to flow to the target service's domains under the current system context?

- Check #2 (Data-to-Source): Are the data types in the return parameter of the target service allowed to flow to the source service's domains under the current system context?

- Check #3 (Object-by-Target): Are the object types manipulated by the target method allowed to be controlled by the target service's domains under the current system context?

- Check #4 (Object-by-Source): Are the object types manipulated by the target method allowed to be controlled by the source service's domains under the current system context?

Note that the objects controlled directly by the source service are excluded from this evaluation process. This is because for the source method to be executing, it must have passed a previous evaluation. First, the Privacy Manager contacts the Service Registry to obtain the descriptions for sid_1 and sid_2. Then the Privacy Manager decomposes the descriptions into data access requests, corresponding to Check #1 and Check #2, and object access requests, corresponding to Check #3 and Check #4. Since the service description notation uses sets that may contain multiple elements of each term, we call these access requests a *request set.* An element of a request set has only one term of each type and is called an *atomic request.* From a service invocation request, $<sid_1, mid_1, sid_2, mid_2>$ and associated service/method descriptions $<s_1, m_1>$, and $<s_2, m_2>$ we formally decompose to four request sets and four types of atomic requests.

Definition 18: The *Data-to-Target request set,* RS_{DT}, corresponding to check #1, authorizes the data flow to the target service, so its request set contains domains and purposes of the target service as well as the set of input data types of the target method. $RS_{DT} = \overline{D} \times \overline{P} \times \overline{T}$, where

1. $\overline{D} = domains(s_2)$ is the set of data users of the target service
2. $\overline{P} = purposes(s_2)$ is the set of purposes of the target service
3. $\overline{T} = data\_in(s_2, m_2)$ is the set of input data items to the target method.

Definition 19: The *Data-to-Source request set,* RS_{DS}, corresponding to check #2, authorizes data flow back to the source service, so its request set requires the domains and purposes of the source service as well as the set of data types returned by the target method. $RS_{DS} = \overline{D} \times \overline{P} \times \overline{T}$ where

1. $\overline{D} = domains(s_1)$ in this case is the set of data users of the source service.
2. $\overline{P} = purposes(s_1)$ in this case is the set of bound purposes of the source service.
3. $\overline{T} = data\_out(s_2, m_2)$ in this case is the set of output data items of target method.

Definition 20: An atomic data request is therefore a triple $<d, p, t>$ where, depending on the request set it was derived from,

1. $d \in \overline{D}$ is either a single data user of the target service or a single data user of the source service.

2. $p \in \overline{P}$ is either a single purpose bound to the target service or a single purpose bound to the target service.
3. $t \in \overline{T}$ is either a single input data item of the target method or single output data item of the target method.

Definition 21: The *Object-by-Target request set, RS_{OT}*, corresponding to check #3, authorizes object control by the target service, so its request set contains domains and purposes of the target service as well as the set of controllable objects of the target method. $RS_{OT} = \overline{D} \, x \, \overline{P} \, x \, \overline{O}$ where

1. $\overline{D}$ = *domains(s_2)* is the set of data users of the target service.
2. $\overline{P}$ = *purposes(s_2)* is the set of purposes bound to the target service.
3. $\overline{O}$ = *objects(s_2, m_2)* is the set of objects controlled by the target method.

Definition 22: The *Object-by-Source request set*, *RS_{OS}* authorizes object control by the source service, so its request set requires the domains and purposes of the source service as well the set of controllable objects returned by the target method. $RS_{OS} = \overline{D} \, x \, \overline{P} \, x \, \overline{O}$ where

1. $\overline{D}$ = *domains(s_1)* is the set of data users of the source service.
2. $\overline{P}$ = *purposes(s_1)* is the set of purposes bound to the source service.
3. $\overline{O}$ = *objects(s_2, m_2)* is the set of objects controlled by the target method.

Definition 23: An *atomic object request* is a triple *<d, p, o>* where, depending on the request set it was derived from,

1. $d \in \overline{D}$ is a single data user of the target service or a single data user of the source service.
2. $p \in \overline{P}$ is a single purpose bound to the target method or a single purpose bound to the source method.
3. $o \in \overline{O}$ is a single object invoked by the target method.

4.5.2 Determining Applicable Rules

After the service request is decomposed into the four sets of atomic requests, the Privacy Manager must retrieve the privacy rules that apply to each atomic request. A privacy rule *applies* to an atomic request if all the terms in the rule are hierarchically equivalent to or ancestors of the corresponding terms in the atomic request. That is, an atomic data request <d, p, t> has the applicable rules which are specified with d or an ancestor domain of d, p or an ancestor purpose of p, and t or an ancestor data type of t. Similarly, for an atomic object request <d, p, o>, the applicable rules set are the rules in the object policy that are specified with d or an ancestor domain, p or an ancestor purpose of p, and o or an ancestor object type of o.

Notation 2: For a given hierarchy H, and an element $h \in H$, the set of ancestors of h denoted *ancestors(H, h)* = { h' | h' is an element on the path from the root of H to h, inclusive}.

Definition 24: For a given atomic data request $rq = <d, p, t>$ and a data policy P_{DATA} we define the *applicable rule set* for $<d, p, t>$ in P_{DATA}, denoted $AR(<d, p, t>, P_{DATA})$, as { $<r, d', p', t', \bar{c}>$ | ($<r, d', p', t', \bar{c}> \in P_{DATA}$) $\wedge$ ($d' \in ancestors(d)$) $\wedge$ ($p' \in ancestors(p)$) $\wedge$ ($t' \in ancestors(t)$) }.

Definition 25: Similarly, for a given atomic object request $rq = <d, p, o>$ and corresponding data policy P_{OBJ} we define the *applicable rule set* for rq in P, denoted $AR(rq, P_{OBJ})$ as { $<r, d', p', o', \bar{c}>$ | ($<r, d', p', o', \bar{c}> \in P_{DATA}$) $\wedge$ ($d' \in ancestors(d)$) $\wedge$ ($p' \in ancestors(p)$) $\wedge$ ($o' \in ancestors(o)$)}.

4.5.3 Evaluating Contexts

After the Privacy Manager retrieves the applicable rules for each atomic request, it must determine which of these rules have contexts that are enabled in the system. To do this, the Privacy Manager must make requests to the Context Manager for current values of all the context variables contained in the conditions of the applicable rules. If when the quantities are received, the context condition evaluates to true, the rule is considered contextually enabled. Otherwise, the clause evaluates to false and the rule is considered contextually disabled.

Definition 26: Let r be a privacy rule, and let $\bar{c}$ be a contextual condition for rule r.
Also, let $assign(\bar{c})$ be an assignment to the context variables retrieved for $\bar{c}$ from the Context Manager. Then,

1. $Enabled(r) \leftrightarrow (assign(\bar{c}) = true)$
2. $Disabled(r) \leftrightarrow (!Enabled(r))$

Definition 27: Let rq be an atomic request and P its corresponding policy, the *enabled rule set*, denoted $ENABLED(rq, P)$ is the subset of applicable rules for rq in P which have contextual conditions that are evaluated to true, and the *disabled rule set*, denoted $DISABLED(rq, P)$, is the subset of applicable rules for rq in P which have contextual conditions that are evaluated to false. Then,

1. $ENABLED(rq, P) = \{r \mid r \in AR(rq, P) \wedge Enabled(r)\}$
2. $DISABLED(rq, P) = \{r \mid r \in AR(rq, P) \wedge Disabled(r)\}$

4.5.4 Making a Decision

After determining the enabled and disabled rule sets for atomic requests, the Privacy Manager must map the information to a global decision. First, it evaluates each atomic request.

Definition 28: Let *rq* be an atomic request and *P* be the corresponding policy, an *atomic authorization decision* is a function

Γ_{ATOMIC}: <*rq, ENABLED(rq, P), DISABLED(rq, P)*> → {allow, deny}.

Next, the Privacy Manager collects all the rulings for atomic requests and evaluates them to reach a global decision.

Definition 29: Let {allow, deny}^{n} be the set of local rulings returned from evaluating all atomic requests, where n = $|RS_{DT}|$+ $|RS_{DS}|$ + $|RS_{OT}|$ + $|RS_{OS}|$. Then the *global authorization decision* is a function

Γ_{GLOBAL}: {allow, deny}^{n} → {allow, deny}.

In our Privacy Manager, the authorization semantics we employ is that the allow decision implies that all applicable allow rules for all atomic requests are enabled and all applicable deny rules for all atomic requests are disabled. That means a deny decision implies either some applicable allow rule for some atomic request is not enabled or some applicable deny rule for some atomic request is enabled. A default ruling is only used when evaluating an atomic request with no applicable rules or applicable rules with only disabled deny rules.

Other semantics we could have chosen could be the first-available rule method, which makes a decision based on the ruling of the first enabled rule. However, the meaning of a policy would then depend on the order the applicable rules are written, sorted, or contextually evaluated. On the other hand, our semantics addresses the presence of multiple applicable rules by essentially conjoining all context conditions. This ensures that any sort of rule ordering does not change the meaning of the policy. Figure 5 shows our algorithm for evaluating a service invocation.

```
EVAL_ATOMIC (Request r, Policy p)
{
      //Determine applicable rules (obtain from Policy Store)
       1. Let AR = set of rules applicable to r in P

      //Obtain and evaluate contexts from Context Manager
       …

      //Allow if all allow applicable rules are enabled and all
      //applicable deny rulings are disabled
       2. If ((FORALL(i in AR)
                ((ruling[i]  == "ALLOW") AND enabled(i)) OR
                ((ruling[i] == "DENY") AND !enabled(i))))
                    return ALLOW

      //Deny if there is an allow rule disabled or a deny rule enabled
       3. Else if ((EXISTS(i in AR) such that
                ((ruling[i]  == "ALLOW") AND !enabled(i)) OR
                ((ruling[i] == "DENY") AND enabled(i)))
                    return DENY

      //Otherwise, apply the default ruling (which is ALLOW or DENY)
       4. Else if ((AR is empty) OR
              (FORALL(i in AR) (ruling[i] == "DENY" AND !enabled(i))))
                    return DEFAULT_RULING from P
 }

EVAL_SERVICE_INVOCATION(ServiceInvocation s, DataPolicy dp, ObjectPolicy op)
{
      // Resolve service/method descriptions (from Service Registry)
        …

      //Store local evaluations in a set
      1.  Let Results = {}
      2.  Let RS_DT = set of atomic requests for Check 1 (Data-to-Target)
      3.  Let RS_DS = set of atomic requests for Check 2 (Data-to-Source)
      4.  Let RS_OT = set of atomic requests for Check 3 (Object-by-Target)
      5.  Let RS_OS = set of atomic requests for Check 4 (Object-by-Source)

      //Evaluate each atomic request and collect results
      6.  FORALL(atomic request r in RS_DT)
                Results = Results ∪ EVAL_ATOMIC(r, dp)
      7.  FORALL(atomic request r in RS_DS)
                Results = Results ∪ EVAL_ATOMIC(r, dp);
      8.  FORALL(atomic request r in RS_OT)
                Results = Results ∪ EVAL_ATOMIC(r, op);
      9.  FORALL(atomic request r in RS_OS)
                Results = Results ∪ EVAL_ATOMIC(r, op);

     //Deny if any atomic request is denied
      10. If DENY∈Results
                Return DENY
      11. Else Return ALLOW
 }
```

Figure 5 Evaluation procedure for a service request invocation

4.6 Possible Policy Conflicts

As with most policy-based systems, it is possible to have conflicts within or between policies. Conflicts may arise from a specification error and introduce inconsistencies or inadvertent side effects on system operations. Therefore conflicts need to be detected and resolved. We have identified several types of possible conflicts that may be present in our policy model: rule unsatisfiability, modality conflicts at the atomic request level, modality conflicts at the request set level, and modality conflicts at the service invocation level. We now discuss how these conflicts manifest themselves and their effects on policy evaluation.

4.6.1 Conflict #1: Unsatisfiability of Rule Contexts

Remember that rule contexts are clauses of predicates over various properties of the inhabitant or environment intended to restrict the manner in which a privacy rule is applied. It is conceivable that a context could be erroneously specified in such a way that it can never be evaluated to true. For example, the condition ((day = Friday) AND (day = Saturday)) is not a satisfiable context. If the context can never be satisfied, then the associated privacy rule can never be enabled. Such a rule not only misleads an unaware policy author, but also creates processing overhead because it may be returned in queries and cause context variables to be retrieved and expressions evaluated. Also, using our evaluation semantics, if the unsatisfiable rule is an allow rule, it could result in the unwarranted denial of any request that matches the rule.

4.6.2 Conflict #2: Modality Conflicts in an Atomic Request

Modality conflicts as explained earlier arise from having the same subjects, targets, and actions in the same policy with different rulings. However, in our system these conflicts are a slightly different because we include purpose, contextual conditions, and hierarchies of terms. Given two rules in the same policy that apply to an atomic request, a modality conflict is possible if three conditions are met. First, the two rules must have opposing rulings. Second, the rules must be applicable at the same time, which only occurs if all corresponding basic terms (i.e. not rulings or contexts) have a common descendant leaf node in their respective hierarchies. Third, the context of one rule must be a subset of the context of the other. Whenever the “smaller” context is enabled the “larger” one will be enabled as well causing these rules to always cancel each other out. This type of conflict thus results in the perpetual denial of the atomic request, as well as any service invocation that contains that request, when the stricter context is satisfied.

4.6.3 Conflict #3: Modality Conflicts in a Request Set

Similarly, service descriptions can contain multiple data users, purposes, data types, or objects. When this happens, service invocations are decomposed into request sets which consist of more than one atomic request. It may be possible that, within the scope of this request set, a rule applied to one atomic request is always evaluated at the same time as an opposing rule in another atomic request. The conditions for this type of modality conflict are slightly different than before. Again, the rulings must be opposite, and the context of one

rule must be a subset of a context of the other rule. However, instead of the rules having to be applicable to the *same* atomic request, they can be applicable to *any* atomic request generated from the same request set. Whereas modality conflicts at the atomic request level can be detected by comparing all rules within a single policy, this type of conflict depends upon relationships between a policy, the available service descriptions, and the set of possible service invocations. What the exact nature of this relationship is will be explored in future work.

4.6.4 Conflict #4: Modality Conflicts between Request Sets

Lastly, because service invocations are decomposed into multiple request sets, it may also be possible for rules applicable to different request sets to create modality conflicts. This type of modality conflict is more interesting because it involves rules spanning both data and object policies. If a service description contains both data types and object types, its subsequent evaluation requires retrieving both data and object privacy rules. It may therefore be possible that some data privacy rule and some object privacy rule produce a modality conflict for a given service invocation. Like the previous type of conflict, this happens when rules are opposing, and contexts share the subset relationship as described above. Now, however, the relationship determining this type of conflict generalizes to include both data and object policies as well as the set of service descriptions and possible invocations. Again, defining this relationship is left to future work.

5. MODEL CHECKING PRIVACY POLICIES AND THEIR EVALUATION

In this section, we explain how we use the SPIN model checker [32][58] to specify and verify a preliminary version of the privacy policy model and request semantics described in the previous section. SPIN is a Linux-based verification tool capable of the simulation and verification of software systems that we chose to use because of its process-centric syntax and our previous experience using the tool. Systems are specified using a symbolic language called Promela (the Process Meta Language). A system in Promela is treated as a collection of processes, where each process contains a sequence of deterministic and non-deterministic statements. Variables may be defined locally within a process or globally for all processes, and inter-process communication occurs either via some global variable or via message passing using shared message channels. Properties are specified using a Linear Temporal Logic (LTL) formula, which is further discussed in section 5.4.1. During a verification run, the property is also translated to a state machine as well and executed in lockstep with other system processes. First, the property executes to check the initial system state. Then some non-deterministically chosen process executes a step, which may itself be either deterministic or non-deterministic. Then the property steps again, etc. If the property is ever violated, SPIN produces an error trail that can be played back in a simulation for analysis. We first introduce an example scenario concerning video privacy, derive the necessary model information (terms, services, and policies), and finish with overviews of the Promela implementation and LTL property specifications. Model source files are contained in appendix A, and property source files are contained in appendix B. We assume that the meaning of SPIN syntax in our code examples is self-explanatory, but [58] contains further information if needed.

5.1. Example Scenario

Consider an elderly individual named Bob who has a balance impairment and lives alone at home. With Bob's consent, his children paid a company to install and supervise a video monitoring system to be able to periodically check that he has not fallen. Bob's house has four rooms, the living room, kitchen, bedroom, and bathroom, and each room has a video camera in it. Using a regular web browser, authorized employees of the company and authorized family members can use video to monitor Bob. The installed system consists of six services. A location tracking service provides Bob's current room. A time service provides the current time of the system in hours. A video service provides a method to retrieve video from a given room. A camera actuator service provides a method to pan or tilt a camera in a given room. Lastly, there are two high level monitoring services, one for the company and one for his family. These services each provide a method to view Bob's current location. This is done by periodically querying the location service to know where Bob is to be able to select an appropriate camera. Then, they repeatedly invoke the first camera service to retrieve the video of Bob's current location. This video is pre-processed using the system time to track Bob's movement. Using the second camera service, the monitor services automatically adjust the camera in Bob's current room the so Bob

always stays in frame. Thus, the monitor services invoke all other services. The possible invocations of the system are shown in below in figure 6.

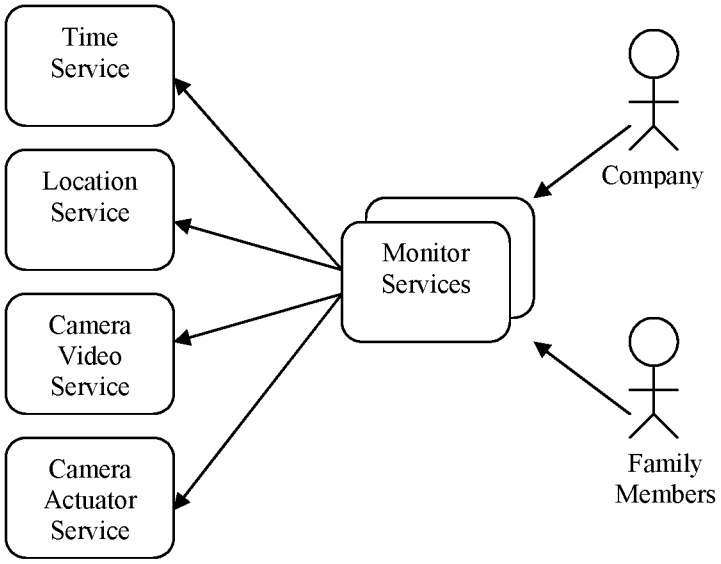

Figure 6 Possible invocations in the example scenario

Though Bob is glad to have someone observe if he falls, he is not comfortable with idea of being watched while he in the bathroom. He also does not want to be watched while he changes his clothes in the bedroom, which happens in the morning between 6 a.m. – 7 a.m. and in the evening between 9 p.m. – 10 p.m. Luckily, the monitoring system is privacy enabled using our privacy model, so Bob has created the following rules to govern the operation of the video monitoring system:

1. Anyone can see his location at any time to for any reason.
2. Video may not be seen nor the camera moved by anyone for any purpose while he is in the bathroom.
3. Video may not be seen nor the camera moved by anyone for any purpose while he is in the bedroom between the hours of 6:00-7:00 a.m. or 9:00-10:00 p.m.
4. Otherwise, everything else is allowed.

5.2. Model Entities

We now derive the terms, services descriptions, and policy specifications for our example scenario.

5.2.1 Terms and Hierarchies

There are two obvious data users in our scenario. Family members and the company can both view video of Bob. However, the time, location, and camera services are not operated by family members or company employees. We therefore assign to them to a default system domain. All of the services in this scenario can be grouped into one of three purposes: providing information, moving a camera, or monitoring Bob, so we label the three purposes in our system are sense, move, and monitor. Location, time, and video are the data types available in the system, and lastly, the only environment objects controlled under control by the system are the cameras. The following table summarizes the terms derived in this example. Note that since location, time, and video are the only data items in the system, they are also the only possible context variables.

Terms	Values
Domains	{System, Family, Company}
Purposes	{Sense, Move, Monitor}
Data Types	{Location, Video, Time}
Object Types	{Camera}

Table 1 Basic terms derived from the example scenario

We create simple hierarchies for these terms by letting each term in table 1 be a leaf in its respective hierarchy and adding a universal parent node for each tree. The resulting hierarchies are shown in figure 7.

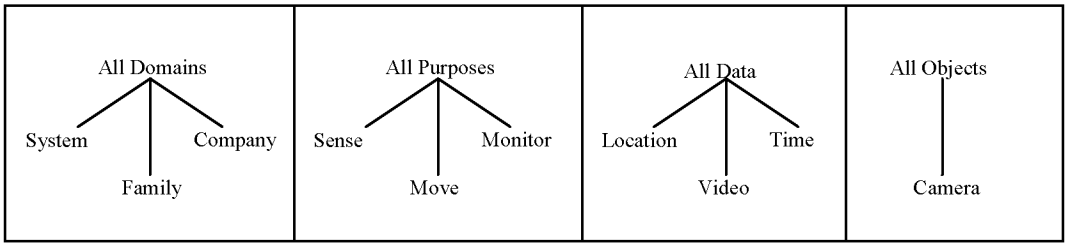

Figure 7 Hierarchies derived from the example scenario

5.2.2 Service Descriptions

In discerning service descriptions for our example scenario, we observe that each service need only contain one method. The time, location, and video service can only be invoked to provide their respective information. The camera actuator service can only be invoked to move a camera. Both camera services require the room of the target camera in order to internally access the appropriate camera. When either monitor service's method is invoked, it makes corresponding method calls to the camera and location services to obtain

information and releases video information as output. The following table summarizes the combined service-method descriptions that entail our smart home space. An "X" indicates the absence of a particular term in a service description.

Service	Domain	Purpose	In Data Type	Out Data Type	Object
Camera Video Service	System	Sense	Location	Video	X
Camera Actuator Service	System	Move	Location	X	Camera
Time Service/Hour	System	Sense	X	Time/hour	X
Location Service	System	Sense	X	Location/Room	X
Company Monitor Service	Company	Monitor	X	Video	X
Family Monitor Service	Family	Monitor	X	Video	X

Table 2 Service descriptions derived from the example scenario

5.2.3 Privacy Policies

Bob's set of constraints on the operation of services was colloquially specified in three rules, with a fourth specifying default behavior. The first three of Bob's rules concern the flow of information, but only the second two refer to controlling objects. Thus, there are three data privacy rules and two object privacy rules. The results are shown in tables 3 and 4, respectively. In both tables, the left column displays Bob's original rule, and the right column contains the policy rule as needed by the system. An allow ruling is shown with a plus sign, and a deny ruling is shown with a minus sign, and an "X" again implies the absence of needing a rule. Context conditions are written with an semi-formal notation.

Original Privacy Rule	Derived Data Privacy Rules <r, d, p, t, c>
Rule #1: Allow location data….	<+,all domains, all purposes, location, true>
Rule #2: Deny video…bathroom.	<-, all domains, sense, video, (location == bathroom)
Rule #3: Deny video…changes clothes.	<-, all domains, sense, video, ((location == bedroom) AND ((6 a.m. < time < 7 a.m.) OR (9 p.m. < time < 10 p.m.)))

Table 3 Data privacy rules derived from the example scenario

Original Privacy Rule	Derived Object Privacy Rule <r, d, p, o, c>
Rule #1: Allow location data….	X
Rule #2: Deny control of camera…bathroom.	<-, company, move, camera, (location == bathroom)
Rule #3: Deny control of camera…changes clothes.	<-, company, move, camera, ((location == bedroom) AND ((6 a.m. < time < 7 a.m.) OR (9 p.m. < time < 10 p.m.)))

Table 4 Object privacy rules derived from the example scenario

5.3 Model Implementation

Though our logical privacy model contains separate entities for the Service Registry, Context Manager, the Policy Store, and the Privacy Manager, more processes in a system results in higher complexity in the model. So, for the sake of optimizing our model description and policy evaluation, we reorganize and serialize the behavior of our architectural components whenever possible. The result is a system that contains two processes: a modified Privacy Manager and a Context Generator. The modified Privacy Manager assimilates the roles of the Service Registry, the Policy Store, the Context Manager, and Privacy Manager by storing service descriptions, storing privacy policies, and retrieving context information, generating service invocation requests, and applying authorization logic. The Context Generator controls the dynamic system behavior that causes contextual factors to change. Each data type is assumed to have a pre-defined range, and the Context Generator simulates changes in system context by randomly assigning a value to each context variables from within its range. We do not include states or transitions of either environment objects or services in the model implementation because they are irrelevant to the correctness properties we are trying to verify. Whenever possible, statements in a process are grouped inside `atomic` or `d_step` constructs in Promela. Statements inside these constructs are executed without being interleaved with other processes, which limits possible executions and thus reduces the state space of the model. The full source Promela source code files for global data structures, the Privacy Manager process, and the Context Generator process are in Appendices A.1, A.2, and A.3, respectively.

5.3.1 Terms and Hierarchies

Hierarchies are specified in two parts. First, symbolic values for all nodes are created. This is done using the Promela "mtype" declaration, which maps a string literal to an integer value from 1 – 255. SPIN concatenates all mtype declarations into a single system-defined enumeration, so there can be at most 255 different mtype elements. The value 0 is reserved as a special NULL constant.

```
mtype = {ALL_DATA, LOCATION, TIME, VIDEO};      /* data items & types */
mtype = {ALL_DOMAINS, SYSTEM, FAMILY, COMPANY}; /* data users & domains */
mtype = {ALL_OBJECTS, CAMERA};                  /* objects & object types*/
mtype = {ALL_PURPOSES, MONITOR, SENSE, MOVE};   /* purposes */
```

Next, we define a data structure to represent the hierarchies. Each hierarchy is viewed as a collection of leaf nodes, with each leaf node storing the values of all its ancestors. Since Promela does not allow dynamic memory allocation, this requires $O(n*lgn)$ space for a hierarchy of n nodes (because at most n/2 leaves store at most lg n ancestors). Another option would have been to treat a hierarchy as a collection of internal nodes with each node storing the values all the leaves under it. However, this would have produced a $O(n^2)$ complexity (as at most n/2 nodes store at most n/2 leaves). Clearly, the first option scales better. In our scenario, there are at most three leaves in any hierarchy, and the maximum depth of any node is one. Thus, each leaf needs only to store the value of one ancestor. The constants `MAX_DEPTH` and `MAX_LEAVES` represent these values. For making

property specification easier to read, we group the four hierarchies together into an array called `terms`. The constants `DOMAIN_INDEX, PURPOSE_INDEX, DATA_INDEX`, and `OBJECT_INDEX` define the offsets of their respective hierarchies.

```
#define MAX_DEPTH 1
#define MAX_LEAVES 3

/* A leaf stores a list of its ancestors */
typedef leaf{
      byte value;
      byte ancestors[MAX_DEPTH];
};

/* A hierarchy is a collection of leaves */
typedef hierarchy{
      leaf leaves[MAX_LEAVES];
};

/* The 4 hierarchies form the set of terms */
typedef terms_t{
      hierararchy h[4];
};

/* Indices of hierarchies */
#define DOMAIN_INDEX 0
#define PURPOSE_INDEX 1
#define DATA_INDEX 2
#define OBJECT_INDEX 3
```

5.3.2 Services

Additionally, we must define a data structure for service definitions. As mentioned previously, the service definition we use in the model is slightly simplified from the definition given in section 4. Namely, since each service has only one method and there is at most one of any service component, we create a data structure in which ach service consists of up to one domain, one purpose, one in data item, one out data item, and one object. In fact, since Promela data structures are allocated statically, we always have exactly four atomic requests to check. If a service description does not contain a data in, data out, or object parameter, then the corresponding vacancy in the data structure is given the Promela NULL value. This is acceptable because all terms have a non-zero value.

```
/*----------------------*/
/* Service descriptions */
/*----------------------*/

typedef service{
      mtype data_in;
      mtype data_out;
      mtype object;
      mtype domain;
      mtype purpose;
};
```

```
service services[NUM_SERVICES];
```

5.3.3 Privacy Policies

We implement a data privacy policy as an array of rules, where each rule has a ruling, a data type, a domain, a purpose, and a context condition. Similarly, an object privacy policy is an array of rules, where each rule has a ruling, object, domain, and purpose and a context condition. Valid rulings are `ALLOW` and `DENY`, and there is a default ruling for each policy: `DATA_DEFAULT` for the data policy and `OBJ_DEFAULT` for the object policy. Both of these default rulings are also either `ALLOW` or `DENY`. Data user, purpose, data item, and object are all mtypes, and because Promela does have the expressiveness to store a condition expression directly, contexts are implemented as globally defined macros. The macros are evaluated at run-time and their value is assigned to the `context` field in the corresponding rule. The three data rule contexts are `#define dc0` through `#define dc2`, and the two object rule contexts are `#define oc0` and `#define oc1`.

```
/* Constants for rulings */
#define NIL 0
#define ALLOW 1
#define DENY 2

/* Global decision variable */
byte RULING;

/* Default policy rulings */
#define DATA_DEFAULT ALLOW
#define OBJ_DEFAULT ALLOW

/*---------------------*/
/* Data Privacy Policy */
/*---------------------*/

typedef data_rule{
       byte ruling;
       mtype data;
       mtype domain;
       mtype purpose;
       bit context;
};

data_rule data_policy[NUM_D_RULES];

/* DATA RULE 0 */
#define dc0 true

/* DATA RULE 1 */
#define dc1 (room == bathroom)

/* DATA RULE 2 */
#define dc2 ((room == bedroom) && (((hour >= 6) && (hour <= 7)) ||
             ((hour >= 21) && (hour <= 22))))

/*-----------------------*/
/* Object Privacy Policy */
/*-----------------------*/
```

```
typedef obj_rule{
       byte ruling;
       mtype object;
       mtype domain;
       mtype purpose;
       bit context;
};

obj_rule obj_policy[NUM_O_RULES];

/* OBJECT RULE 0 */
#define oc0 (room == bathroom)

 /* OBJECT RULE 1 */
 #define oc1 ((room == bedroom) && (((hour >= 6) && (hour <= 7)) || \
                        ((hour >= 21) && (hour <= 22))))
```

5.3.4 Context Variables

Context variables are global variables so the Context Generator can assign to them and the Privacy Manager can retrieve their values without worrying about the semantics and complexity of communication channels. In our example scenario we have two context variables associated with the context conditions of privacy rules.

```
/* Environment context variables */
byte hour /* values from 0 to 23 */

/* Occupant context variables */
mtype = {living, bath, bed, kitchen};
byte room;
```

5.3.4 The Privacy Manager Process

The Privacy Manager implementation executes in four phases. The first phase initializes the system by setting the privacy policy rules and service descriptions to the appropriate values as decided by our example scenario. Again, if some field of a method or service is not present, then that field is assigned a null value. This phase occurs only once at the beginning of the execution. The last three phases occur in a loop until the model has been verified. Earlier, we described an algorithm that received a request and then contacted separate components to retrieve service descriptions, retrieve applicable rules, and context values. We optimize the model's performance by reorganizing this algorithm slightly. Our model implementation first non-deterministically generates a service invocation request by generating an allowable request from indices from the `services` array. Then, with global access to context variables, the Privacy Manager evaluates the all the contextual conditions for both policies by assigning the truth value of their macros to the appropriate rule contexts.

```
/* "Evaluate" data rule contexts */
data_policy[0].context = dc0;
data_policy[1].context = dc1;
data_policy[2].context = dc2;
```

```
/* "Evaluate" object rule contexts */
object_policy[0].context = oc0;
object_policy[1].context = oc1;
```

The Privacy Manager then generates the four atomic requests from the descriptions in the `service` array and iterates over all policy rules to determine the appropriate rules for each atomic request. This is done sequentially in two loops. One loop iterates through the data policy and checks for applicability to the atomic data requests (check #1 and check #2). The other loop iterates through the object policy to check for rules applicable to the atomic object requests (check #3 and check #4). Intermediate rulings are stored incrementally by keeping a separate Boolean tally for allow and deny decisions for each request.

```
/* rq1 - data-to-target request */
/* rq2 - data-to-source request */
/* rq3 - object-by-target request */
/* rq4 - object-by-source request */
bool allow_rq1, deny_rq1, allow_rq2, deny_rq2;
bool allow_rq3, deny_rq3, allow_rq4, deny_rq4;
```

If an atomic request contains a null term (either a null data item or null object), the applicability testing is skipped (via the `unless` construct in Promela) and the request is allowed. Otherwise, applicability of a rule to an atomic request is determined by checking if the terms in the rule are ancestors of corresponding terms in the request. This hierarchical relationship is tested with the following macros:

```
/* Stores the index of the leaf with value x in hierarchy of index z in i
and 255 if x is not in z */
INDEX(x, z, i);

/* 'res' is true if value y is in the ancestor list of leaf index x in
hierarchy with index z, and false otherwise */
IS_ANCESTOR(x, y, z, res);
```

The evaluation logic concatenates each rule evaluation on all previous rules' evaluations. The evaluation logic for the Data-to-Target check is shown below. First, the current rule is obtained. Then, if the data item in the request is null `allow_rq1` is set to 1 (by the `unless` clause). Otherwise, variables `test1`, `test2`, and `test3` hold the results of the ancestor checking. Next, the rule is checked for its ruling and its enabled status. If it is an enabled allow, then the `allow_rq1` variable is set to 1, otherwise it retains its previous value. Similarly, if the rule is a disabled allow or an enabled deny `deny_rq1` is set to 1, otherwise it retains its previous value.

```
/* Data-to-Target Check … */
do
:: (index < NUM_D_RULES) ->

       { /* Start of unless statement */
```

```
                /* Get current data rule */
                drl.ruling = data_policy[index].ruling;
                drl.data = data_policy[index].data;
                drl.domain = data_policy[index].domain;
                drl.purpose = data_policy[index].purpose;

                /* Reset ancestor test variables */
                test1 = 0; test2 = 0; test3 = 0;

                /* Does current rule apply to target domain? */
                INDEX(services[tgt].domain, DOMAIN_INDEX, leaf_index);
                IS_ANCESTOR(leaf_index, drl.domain, DOMAIN_INDEX, test1);

                /* Does current rule apply to target purpose? */
                INDEX(services[tgt].purpose, PURPOSE_INDEX, leaf_index);
                IS_ANCESTOR(leaf_index, drl.purpose, PURPOSE_INDEX, test2);

                /* Does current rule apply to target in data? */
                INDEX(services[tgt].data_in, DATA_INDEX, leaf_index);
                IS_ANCESTOR(leaf_index, drl.data, DATA_INDEX, test3);

                /* Set value for allowed request (enabled ALLOW) */
                allow_rq1 = allow_rq1 ||
                       ((drl.ruling == ALLOW) &&
                       test1 && test2 && test3 &&
                       data_policy[index].context);

                /* Set value for a denied request (disabled ALLOW) */
                deny_rq1 = deny_rq1 ||
                       ((drl.ruling == ALLOW) &&
                       test1 && test2 && test3 &&
                       !data_policy[index].context);

                /* Set value for a denied request (enabled DENY) */
                deny_rq1 = deny_rq1 ||
                       ((drl.ruling == DENY) &&
                       test1 && test2 && test3 &&
                       data_policy[index].context);
            }
             unless
            {
               /* Non-existent request */
               services[tgt].data_in == 0;
               allow_rq1 = 1;
            }

            /*-------------------------------------------------*/
            /*                                                 */
            /* Data-to-Source Check performed here, but omitted */
            /* for space considerations                        */
            /*                                                 */
            /*-------------------------------------------------*/

            index++;

        /* finished with policy */
        :: else -> break;

        od;
}
```

The other data request check is performed where alluded to in the code. The object request checks are performed in a similar manner in a separate loop. Then, after all request sets have been checked, each pair of allow/deny variables is examined to see if the default ruling should be applied to that request, which is necessary if both variables are zero (i.e. no policy rule was appied). The logic for only one pair of variables is shown below. Then, the global decision is reached and assigned to the global variable `RULING` as is expressed in the following code fragment. (NOTE: the expression (x -> y : z) is a conditional expression in Promela. If x is true, y happens, otherwise z happens.)

```
/* Apply default ruling to data request if needed */
allow_rq1 = ((!allow_rq1 && !deny_rq1) ->
        ((DATA_DEFAULT == ALLOW) -> 1 : 0) : allow_rq1);
deny_rq1 = ((!allow_rq1 && !deny_rq1) ->
        ((DATA_DEFAULT == ALLOW) -> 0 : 1) : deny_rq1);

/*-----------------------------------------------------------*/
/*                                                           */
/* More default ruling code here for allow_rq2 - deny_rq4 */
/*                                                           */
/*-----------------------------------------------------------*/

/* Global decision
            -allow if all atomic requests were allowed and none
             were denied.
            -deny if any atomic request was denied */

RULING = ((allow_rq1 && allow_rq2 && allow_rq3 && allow_rq4 &&
        !deny_rq1 && !deny_rq2 && !deny_rq3 && !deny_rq4) ->
        ALLOW : DENY);
```

After the decision is made, the procedure starts over again. A new request is generated, contexts are re-evaluated, and authorization logic is applied. Each iteration of the main loop is constrained to happen atomically to minimize the transition space of possible executions. It also ensures that contexts do not change during the evaluation process.

5.3.5 The Context Generator Process

The Context Generator implementation simulates dynamic system behavior by non-deterministically assigning to context variables. There are only two context variables used in Bob's rules: time and location. The passage of time is simulated by incrementing the variable `hour` a random number of times. Occupant movement is simulated by randomly assigning to the variable `room` from its domain of `{living, kitchen, bath, bed}`. This process is implemented in such a way that at least one context variable changes with every execution of this process. Both the Context Generator and the Privacy Manager are infinite loops so that all possible combinations of contexts and service invocations are checked by the verifier.

5.4 Property Specification

We have previously described some conflicts that can occur in our policy model and that these conflicts can represent errors in policy specification. We now explain how we verify the absence of these conflicts our privacy model. We also explain how we verify the correctness of our authorization semantics and possible service invocations. All property files are in Appendix B.

5.4.1 LTL Syntax

Properties in SPIN are specified using a Linear Temporal Logic (LTL) formula. Remember that each process is translated to a state machine and that execution is simulated by non-deterministically taking a transition from one of the process state machine. Temporal logic in general is used to determine if some execution path or set of execution paths fulfill a particular property. LTL specifically quantifies over the set of all (infinite) linear paths in the execution branch resulting from a given start state. In the properties we have specified, we only make use of two LTL temporal operators, EVENTUALLY (<>) and ALWAYS ([ ]).

> Definition 30: Semantics of EVENTUALLY (<>). An LTL formula, <>(p) is satisfied for an infinite computational path if and only if p becomes true in some state in the path.

> Definition 31: Semantics of ALWAYS ([ ]): An LTL formula, [ ](p) is satisfied for an infinite computational path if and only if p is true for all states in the path.

We use these temporal operators, as well as some additional macros to specify our desired correctness properties. Specifically, we check the model implementation generated from our scenario for the satisfiability of rule contexts, the correctness of all allow decisions, the correctness of all deny decisions, the correctness of all allowed method invocations, and for modality conflicts within each policy. We leave the specification of inter-policy modality conflicts to future work. In this section, we discuss the specification of these properties. When appropriate, we use macro definitions or abstract descriptions to facilitate the presentation of properties. Table 5 shows some common macro titles and their meanings.

MACRO TITLE	MACRO MEANING
ALLOW_DATA(i)	True if and only if the i^{th} rule of the Data Policy has an *allow* ruling.
DENY_DATA (i)	True if and only if the i^{th} rule of the Data Policy has a *deny* ruling
ALLOW_OBJ(i)	True if and only if the i^{th} rule of the Object Policy has an *allow* ruling.
DENY_OBJ (i)	True if and only if the i^{th} rule of the Object Policy has a *deny* ruling
ENABLED_DATA (i)	True if and only if the i^{th} rule of the Data Policy has an enabled contextual condition
ENABLED_OBJ (i)	True if and only if the i^{th} rule of the Object Policy has an enabled contextual condition
APP_DATA_TGT(i)	True if and only if the i^{th} rule of the Data Policy is applicable for Check #1
APP_DATA_SRC(i)	True if and only if the i^{th} rule of the Data Policy is applicable for Check #2
APP_OBJ_TGT(i)	True if and only if the i^{th} rule of the Object Policy is applicable for Check #3
APP_OBJ_SRC(i)	True if and only if the i^{th} rule of the Object Policy is applicable for Check #4

Table 5 Common macros for properties

5.4.2 Satisfiability of Rule Contexts

In order to be sure that each rule context is satisfied infinitely often, we check that each context of each rule of each policy can "ALWAYS EVENTUALLY become enabled". This is simply done by enumerating over all of the rule contexts, making sure that each context can become true infinitely often (i.e. ALWAYS EVENTUALLY). Since dc0-dc2, oc0, and oc1 are the rule contexts, the specification is

```
([](<>dc0 && <>dc1 && <>dc2 && <>oc0 && <>oc1))
```

Note that [] denotes temporal operator ALWAYS, <> denotes temporal operator EVENTUALLY, && denotes logical operator AND, and || denotes logical operator OR.

5.4.3 Allow Correctness

When checking for the correctness of an allow decision, we want to make sure that if the global decision variable RULING == ALLOW, then all four of the request set checks are allowed. A single request is allowed in one of three ways: 1) the request does not actually exist because either a data item or an environment object is null, 2) all allow rules applicable to that request are enabled with all applicable denies disabled, or 3)

there is no applicable rule and the default for the policy is ALLOW. We define macros to check if each atomic request is not valid.

```
#define NULL_DATA_TGT (services[tgt].data_in == 0)
#define NULL_DATA_SRC (services[tgt].data_out == 0)
#define NULL_OBJ_TGT (services[tgt].object == 0)
#define NULL_OBJ_SRC (services[tgt].object == 0)
```

If P_{DATA} is the data policy, here is an abstract example specification for the Data-to-Target check.

```
FORALL(Rules i∈ PDATA)
       ((RULING == ALLOW) →
              ((NULL_DATA_TGT)
              ||
              ((ALLOW_DATA(i)  && APP_DATA_TGT(i)) → ENABLED_DATA(i)) &&
              ((DENY_DATA(i)  && APP_DATA_TGT(i)) → !ENABLED_DATA(i))
              ||
              ((DEFAULT_DATA == ALLOW) && !APP_DATA_TGT(i))))
```

Using the disjunctive form for implication, we obtain

```
FORALL(Rules i∈ PDATA)
       (! (RULING == ALLOW) || (
              (NULL_DATA_TGT)
              ||
              ((!(ALLOW_DATA(i)  && APP_DATA_TGT(i)) || ENABLED_DATA(i)) &&
              (!(DENY_DATA(i)  && APP_DATA_TGT(i)) || !ENABLED_DATA(i)))
              ||
              ((DEFAULT_DATA == ALLOW) && !APP_DATA_TGT(i))))
```

The other request checks are given similar properties but are specific to their request sets. The macros are then combined with conjunctions into one global safety property `all_allow_logic`. Conjunctions are used because all requests must allow. If the macro definitions for the request sets are `allow_data_tgt`, `allow_data_src`, `allow_obj_tgt`, and `allow_obj_src`, then `all_allow_logic` is defined as

```
#define all_allow_logic (allow_data_tgt && allow_data_src && allow_obj_tgt
                         && allow_obj_src)
```

The final specification for allow correctness becomes

```
([] (all_allow_logic))
```

Note that the property vacuously holds if the global decision does not equal `ALLOW`.

5.4.4 Deny Correctness

When checking for the correctness of a deny decision, we want to make sure that if the global decision variable RULING == DENY, then at least one of the request set checks has been denied. This means that, for some request set, either some applicable allow rule is disabled or some applicable deny rule is enabled. If P_{DATA} again is the data policy, here is an abstract example of the deny correctness property for the Data-to-Target check.

```
FORALL(Rules i∈ P_DATA)
       ((RULING == DENY) → (
              ((ALLOW_DATA(i)  && APP_DATA_TGT(i))→ !ENABLED_DATA(i)) ||
              ((DENY_DATA(i)  && APP_DATA_TGT(i)) → ENABLED_DATA(i)) ||
               ((DATA_DEFAULT == DENY) && !APP_DATA_TGT(i))))
```

Using the disjunctive form for implication again, we get

```
FORALL(Rules i∈ P_DATA)
       (!(RULING == DENY) ||(
              (!(ALLOW_DATA(i) && APP_DATA_TGT(i))|| !ENABLED_DATA(i)) ||
              (!(DENY_DATA(i) && APP_DATA_TGT(i)) || ENABLED_DATA(i)) ||
              ((DATA_DEFAULT == DENY) && !APP_DATA_TGT(i))))
```

The property again is broken into four macros. This time, in keeping with our semantics, the macros are combined with disjunctions because only one atomic request needs to be denied. If the macro definitions for the request sets are `deny_data_tgt, deny_data_src, deny_obj_tgt`, and `deny_obj_src`, and they are combined into macro `all_deny_logic`, in the following way,

```
#define all_deny_logic (deny_data_tgt || deny_data_src || deny_obj_tgt ||
                        deny_obj_src)
```

The final specification for deny correctness becomes

```
([] (all_deny_logic))
```

Note again that the property vacuously holds if the global decision does not equal `DENY`.

5.4.5 Correctness of Service Invocations

Earlier we listed several possible invocations in our example scenario. We seek to show that if these service invocations occur, then they occur within the boundaries of our authorization semantics. This in effect shows the correctness of our system execution. We define the following macros for each possible service invocation where an integer represents the index of a service in the `services` data structure. The indices 0 – 5 are mapped to table 2 from top to bottom.

```
#define src4tgt0 ((RULING == ALLOW) && (src == 4) && (tgt == 0))
#define src4tgt1 ((RULING == ALLOW) && (src == 4) && (tgt == 1))
…
… /* Other #defines omitted */
…
#define src5tgt2 ((RULING == ALLOW) && (src == 5) && (tgt == 2))
#define src5tgt3 ((RULING == ALLOW) && (src == 5) && (tgt == 3))
```

We reuse the allow and deny correctness properties to compose a new property that states that "it is ALWAYS the case that allow and deny correctness are true and each desired service invocation EVENTUALLY happens." The final specification of this property is

```
([](all_allow_logic && all_deny_logic &&
      <>src4tgt0 && <>src4tgt1 && <>src4tgt2 && <>src4tgt3 &&
      <>src5tgt0 && <>src5tgt1 && <>src5tgt2 && <>src5tgt3 &&))
```

5.4.6 Modality Conflicts in Atomic Requests

To show the absence of modality conflicts for atomic requests, we must show that if two rules within a policy have opposing rulings, and one rule has terms that are ancestors of corresponding terms in the other rule, then the context of one rule is not a subset of the context of the other rule. We define macros to check for opposing rules and ancestor relationships between rules, and the context relationship is handled in the property itself.

`MODAL_DATA_PAIR(i, j)` is true if the i^{th} data rule has an opposite ruling than the j^{th} data rule but their domain, purpose, and data fields are hierarchically related.

`MODAL_OBJ_PAIR(m, n)` is true if the m^{th} object rule has an opposite ruling than the n^{th} object rule but their domain, purpose, and object fields are hierarchically related.

Furthermore, two contexts are not subsets of one another if each is true at some point when the other is false, which, if *i* and *j* are indices of policy rules, can be stated abstractly as

```
(<>(ENABLED(i) && !ENABLED(j)) && (<>!ENABLED(i) && ENABLED(j))).
```

The entire abstract specification for this property is then

```
FORALL(Rules i,j∈ P_DATA AND Rules m,n ∈ P_OBJ such that i != j and m != n)
      ((MODAL_DATA_PAIR(i,j) ->
            (<>(ENABLED_DATA(i) && !ENABLED_DATA(j)) &&
             <>(!ENABLED_DATA(i) && ENABLED_DATA(j))) &&
      ((MODAL_OBJ_PAIR(m,n) ->
            (<>(ENABLED_DATA(m) && !ENABLED_DATA(n)) &&
```

```
<>(!ENABLED_DATA(m) && ENABLED_DATA(n)) &&
```

The actual specification explicitly enumerates all possible combinations of rules.

6. CONCLUSIONS AND FUTURE WORK

In this thesis, we present a novel service-oriented privacy policy model for smart home environments that incorporates the selective control of both personal information and environment object usage. We then show how the SPIN model checker can be used to verify certain aspects of a simple example scenario of our privacy model, including the absence of certain conflicts, and the correctness of service invocations and evaluation decisions. The main contributions of this work are

- The extension of personal privacy to include the control of how household objects are used by smart home services.
- The introduction of service-oriented computing to help bind system resources to the policy space.
- The formal definition of privacy model that authorizes both the flow of personally sensitive data and the control of environment objects based on inhabitant preferences and system contexts.
- The introduction of model checking to verify privacy policy models and their enforcement.

We use a service-oriented approach because the service-oriented architecture addresses the heterogeneity and dynamicity that smart home environments provide. Furthermore, because the functionality of a smart home is defined by the abilities of the services within the smart home, we use service descriptions to bind resources to the policy space. Another feature of our model is the heavy use of logical hierarchies to order physical resources. Using hierarchies allows for simpler rule specification because rules are implicitly inherited down the hierarchy, requiring fewer rules to be specified in total. However, evaluating a request can become more costly because the hierarchical relationships between terms need to be explored to achieve a ruling. Some works, such as EPAL/E-P3P expand applicable rules on the fly during the request. This can be a very expensive process and can result in conflicting rulings. We avoid this extra cost by requiring service descriptions to map to leaf entities in hierarchies and storing hierarchy path information. This optimization could be used in a real system implementation as well.

Despite its novelty, our privacy model does have some limitations that should be addressed in future work. First, assuming instantaneous evaluation and service execution is not very realistic, so we need to extend our privacy model to include continuous methods and non-instantaneous evaluation. This could possibly be addressed with some sort of context subscription mechanism in which component subscribes to a particular context and receive notifications when the context changes. These notifications could then somehow be used to halt a previously authorized service execution that now violates the privacy model. However, support for such a mechanism will require effort at both the policy model and the SOA implementation levels. Second, we need to investigate the application of our model to a decentralized environment. Some SOAs, such as web services, do not use centralized invocation, so our current privacy model is not usable for these architectures. Also, centralized evaluation may degrade system performance, especially as the number of requests and the number of applicable rules to a request increases. We will look into the possibility of distributing the storage and evaluation of policies, but then deployment and synchronization issues become increasingly important. Other

future work can also include distinguishing between data collection, storage, and data usage policies; investigating alternative views for object control; extending the model to include multiple inhabitants, service dynamics, and policy changes; formally defining conflicts in our model; and investigating the relationship of purposes and services. We also plan to investigate the use of other model checkers and hope to ultimately provide tool support for the specification and automatic implementation and verification of our privacy model.

7. REFERENCES

[1] R. Babbitt, D. Lu, C. Chang, and J. Wong, "Requirements Engineering for Smart Homes to Support Successful Aging, Disability, and Independence" European Academy of Sciences Annals 2005, pp. 107-127.

[2] R. Babbitt, J. Wong, S. Mitra, and C. Chang, "Privacy Management in Smart Homes: Design and Analysis," Proc. of International Conference on Aging, Disability, Independence (ICADI), pp. 55-64, February 2006.

[3] S. Helal, "Programming Pervasive Spaces", IEEE Pervasive Computing, Vol. 4, No. 1, pp 84-87, 2005.

[4] A. I. Anton, Q. He, D. and Baumer, "Inside JetBlue's Privacy Policy Violations", IEEE Security and Privacy, Vol. 02, No. 6, pp. 12-18, 2004.

[5] A. Helal, W. Mann, H. Elzabadani, J. King, Y. Kaddourah and E. Jansen, "Gator Tech Smart House: A Programmable Pervasive Space", IEEE Computer Magazine, March 2005, pp 64-74.

[6] University of Texas, Arlington, MavHome: Managing an Adaptive Versatile Home, http://cygnus.uta.edu/mavhome/

[7] Georgia Institute of Technology, The Aware Home Research Initiative, http://www.awarehome.gatech.edu/

[8] Massey University, The Massey University Smart House Project, http://smarthouse.massey.ac.nz/

[9] M. Papazoglou and D. Georgakopoulos, "Service-oriented Computing", Communications of the ACM Vol. 46, No. 10, pp. 25-28, 2003.

[10] M. Papazoglou, "Extending the Service Oriented Architecture", Business Integration Journal, February 2005, pp. 18-21.

[11] OSGi Alliance, http://www.osgi.org

[12] World Wide Web Consortium, SOAP Version 1.2, Part 0: Primer, World Wide Web Consortium Recommendation, June 2003, http://www.w3.org/TR/2003/REC-soap12-part0-20030624/

[13] World Wide Web Consortium, Web Service Description Language (WSDL)Version 2.0, Part 0: Primer, World Wide Web Consortium Recommendation, March 2006, http://www.w3.org/TR/2006/CR-wsdl20-primer-20060327/.

[14] OASIS, UDDI Version 3.0.2, http://uddi.org/pubs/uddi-v3.0.2-20041019.htm

[15] Universal Plug and Play Device Architecture, Version 1.0, 08 June 2000, http://www.upnp.org/download/UPnPDA10_20000613.htmhttp://www.upnp.org.

[16] Sun Microsystems, Jini Specifications Archive, Version 2.0.1, 2005, http://java.sun.com/products/jini/2_0_1index.html.

[17] International Telecommunications Union, "Privacy and Ubiquitous Network Societies", Background Paper, ITU Workshop on Ubiquitous Network Societies, April 2005.

[18] S. J. Milberg, S. J. Burke, H. J. Smith, E. A. Kallman, "Values, Personal Information Privacy and Regulatory Approaches", Communication of the ACM, Vol. 38, No.12, 1995, pp. 65-74.

[19] S.T. Margulis, "On the Status and Contribution of Westin's and Altman's Theories of Privacy", Journal of Social Issues, Vol. 59, No. 2, pp. 411-429, July 2003.

[20] A. Westin. Privacy and Freedom. Atheneum, New York, NY, 1967.

[21] I. Altman. The Environment and Social Behavior. Brookes/Cole, Monterey, CA, 1975.

[22] Organisation for Economic Co-operation and Development (OECD). OECD Guidelines on the Protection of Privacy and Transborder Flows of Personal Data, 2001. Available: http://www1.oecd.org/publications/e-book/9302011E.PDF

[23] Federal Trade Commission. Privacy Online: A report to Congress, 1998, Available: http://www.ftc.gov/reports/privacy3/fairinfo.htm

[24] Canadian Standards Association. Model Code for the Protection of Personal Information. March 1996. Available: http://www.csa.ca/standards/privacy/code/Default.asp?articleID=5286&language=English

[25] G. Stone, B. Lundy, G. Xie, "Network Policy Languages: A Survey and a New Approach," IEEE Network, Vol. 15, No. 1, pp. 10–21, January 2001.

[26] M. Sloman, E. Lupu, "Security and Management Policy Specification", IEEE Network, Vol. 16, No. 2, pp.10-19, Mar/Apr 2002.

[27] E. C. Lupu, M. Sloman. "Conflicts in Policy-Based Distributed Systems Management"" IEEE Transactions on Software Engineering, Vol. 25, No. 6, pp. 852-869, 1999.

[28] N. Dunlop, J. Indulska, and K. Raymond, "Dynamic Conflict Detection in Policy-Based Management systems," Proc. of Sixth International Enterprise Distributed Object Computing Conference, pp. 15- 26, 2002.
[29] N. Dunlop, J. Indulska, and K. Raymond, "Methods for conflict detection in policy-based management systems," Proc. of Seventh International Enterprise Distributed Object Computing Conference, pp. 98-109, 2003
[30] T. Dursun and B. Orencik,, "POLICE Distributed Conflict Detection Architecture," IEEE Conference on Communication, Vol.4, No. 5, pp. 2081- 2085, 2004.
[31] E. Lupu, and M. Sloman, "Conflicts in Policy-based Distributed Systems Management", IEEE Transactions On Software Engineering, Vol 25. No 6, pp. 852-869, 1999.
[32] Holzmann, G. J., The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley, 2004.
[33] V. Bellotti and A. Sellen, "Design for Privacy in Ubiquitous Computing Environments," in Proc. of The Third European Conference on Computer Supported Cooperative Work 1993 (ECSCW'93), pp. 77-92.
[34] M. Langheinrich, "Privacy by Design - Principles of Privacy-Aware Ubiquitous Systems," Lecture Notes in Computer Science, Vol. 2201, pp. 273, January 2001.
[35] X. Jiang, J. I. Hong, J. A. Landay. "Approximate Information Flows: Socially-Based Modeling of Privacy in Ubiquitous Computing," Lecture Notes in Computer Science, Volume 2498, Jan 2002, Page 176.
[36] L. Palen and P. Dourish, "Unpacking "Privacy" for a Networked World," CHI Letters, Vol. 5, No. 1, pp. 129-136, 2003.
[37] J. I.Hong, J. Ng, S. Lederer, and J. A. Landay, "Privacy Risk Models for Designing Privacy-Sensitive Ubiquitous Computing Systems," in Proc. of Designing Interactive Systems: Processes, Practices, Methods, Techniques, pp. 91-100, 2004.
[38] B.A. Price, K Adam, and B Nuseibeh, "Keeping ubiquitous computing to yourself: A Practical Model for User Control of Privacy", International Journal of Human-Computer Studies, Vol. 63, Issues 1-2, pp. 228-253, July 2005.
[39] L. Cranor et al., The Platform for Privacy Preferences 1.0 (P3P1.0) Specification, World Wide Web Consortium Recommendation, April 2002; www.w3.org/TR/P3P/.
[40] L. Cranor, M. Langheinrich, and M. Marchiori. A P3P Preference Exchange Language 1.0 (APPEL1.0). W3C Working Draft, April 2002.
[41] R. Agrawal, J. Kiernan, R.Srikant, and Y. Xu, "An XPath-based preference language for P3P," in Proc. of the Twelfth International World Wide Web Conference (WWW2003), pp. 629–639, May 2003.
[42] T.Yu, N. Li, A. Anton, "A Formal Semantics for P3P," in Proc. of the 2004 Workshop on Secure Web Services, pp. 1-8, October 2004.
[43] M. Langheinrich, "A Privacy Awareness System for Ubiquitous Computing Environments," Lecture Notes in Computer Science, Vol. 2498, pp. 237, January 2002.
[44] M. S. Ackerman, "Privacy in pervasive environments: next generation labeling protocols," Personal and Ubiquitous Computing, Vol. 8, pp. 430–439, 2004.
[45] D. Hong, M. Yuan and V. Y. Shen, "Dynamic Privacy Management: a Plug-in Service for the Middleware in Pervasive Computing," in Proc. of 7th International Conference on Human Computer Interaction with Mobile Devices and Services (MobileHCI'05), pp. 1-8, September 2005.
[46] L. C. J. Dreyer and M. S. Olivier. "A Workbench for Privacy Policies," Proc. of 22nd Annual International Computer Software and Applications Conference (COMPSAC'98), pp. 350-355, 1998.
[47] L. C. J. Dreyer and M.S. Olivier. "Dynamic Aspects of the InfoPriv Model for Information Privacy," Proc. of the 9th International Workshop on Database and Expert Systems Applications, pp. 340-345, IEEE, 1998.
[48] S. Fischer-Hubner and A.Ott, In Proc. of the 21st National Information Systems Security Conference, October 1998, http://www.rsbac.org/niss98.htm.
[49] He, Q. "Privacy Enforcement with an Extended Role-Based Access Control Model." NCSU Computer Science Technical Report. 2003.
[50] Byun, J., Bertino,, E. and Li, N. "Purpose Based Access Control of Complex Data for Privacy Protection." In Proceedings of the Tenth ACM Symposium of Access Control Models and Technologies (SACMAT '05), pp. 102-110, 2005.
[51] Enterprise Privacy Authorization Language (EPAL),Version 1.2, 2003; the version submitted to the W3C. Available at http://www.w3.org/Submission/2003/SUBMEPAL-20031110/.

[52] P. Ashley, S. Hada, and G. Karjoth, "E-P3P Policies and Privacy Authorization, Workshop of Privacy. In Workshop on Privacy and the Electronic Society, WPES'02, November 2002.
[53] M. Backes, B. Pfitzmann, and M. Schunter. "A Toolkit for Managing Enterprise Privacy Policies. In Proceedings of 8th European Symposium on Research in Computer Security," ESORICS '03, pp. 162 – 180, October 2003.
[54] M. Backes, M. Duermuth, and R. Steinwandt." An Algebra for Composing Enterprise Privacy Policies," In Proc. of 9th European Symposium on Research in Computer Security, ESORICS '04, Vol. 3193, pp. 33 - 52, September 2004.
[55] A. H. Anderson. "A Comparison of Two Privacy Policy Languages: EPAL and XACML. In Proceedings of the 3rd ACM Workshop on Secure Web Services (SWS '06)," pp. 53-60, November 2006.
[56] X. Jiang, J. I. Hong and J. Landay, "Approximate Information Flows: Socially-based Modeling of Privacy in Ubiquitous Computing." In Proc. of Fourth International Conference on Ubiquitous Computing (Ubicomp'2002), pp. 176-193, 2002.
[57] X. Jiang and J. A. Landay. "Modeling Privacy Control in Context-Aware Systems," IEEE Pervasive Computing, Volume 1 Issue 3, pp. 59-63, July-September, 2002.
[58] Spin – Formal Verification. http://www.spinroot.com

APPENDIX A: MODEL SOURCE CODE

The files in this appendix are for the Promela implementation of our privacy policy model. The file `global.pml` contains the definitions of the data structures and terms used by the Privacy Manager and Context Generator processes. The file `privacy-manager.pml` contains the implementation of the Privacy Manager process. The file `context-generator.pml` contains the implementation of the Context Generator process.

A.1 global.pml

```
/*-----------------------------------------------------------------*/

  AUTHOR: Ryan Babbitt, Iowa State University
  FILE: global.pml
  DATE: November 26, 2006

  DESCRIPTION:
       This file defines all of the global variables used by the
  privacy model. It defines the structures, privacy rules, and
  service invocation requests, as well as context variables.

-----------------------------------------------------------------*/

/* Constants for rulings */
#define NIL 0
#define ALLOW 1
#define DENY 2

/* Global decision variable */
byte RULING;

/* Default policy rulings */
#define DATA_DEFAULT ALLOW
#define OBJ_DEFAULT ALLOW

/*-----------------------*/
/*        Terms      */
/*-----------------------*/

mtype = {ALL_DATA, LOCATION, TIME, VIDEO};     /* data items & types */
mtype = {ALL_DOMAINS, SYSTEM, FAMILY, COMPANY};      /* data users & domains */
mtype = {ALL_OBJECTS, CAMERA};                 /* objects & object types */
mtype = {ALL_PURPOSES, MONITOR, SENSE, MOVE};  /* purposes */

/*-------------------------------*/
/*         Hierarchies           */
/*-------------------------------*/

/* properties */
#define MAX_DEPTH 1 /* longest depth in any hierarchy */
#define MAX_LEAVES 3 /* Most leaves of any hierarcy */

/* A leaf stores a list of its ancestors */
typedef leaf{
      byte value;
      byte ancestors[MAX_DEPTH];
};

/* A hierarchy is a collection of leaves */
```

```
typedef hierarchy{
      leaf leaves[MAX_LEAVES];
};

/* The 4 hierarchies form the set of terms */
typedef terms_t{
      hierarchy h[4];
};

terms_t terms;

/* Indices of hierarchies */
#define DOMAIN_INDEX 0
#define PURPOSE_INDEX 1
#define DATA_INDEX 2
#define OBJECT_INDEX 3

/*-------------------------------*/
/* Privacy policy rule structures */
/*-------------------------------*/

typedef data_rule{
      byte ruling;
      mtype data;
      mtype domain;
      mtype purpose;
      bit context;
};

typedef obj_rule{
      byte ruling;
      mtype object;
      mtype domain;
      mtype purpose;
      bit context;
};

/*------------------------*/
/* The privacy policies  */
/*------------------------*/

 #define NUM_D_RULES 3
 #define NUM_O_RULES 2

 data_rule data_policy[NUM_D_RULES];
 obj_rule obj_policy[NUM_O_RULES];

/*-------------------------------*/
/*     Contexts for rules   */
/*-------------------------------*/

 /* DATA RULE 0 */
 #define dc0 true

 /* DATA RULE 1 */
 #define dc1 (room == bathroom)

 /* DATA RULE 2 */
 #define dc2 ((room == bedroom) && (((hour >= 6) && (hour <= 7)) || \
                                ((hour >= 21) && (hour <= 22))))
```

```
 /* OBJECT RULE 0 */
 #define oc0 (room == bathroom)

 /* OBJECT RULE 1 */
 #define oc1 ((room == bedroom) && (((hour >= 6) && (hour <= 7)) || \
                              ((hour >= 21) && (hour <= 22))))

/*----------------------*/
/* Service descriptions */
/*----------------------*/

typedef service{
       mtype data_in;
       mtype data_out;
       mtype object;
       mtype domain;
       mtype purpose;
};

 #define NUM_SERVICES 6
 service services[NUM_SERVICES];

 byte tgt; /* target index for requests */
 byte src; /* source index for requests */

/*-------------------------------*/
/* Environment context variables */
/*-------------------------------*/

 byte hour = 0; /* values from 0 - 23 */

/*-------------------------------*/
/*  Occupant context variables   */
/*-------------------------------*/

 mtype = {living, bathroom, kitchen, bedroom};   /* values for room */
 mtype room;
```

==========

A.2 privacy-manager.pml

==========

```
/* ------------------------------------------------------------------
File: privacy-manager.pml
Author: Ryan Babbitt, Iowa State University
Date: November 30, 2006

Description: This process models the privacy management system in a
smart home. Its execution occurs in 4 phases:

       1) Populate the hierarchies, privacy policies, and service registry
       2) Randomly generate a request
       3) Evaluate all the context conditions (atomically)
       4) Evaluate request sets/atomic requests
              a) Data-to-Target check
              b) Data-to-Source check
              c) Object-by-Target check
              d) Object-by-Source check

Note: Steps 2-4 are then repeated ad infinitum.
```

```
-----------------------------------------------------------------------*/

/* What is the index of value x in hierchy z? The answer is stored in i. */
inline INDEX(x, z, i)
{
  i =
       ((terms.h[(z)].leaves[0].value == (x)) ->  0 :
       ((terms.h[(z)].leaves[1].value == (x)) ->  1 :
       ((terms.h[(z)].leaves[2].value == (x)) ->  2 : 255)))
}

/* Is the value y an ancestor of the leaf at index x in hierarchy z?
The response is stored in r */
inline IS_ANCESTOR(x, y, z, res)
{
  res =
       ((x != 255) && terms.h[(z)].leaves[(x)].value == (y)) ||
       ((x != 255) && terms.h[(z)].leaves[(x)].ancestors[0] == (y))
}


/* Privacy Manger Process */

active proctype pms()
 {
       byte index; /* rule iterator */
       data_rule drl; /* data rule container */
       obj_rule orl; /* object rule container */

       /* variables for intermediate decision logic of atomic requests */
       /* rq1 - data-to-target, rq2 - data-to-source  */
       /* rq3 - objecy-by-target, rq4 - object-by-source */
       bool allow_rq1, deny_rq1, allow_rq2, deny_rq2;
       bool allow_rq3, deny_rq3, allow_rq4, deny_rq4;

        /* stores results of applicability testing */
       bool test1, test2, test3;

       /* stores the index of a leaf */
       byte leaf_index;

       d_step
       {
              /* Domain Hierarchy */
              terms.h[DOMAIN_INDEX].leaves[0].value = SYSTEM;
              terms.h[DOMAIN_INDEX].leaves[0].ancestors[0] = ALL_DOMAINS;
              terms.h[DOMAIN_INDEX].leaves[1].value = FAMILY;
              terms.h[DOMAIN_INDEX].leaves[1].ancestors[0] = ALL_DOMAINS;
              terms.h[DOMAIN_INDEX].leaves[2].value = COMPANY;
              terms.h[DOMAIN_INDEX].leaves[2].ancestors[0] = ALL_DOMAINS;

              /* Purpose Hierarchy */
              terms.h[PURPOSE_INDEX].leaves[0].value = MOVE;
              terms.h[PURPOSE_INDEX].leaves[0].ancestors[0] = ALL_PURPOSES;
              terms.h[PURPOSE_INDEX].leaves[1].value = MONITOR;
              terms.h[PURPOSE_INDEX].leaves[1].ancestors[0] = ALL_PURPOSES;
              terms.h[PURPOSE_INDEX].leaves[2].value = SENSE;
              terms.h[PURPOSE_INDEX].leaves[2].ancestors[0] = ALL_PURPOSES;

              /* Data Type Hierarchy */
```

```
terms.h[DATA_INDEX].leaves[0].value = LOCATION;
terms.h[DATA_INDEX].leaves[0].ancestors[0] = ALL_DATA;
terms.h[DATA_INDEX].leaves[1].value = TIME;
terms.h[DATA_INDEX].leaves[1].ancestors[0] = ALL_DATA;
terms.h[DATA_INDEX].leaves[2].value = VIDEO;
terms.h[DATA_INDEX].leaves[2].ancestors[0] = ALL_DATA;

/* Object Type Hierarchy */
terms.h[OBJECT_INDEX].leaves[0].value = CAMERA;
terms.h[OBJECT_INDEX].leaves[0].ancestors[0] = ALL_OBJECTS;

/* Data Privacy Policy */
data_policy[0].ruling = ALLOW;
data_policy[0].domain = ALL_DOMAINS;
data_policy[0].purpose = ALL_PURPOSES;
data_policy[0].data = LOCATION;

data_policy[1].ruling = DENY;
data_policy[1].domain = ALL_DOMAINS;
data_policy[1].purpose = SENSE;
data_policy[1].data = VIDEO;

data_policy[2].ruling = DENY;
data_policy[2].domain = ALL_DOMAINS;
data_policy[2].purpose = SENSE;
data_policy[2].data = VIDEO;

/* Object Privacy Policy */
obj_policy[0].ruling = DENY;
obj_policy[0].domain = ALL_DOMAINS;
obj_policy[0].purpose = MOVE;
obj_policy[0].object = CAMERA;

obj_policy[1].ruling = DENY;
obj_policy[1].domain = ALL_DOMAINS;
obj_policy[1].purpose = MOVE;
obj_policy[1].object = CAMERA;

/* Camera Video Service */
services[0].domain = SYSTEM;
services[0].purpose = SENSE;
services[0].data_in = LOCATION;
services[0].data_out = VIDEO;
services[0].object = 0;

/* Camera Actuator Service */
services[1].domain = FAMILY;
services[1].purpose = MOVE;
services[1].data_in = LOCATION;
services[1].data_out = 0;
services[1].object = CAMERA;

/* Time Service/hour */
services[2].domain = SYSTEM;
services[2].purpose = SENSE;
services[2].data_in = 0;
services[2].data_out = TIME;
services[2].object = 0;

/* Location Service */
services[3].domain = SYSTEM;
```

```
        services[3].purpose = SENSE;
        services[3].data_in = 0;
        services[3].data_out = LOCATION;
        services[3].object = 0;

        /* Monitor Service/Company */
        services[4].domain = COMPANY;
        services[4].purpose = MONITOR;
        services[4].data_in = 0;
        services[4].data_out = VIDEO;
        services[4].object = 0;

        /* Monitor Service/Family */
        services[5].domain = FAMILY;
        services[5].purpose = MONITOR;
        services[5].data_in = 0;
        services[5].data_out = VIDEO;
        services[5].object = 0;
}

/* Request generation and evaluation loop */
do
::
        atomic
        {
                /* select source service/method */
                do
                :: src=4;
                   break;
                :: src=5;
                   break;
                od;

                /* select a target service/method different
                from the source service/method */
                do
                :: tgt=0;
                   break;
                :: tgt=1;
                   break;
                :: tgt=2;
                   break;
                :: tgt=3;
                   break;
                od;

                /* Evaluate the contextual conditions */
                data_policy[0].context = dc0;
                data_policy[1].context = dc1;
                data_policy[2].context = dc2;

                obj_policy[0].context = oc0;
                obj_policy[1].context = oc1;

                /* Evaluate the request */

                /* reset evaluation logic variables */
                index = 0;
                allow_rq1 = 0;
                deny_rq1 = 0;
                allow_rq2 = 0;
```

```
deny_rq2 = 0;
allow_rq3 = 0;
deny_rq3 = 0;
allow_rq4 = 0;
deny_rq4 = 0;

/* Data-to-Target Check and Data-to-Source Check */
do
:: (index < NUM_D_RULES) ->

{/* Check for null data item */

       /* Current data_rule */
       drl.ruling = data_policy[index].ruling;
       drl.data = data_policy[index].data;
       drl.domain = data_policy[index].domain;
       drl.purpose = data_policy[index].purpose;

       test1 = 0; test2 = 0; test3 = 0;

/* Does current rule apply to target domain? */
INDEX(services[tgt].domain, DOMAIN_INDEX, leaf_index);
IS_ANCESTOR(leaf_index, drl.domain, DOMAIN_INDEX, test1);

/* Does current rule apply to target purpose? */
INDEX(services[tgt].purpose, PURPOSE_INDEX, leaf_index);
IS_ANCESTOR(leaf_index, drl.purpose, PURPOSE_INDEX, test2);

/* Does current rule apply to target in data? */
INDEX(services[tgt].data_in, DATA_INDEX, leaf_index);
IS_ANCESTOR(leaf_index, drl.data, DATA_INDEX, test3);

       /* Applicable and enabled allow rule */
       allow_rq1 = allow_rq1 ||
       ((drl.ruling == ALLOW) &&
       test1 && test2 && test3 &&
       data_policy[index].context);

       /* Applicable and disabled allow data_rule */
       deny_rq1 = deny_rq1 ||
       ((drl.ruling == ALLOW) &&
       test1 && test2 && test3 &&
       !data_policy[index].context);

       /* Applicable and enabled deny data_rule */
       deny_rq1 = deny_rq1 ||
       ((drl.ruling == DENY) &&
       test1 && test2 && test3 &&
       data_policy[index].context);

} unless {
       (services[tgt].data_in == 0);
       allow_rq1 = 1;
};

{ /* Check null return data */

       test1 = 0; test2 = 0; test3 = 0;

/* Does current rule apply to source domain? */
INDEX(services[src].domain, DOMAIN_INDEX, leaf_index);
```

```
        IS_ANCESTOR(leaf_index, drl.domain, DOMAIN_INDEX, test1);

        /* Does current rule apply to source purpose? */
        INDEX(services[src].purpose, PURPOSE_INDEX, leaf_index);
        IS_ANCESTOR(leaf_index, drl.purpose, PURPOSE_INDEX, test2);

        /* Does current rule apply to target out data? */
        INDEX(services[tgt].data_out, DATA_INDEX, leaf_index);
        IS_ANCESTOR(leaf_index, drl.data, DATA_INDEX, test3); */

                /* Applicable and enabled allow rule */
                allow_rq2 = allow_rq2 ||
                ((drl.ruling == ALLOW) &&
                test1 && test2 && test3 &&
                data_policy[index].context);

                /* Applicable and disabled allow data_rule */
                deny_rq2 = deny_rq2 ||
                ((drl.ruling == ALLOW) &&
                test1 && test2 && test3 &&
                !data_policy[index].context);

                /* Applicable and enabled deny data_rule */
                deny_rq2 = deny_rq2 ||
                ((drl.ruling == DENY) &&
                test1 && test2 && test3 &&
                data_policy[index].context);

        } unless
        {
                (services[tgt].data_out == 0);
                allow_rq2 = 1;
        };

        index++;

:: else -> break;
od;

        /* Object-by-Target and Object-by-Source Checks */
        index = 0;
        do
        :: (index < NUM_O_RULES) ->

        {/* Check for null object */

                /* Current obj_rule */
                orl.ruling = obj_policy[index].ruling;
                orl.object = obj_policy[index].object;
                orl.domain = obj_policy[index].domain;
                orl.purpose = obj_policy[index].purpose;

                test1 = 0; test2 = 0; test3 = 0;

        /* Does current rule apply to target domain? */
        INDEX(services[tgt].domain, DOMAIN_INDEX, leaf_index);
        IS_ANCESTOR(leaf_index, orl.domain, DOMAIN_INDEX, test1);

        /* Does current rule apply to target purpose? */
        INDEX(services[tgt].purpose, PURPOSE_INDEX, leaf_index);
        IS_ANCESTOR(leaf_index, orl.purpose, PURPOSE_INDEX, test2);
```

```
/* Does current rule apply to target object? */
INDEX(services[tgt].object, OBJECT_INDEX, leaf_index);
IS_ANCESTOR(leaf_index, orl.object, OBJECT_INDEX, test3); */

      /* applicable and enabled allow obj_rule */
      allow_rq3 = allow_rq3 ||
      ((orl.object == ALLOW) &&
      test1 && test2 && test3 &&
      obj_policy[index].context);

      /* Applicable and disabled allow obj_rule */
      deny_rq3 = deny_rq3 ||
      ((orl.ruling == ALLOW) &&
      test1 && test2 && test3 &&
      !obj_policy[index].context);

      /* Applicable and enabled deny obj_rule */
      deny_rq3 = deny_rq3 ||
      ((orl.ruling == DENY) &&
      test1 && test2 && test3 &&
      obj_policy[index].context);

test1 = 0; test2 = 0; test3 = 0;

/* Does current rule apply to source domain? */
INDEX(services[src].domain, DOMAIN_INDEX, leaf_index);
IS_ANCESTOR(leaf_index, orl.domain, DOMAIN_INDEX, test1);

/* Does current rule apply to source purpose? */
INDEX(services[src].purpose, PURPOSE_INDEX, leaf_index);
IS_ANCESTOR(leaf_index, orl.purpose, PURPOSE_INDEX, test2);

/* Does current rule apply to target object? */
INDEX(services[tgt].object, OBJECT_INDEX, leaf_index);
IS_ANCESTOR(leaf_index, orl.object, OBJECT_INDEX, test3);

      /* Check object of response domain */
      /* applicable and enabled allow obj_rule */
      allow_rq4 = allow_rq4 ||
      ((orl.object == ALLOW) &&
      test1 && test2 && test3 &&
      obj_policy[index].context);

      /* Applicable and disabled allow obj_rule */
      deny_rq4 = deny_rq4 ||
      ((orl.ruling == ALLOW) &&
      test1 && test2 && test3 &&
      !obj_policy[index].context);

      /* Applicable and enabled deny obj_rule */
      deny_rq4 = deny_rq4 ||
      ((orl.ruling == DENY) &&
      test1 && test2 && test3 &&
      obj_policy[index].context);

} unless
{
      (services[tgt].object == 0);
      allow_rq3 = 1;
      allow_rq4 = 1;
```

```
                };

                index++;

                :: else -> break;
                od;

                /*----------------------------------------*/
                /* Decide ruling using default if necessary */
                /*----------------------------------------*/

            /* Apply default ruling to data request if needed */
            allow_rq1 = ((!allow_rq1 && !deny_rq1) ->
                    ((DATA_DEFAULT == ALLOW) -> 1 : 0) : allow_rq1);
            deny_rq1 = ((!allow_rq1 && !deny_rq1) ->
                    ((DATA_DEFAULT == ALLOW) -> 0 : 1) : deny_rq1);

            /* Apply default ruling to data response if needed */
            allow_rq2 = ((!allow_rq2 && !deny_rq2) ->
                    ((DATA_DEFAULT == ALLOW) -> 1 : 0) : allow_rq2);
            deny_rq2 = ((!allow_rq2 && !deny_rq2) ->
                    ((DATA_DEFAULT == ALLOW) -> 0 : 1) : deny_rq2);

            /* Apply default ruling to object request if needed */
            allow_rq3 = ((!allow_rq3 && !deny_rq3) ->
                    ((OBJ_DEFAULT == ALLOW) -> 1 : 0) : allow_rq3);
            deny_rq3 = ((!allow_rq3 && !deny_rq3) ->
                    ((OBJ_DEFAULT == ALLOW) -> 0 : 1) : deny_rq3);

            /* Apply default ruling to data response if needed */
            allow_rq4 = ((!allow_rq4 && !deny_rq4) ->
                    ((OBJ_DEFAULT == ALLOW) -> 1 : 0) : allow_rq4);
            deny_rq4 = ((!allow_rq4 && !deny_rq4) ->
                    ((OBJ_DEFAULT == ALLOW) -> 0 : 1) : deny_rq4);


            /* Global decision
                 - allow if all atomic requests were allowed.
                 - deny if any atomic request was denied */
            RULING = ((allow_rq1 && allow_rq2 && allow_rq3 && allow_rq4 &&
                    !deny_rq1 && !deny_rq2 && !deny_rq3 && !deny_rq4) ->
                    ALLOW : DENY);

            /* Reset DECISION */
            RULING = NIL;

      } /* end atomic */
      od
 }
```

==========

A.2 context-generator.pml

==========

```
/*-------------------------------------------------------------------
   File: context-generator.pml
   Author: Ryan Babbitt, Iowa State University
   Date: November 26, 2006

   Description:
```

```
      This process creates the dynamic aspects of the system, which
   happen to be the context variables that are contained in the privacy
   policy rules. There are two such variables.

-------------------------------------------------------------------------*/

active proctype cg()
 {
      do
      ::    /* cause time to elapse */
            atomic{
                  hour++;

                  if
                  :: (hour < 24) ->
                        skip
                  :: (hour == 24) ->
                        hour=0;
                  fi;

            }

            /* move inhabitant to a room */
      ::
            room = living;
      ::
            room = kitchen;
      ::
            room = bathroom;
      ::
            room = bedroom;
      od;
 }
```

APPENDIX B: MACROS AND PROPERTY FILES

This appendix contains the files used for the specification of our various correctness properties. The file `common_macros` contains definitions of macros used by multiple properties. The file `sat.txt` contains the property specification for the satisfiability of rule contexts. The file `allow_macros` contains definitions used in the property for correctness of allow decisions. The file `allow.txt` contains the actual property specification for allow correctness. The file `deny_macros` contains definitions used in the property for deny correctness. The file `deny.txt` contains the actual property specification for deny correctness. The file `invocation_macros` contains definitions used in the property for service invocation correctness. The file `invocation.txt` contains the actual property specification for invocation correctness. The file `modality_macros` contains definitions used in the property for the absence of modality conflicts. The file `modality.txt` contains the actual property specification for the absence of modality conflicts.

B.1 common_macros

```
/*-------------------------------------------------------------------
   File: common_macros
   Author: Ryan Babbitt, Iowa State University
   Date: November 26, 2006

   Description:
       This file defines macros used in other macros. See comments
   below for more details.
-------------------------------------------------------------------*/

  /* Finds the index with value x in hierarchy z. 255 is returned if the
  index is not found */
  #define INDEX_OF(x, z) ( \
       ((terms.h[(z)].leaves[0].value == (x)) -> 0 : \
       ((terms.h[(z)].leaves[1].value == (x)) -> 1 : \
       ((terms.h[(z)].leaves[2].value == (x)) -> 2 : 255))))

  /* Determines if the value x has ancestor value y in hierarchy z */
  #define HAS_ANCESTOR(x, y, z) ( \
       (terms.h[(z)].leaves[INDEX_OF((y), (z))].value == (x)) || \
       (terms.h[(z)].leaves[INDEX_OF((y), (z))].ancestors[0] == (x)))


  /* Determine the ruling of a particular rule */
  #define ALLOW_DATA(i) (data_policy[(i)].ruling == ALLOW)
  #define DENY_DATA(i) (data_policy[(i)].ruling == DENY)
  #define ALLOW_OBJ(i) (obj_policy[(i)].ruling == ALLOW)
  #define DENY_OBJ(i) (obj_policy[(i)].ruling == DENY)


  /* Is a particular rule contextually enabled? */
  #define ENABLED_DATA(i) (data_policy[(i)].context)
  #define ENABLED_OBJ(i) (obj_policy[(i)].context)


  /* Determines if data policy rule i is applicable to the current
  Data-to-Target request */
  #define APP_DATA_TGT(i) ( \
       HAS_ANCESTOR(data_policy[(i)].domain, services[tgt].domain, \
             DOMAIN_INDEX) && \
       HAS_ANCESTOR(data_policy[(i)].purpose, services[tgt].purpose, \
```

```
            PURPOSE_INDEX) && \
      HAS_ANCESTOR(data_policy[(i)].data, services[tgt].data_in, \
            DATA_INDEX))


  /* Determines if data policy rule i is applicable to the current
  Data-to-Source request */
  #define APP_DATA_SRC(i) ( \
      HAS_ANCESTOR(data_policy[(i)].domain, services[src].domain, \
            DOMAIN_INDEX) && \
      HAS_ANCESTOR(data_policy[(i)].purpose, services[src].purpose, \
            PURPOSE_INDEX) && \
      HAS_ANCESTOR(data_policy[(i)].data, services[tgt].data_out, \
            DATA_INDEX))


  /* Determines if object policy rule i is applicable to the current
  Object-by-Target request */
  #define APP_OBJ_TGT(i) ( \
      HAS_ANCESTOR(obj_policy[(i)].domain, services[tgt].domain, \
            DOMAIN_INDEX) && \
      HAS_ANCESTOR(obj_policy[(i)].purpose, services[tgt].purpose, \
            PURPOSE_INDEX) && \
      HAS_ANCESTOR(obj_policy[(i)].object, services[tgt].object, \
            OBJECT_INDEX))


  /* Determines if object policy rule i is applicable to the current
  Object-by-Source request */
  #define APP_OBJ_SRC(i) ( \
      HAS_ANCESTOR(obj_policy[(i)].domain, services[src].domain, \
            DOMAIN_INDEX) && \
      HAS_ANCESTOR(obj_policy[(i)].purpose, services[src].purpose, \
            PURPOSE_INDEX) && \
      HAS_ANCESTOR(obj_policy[(i)].object, services[tgt].object, \
            OBJECT_INDEX))
```

==========

B.2 sat.txt

==========

```
!([](<>dc0 && <>dc1 && <>dc2 && <>oc0 && <>oc1))

/* "Every rule context is satisfied infinitely often".

Macros dc<i> and oc<j> defined in "src/global.pml"  */
```

==========

B.3 allow_macros

==========

```
/*-------------------------------------------------------------------
   File: allow_macros
   Author: Ryan Babbitt, Iowa State University
   Date: November 26, 2006
   Description:

      According to our evaluation logic, for a service invocation
   request to be allowed, all 4 atomic checks must be allowed. This
```

```
   means that, for each check, 1) a data item or object is null (i.e.
   the request doesn't exist, 2) all applicable allow rules in the
   corresponding policy are enabled and all applicable deny rules in
   the corresponding policy are disabled, or 3) no rule applies and
   the corresponding default ruling is allow.
-----------------------------------------------------------------------*/


  #define NULL_DATA_TGT (services[tgt].data_in == 0)
  #define NULL_DATA_SRC (services[tgt].data_out == 0)
  #define NULL_OBJ_TGT (services[tgt].object == 0)
  #define NULL_OBJ_SRC (services[tgt].object == 0)

  /* Check #1: Data-to-Target */
  #define allow_data_tgt (!(RULING == ALLOW) || (\
       \
       (NULL_DATA_TGT) \
       \
       || \
       \
       ((!(ALLOW_DATA(0) && APP_DATA_TGT(0)) || ENABLED_DATA(0)) && \
       (!(ALLOW_DATA(1) && APP_DATA_TGT(1)) || ENABLED_DATA(1)) && \
       (!(ALLOW_DATA(2) && APP_DATA_TGT(2)) || ENABLED_DATA(2)) && \
       \
       (!(DENY_DATA(0) && APP_DATA_TGT(0)) || !ENABLED_DATA(0)) && \
       (!(DENY_DATA(1) && APP_DATA_TGT(1)) || !ENABLED_DATA(1)) && \
       (!(DENY_DATA(2) && APP_DATA_TGT(2)) || !ENABLED_DATA(2))) \
       \
       || \
       \
       ((DATA_DEFAULT == ALLOW) && \
       !APP_DATA_TGT(0) && \
       !APP_DATA_TGT(1) && \
       !APP_DATA_TGT(2))))

 /* Check #2: Data-to-Source */
 #define allow_data_src (!(RULING == ALLOW) || (\
       \
       (NULL_DATA_SRC) \
       \
       || \
       \
       ((!(ALLOW_DATA(0) && APP_DATA_SRC(0)) || ENABLED_DATA(0)) && \
       (!(ALLOW_DATA(1) && APP_DATA_SRC(1)) || ENABLED_DATA(1)) && \
       (!(ALLOW_DATA(2) && APP_DATA_SRC(2)) || ENABLED_DATA(2)) && \
       \
       (!(DENY_DATA(0) && APP_DATA_SRC(0)) || !ENABLED_DATA(0)) && \
       (!(DENY_DATA(1) && APP_DATA_SRC(1)) || !ENABLED_DATA(1)) && \
       (!(DENY_DATA(2) && APP_DATA_SRC(2)) || !ENABLED_DATA(2))) \
       \
       || \
       \
       ((DATA_DEFAULT == ALLOW) && \
       !APP_DATA_SRC(0) && \
       !APP_DATA_SRC(1) && \
       !APP_DATA_SRC(2))))

  /* Check #3: Object-by-Target */
  #define allow_obj_tgt (!(RULING == ALLOW) || ( \
       \
       (NULL_OBJ_TGT) \
```

```
      \
      || \
      \
      ((!(ALLOW_OBJ(0) && APP_OBJ_TGT(0)) || ENABLED_OBJ(0)) && \
      (!(ALLOW_OBJ(1) && APP_OBJ_TGT(1)) || ENABLED_OBJ(1)) && \
      \
      (!(DENY_OBJ(0) && APP_OBJ_TGT(0)) || !ENABLED_OBJ(0)) && \
      (!(DENY_OBJ(1) && APP_OBJ_TGT(1)) || !ENABLED_OBJ(1))) \
      \
      || \
      \
      ((OBJ_DEFAULT == ALLOW) && \
      !APP_DATA_TGT(0) && \
      !APP_DATA_TGT(1))))

  /* Check #4: Object-by-Source */
  #define allow_obj_src (!(RULING == ALLOW) || ( \
      \
      (NULL_OBJ_SRC) \
      \
      || \
      \
      ((!(ALLOW_OBJ(0) && APP_OBJ_SRC(0)) || ENABLED_OBJ(0)) && \
      (!(ALLOW_OBJ(1) && APP_OBJ_SRC(1)) || ENABLED_OBJ(1)) && \
      \
      (!(DENY_OBJ(0) && APP_OBJ_SRC(0)) || !ENABLED_OBJ(0)) && \
      (!(DENY_OBJ(1) && APP_OBJ_SRC(1)) || !ENABLED_OBJ(1))) \
      \
      || \
      \
      ((OBJ_DEFAULT == ALLOW) && \
      !APP_DATA_SRC(0) && \
      !APP_DATA_SRC(1))))

  /* Aggregated allow logic */
  #define all_allow_logic (allow_data_tgt && allow_data_src && \
      allow_obj_tgt && allow_obj_src)
```

==========

B.4 allow.txt

==========

```
!([] (all_allow_logic))

/* Macro "all_allow_logic" is defined in "macros/allow_macros" */
```

==========

B.5 deny_macros

==========

```
/*-------------------------------------------------------------
   File: deny_macros
   Author: Ryan Babbitt, Iowa State University
   Date: November 26, 2006
   Description:

      According to our evaluation logic, for a service invocation
   request to be denied, at least one of the 4 atomic checks must be
   denied. This means that, for some check, 1) some applicable deny
   rule in the corresponding policy is disabled, or 2) no rule
```

```
  applies and the corresponding default ruling is deny.
-----------------------------------------------------------------------*/

/* Check #1: Data-to-Target */
#define deny_data_tgt (!(RULING == DENY) || ( \
      \
      ((!(DENY_DATA(0) && APP_DATA_TGT(0)) || ENABLED_DATA(0)) || \
      (!(DENY_DATA(1) && APP_DATA_TGT(1)) || ENABLED_DATA(1)) || \
      (!(DENY_DATA(2) && APP_DATA_TGT(2)) || ENABLED_DATA(2)) || \
      \
      (!(ALLOW_DATA(0) && APP_DATA_TGT(0)) || !ENABLED_DATA(0)) || \
      (!(ALLOW_DATA(1) && APP_DATA_TGT(1)) || !ENABLED_DATA(1)) || \
      (!(ALLOW_DATA(2) && APP_DATA_TGT(2)) || !ENABLED_DATA(2)) || \
      \
      ((DATA_DEFAULT == DENY) && \
       !APP_DATA_TGT(0) && \
       !APP_DATA_TGT(1) && \
       !APP_DATA_TGT(2)))))

/* Check #2: Data-to-Source */
#define deny_data_src (!(RULING == DENY) || ( \
      \
      ((!(DENY_DATA(0) && APP_DATA_SRC(0)) || ENABLED_DATA(0)) || \
      (!(DENY_DATA(1) && APP_DATA_SRC(1)) || ENABLED_DATA(1)) || \
      (!(DENY_DATA(2) && APP_DATA_SRC(2)) || ENABLED_DATA(2)) || \
      \
      (!(ALLOW_DATA(0) && APP_DATA_SRC(0)) || !ENABLED_DATA(0)) || \
      (!(ALLOW_DATA(1) && APP_DATA_SRC(1)) || !ENABLED_DATA(1)) || \
      (!(ALLOW_DATA(2) && APP_DATA_SRC(2)) || !ENABLED_DATA(2)) || \
      \
      ((DATA_DEFAULT == DENY) && \
       !APP_DATA_SRC(0) && \
       !APP_DATA_SRC(1) && \
       !APP_DATA_SRC(2)))))

/* Check #3: Object-by-Target */
#define deny_obj_tgt (!(RULING == DENY) || ( \
      \
      ((!(DENY_OBJ(0) && APP_OBJ_TGT(0)) || ENABLED_OBJ(0)) || \
      (!(DENY_OBJ(1) && APP_OBJ_TGT(1)) || ENABLED_OBJ(1)) || \
      \
      (!(ALLOW_OBJ(0) && APP_OBJ_TGT(0)) || !ENABLED_OBJ(0)) || \
      (!(ALLOW_OBJ(1) && APP_OBJ_TGT(1)) || !ENABLED_OBJ(1)) || \
      \
      ((OBJ_DEFAULT == DENY) && \
      !APP_OBJ_TGT(0) && \
      !APP_OBJ_TGT(1)))))

/* Check #4: Object-by-Source */
#define deny_obj_src (!(RULING == DENY) || ( \
      \
      ((!(DENY_OBJ(0) && APP_OBJ_SRC(0)) || ENABLED_OBJ(0)) || \
      (!(DENY_OBJ(1) && APP_OBJ_SRC(1)) || ENABLED_OBJ(1)) || \
      \
      (!(ALLOW_OBJ(0) && APP_OBJ_SRC(0)) || !ENABLED_OBJ(0)) || \
      (!(ALLOW_OBJ(1) && APP_OBJ_SRC(1)) || !ENABLED_OBJ(1)) || \
      \
      ((OBJ_DEFAULT == DENY) && \
      !APP_OBJ_SRC(0) && \
      !APP_OBJ_SRC(1)))))
```

```
 /* Aggregated deny logic */
 #define all_deny_logic (deny_data_tgt || deny_data_src || \
       deny_obj_tgt || deny_obj_src)
```

==========

B.6 deny.txt

==========

```
!([](all_deny_logic))

/* Macro "all_deny_logic" is defined in "macros/deny_macros" */
```

==========

B.7 invocation_macros

==========

```
/*-------------------------------------------------------------------
   File: invocation_macros
   Author: Ryan Babbitt, Iowa State University
   Date: November 26, 2006

   Description:
       This file defines macros for allowable service invocations.
   These can be used to make sure that the allowable service invocations
   happen in conjunction with correct evlaution logic (defined in
   allow_macros and deny_macros). All macros are of the form src"i"tgt"j",
   where i is the source service/method index and j is the target
   service/method index. Basically, in our example scenario, the two
   monitoring services can invoke all other services.
-------------------------------------------------------------------*/

 #define src4tgt0 ((RULING == ALLOW) && (src == 4) && (tgt == 0))
 #define src4tgt1 ((RULING == ALLOW) && (src == 4) && (tgt == 1))
 #define src4tgt2 ((RULING == ALLOW) && (src == 4) && (tgt == 2))
 #define src4tgt3 ((RULING == ALLOW) && (src == 4) && (tgt == 3))
 #define src5tgt0 ((RULING == ALLOW) && (src == 5) && (tgt == 0))
 #define src5tgt1 ((RULING == ALLOW) && (src == 5) && (tgt == 1))
 #define src5tgt2 ((RULING == ALLOW) && (src == 5) && (tgt == 2))
 #define src5tgt3 ((RULING == ALLOW) && (src == 5) && (tgt == 3))
```

==========

B.8 inv.txt

==========

```
!([](all_allow_logic && all_deny_logic && <>src4tgt0 && <>src4tgt1 && <>src4tgt2 &&
<>src4tgt3 && <>src5tgt0 && <>src5tgt1 && <>src5tgt2 && <>src5tgt3))

/* Macro "all_allow_logic" defined in "macros/allow_macros".
   Macro "all_deny_logic" defined in "macros/deny_macros".
   Macros src<i>tgt<j> defined in "macros/invocation_macros". */
```

==========

B.9 modality_macros

==========

```
/*-------------------------------------------------------------------
    File: modality_macros
```

```
    Author: Ryan Babbittt, Iowa State University
    Date: November 30, 2006

    Description:
       This file defines a test for atomic request modality conflicts.
    Three conditions need to be satisfied:
       1) The rulings are opposite
       2) All three terms have a common leaf descendant
       3) The context of one is the subset of the context of the other.
          (not specified here)
-------------------------------------------------------------------------------*/


/* Do data rules i and i create a modality conflict? */
  #define MODAL_DATA_PAIR(i,j) ( \
       ((ALLOW_DATA((i)) && DENY_DATA((j))) || \
        (DENY_DATA((i)) && ALLOW_DATA((j)))) \
       \
       && \
       \
        (HAS_ANCESTOR(data_policy[(i)].domain, data_policy[(j)].domain, \
              DOMAIN_INDEX) || \
         HAS_ANCESTOR(data_policy[(j)].domain, data_policy[(i)].domain, \
              DOMAIN_INDEX)) \
       \
       && \
       \
       (HAS_ANCESTOR(data_policy[(i)].purpose, data_policy[(j)].purpose, \
              PURPOSE_INDEX) || \
       HAS_ANCESTOR(data_policy[(j)].purpose, data_policy[(i)].purpose, \
              PURPOSE_INDEX)) \
       \
       && \
       \
       (HAS_ANCESTOR(data_policy[(i)].data, data_policy[(j)].data, \
              DATA_INDEX) || \
       HAS_ANCESTOR(data_policy[(j)].data, data_policy[(i)].data, \
              DATA_INDEX)))

/* All possible pairs of data rules */
  #define mdp01 (MODAL_DATA_PAIR(0,1))
  #define mdp02 (MODAL_DATA_PAIR(0,2))
  #define mdp12 (MODAL_DATA_PAIR(1,2))

/* Do object rules i and i create a modality conflict? */
  #define MODAL_OBJ_PAIR(i,j) ( \
       ((ALLOW_OBJ((i)) && DENY_OBJ((j))) || \
       (DENY_OBJ((i)) && ALLOW_OBJ((j)))) \
       \
       && \
       \
        (HAS_ANCESTOR(obj_policy[(i)].domain, obj_policy[(j)].domain, \
              DOMAIN_INDEX) || \
         HAS_ANCESTOR(obj_policy[(j)].domain, obj_policy[(i)].domain, \
              DOMAIN_INDEX)) \
       \
       && \
       \
       (HAS_ANCESTOR(obj_policy[(i)].purpose, obj_policy[(j)].purpose, \
              PURPOSE_INDEX) || \
       HAS_ANCESTOR(obj_policy[(j)].purpose, obj_policy[(i)].purpose, \
```

```
            PURPOSE_INDEX)) \
      \
      && \
      \
      (HAS_ANCESTOR(obj_policy[(i)].object, obj_policy[(j)].object, \
            OBJECT_INDEX) || \
      HAS_ANCESTOR(obj_policy[(j)].object, obj_policy[(i)].object, \
            OBJECT_INDEX)))

/* All possible pairs of object rules */
  #define mop01 (MODAL_OBJ_PAIR(0,1))
```

==========

B.10 modality.txt

==========

```
!((mdp01 -> (<>(!dc0 && dc1) && <>(dc0 && !dc1))) && (mdp02 -> (<>(!dc0 && dc2) &&
<>(dc0 && !dc2))) && (mdp12 -> (<>(!dc1 && dc2) && <>(dc1 && !dc2))) && (mop01 ->
(<>(!oc0 && oc1) && <>(oc0 && !oc1))))

/* Three parts to a modality conflict
     1) Rules are opposing
     2) All corresponding terms have a common leaf
     2) The context of one rule is a subset of the context of the other
        rule.

   Macros mdp<i><j> defined in "macros/modality_macros".
   Macros dc<k> defined in "src/global.pml"
*/
```