The Computer Journal Advance Access published September 7, 2012

doi:10.1093/comjnl/bxs102

Dynamic Access Control Policies: Specification and Verification

H. JANICKE*, A. CAU, F. SIEWE AND H. ZEDAN

Software Technology Research Laboratory, De Montfort University, LE1 9BH Leicester, UK
**Corresponding author: heljanic@dmu.ac.uk*

Security requirements deal with the protection of assets against unauthorized access (disclosure or modification) and their availability to authorized users. Temporal constraints of history-based access control policies are difficult to express naturally in traditional policy languages. We propose a compositional formal framework for the specification and verification of temporal access control policies for security critical systems in which history-based policies and other temporal constraints can be expressed. In particular, our framework allows for the specification of policies that can change dynamically in response to time or events enabling dynamic reconfiguration of the access control mechanisms. The framework utilizes a single well-defined formalism, interval temporal logic, for defining the semantics of these policies and to reason about them. We illustrate our approach with a detailed case study of an electronic paper submission system showing the compositional verification of their safety, liveness and information flow properties.

Keywords: access control; policy; compositional specification; semantics; verification; interval temporal logic

Received 1 December 2011; revised 4 April 2012
Handling editor: Albert Levi

1. INTRODUCTION

Government and non-Government organizations alike are faced with increasing amounts of digital information assets that are routinely communicated to a constantly changing number of employees or to other collaborating organizations. Whilst the importance of protecting information against unauthorized access is widely recognized, there is evidence that the need to share information efficiently can lead to a relaxation of access control restrictions to the information—resulting in inappropriate levels of protection.

In part, this is caused by the increased complexity of managing and implementing security policies. The larger the number of resources and the larger the number of accessing subjects become the more complex will be the corresponding access control policies. Security research has long addressed this issue by providing abstractions such as role-based access control [1] or attribute-based access control [2], where authorizations do not depend directly on the subject or object but its role or attributes. Such abstractions can greatly reduce the complexity of authorizations.

However, abstraction only partly addresses the dynamics of today's information systems and their inter-connectivity. Abstraction allows one to deal with the dynamic addition/removal of subjects, respectively, objects in the information system, without having to redefine the systems security policies. In a corporate world where the (temporary) collaboration between organizations or government bodies is becoming the norm, access control mechanisms must be able to adapt quickly to changing requirements with respect to the sharing of information. One approach to allow for systems to adapt rapidly to new requirements is policy-based management [3, 4].

Policy-based management is a modular and scalable approach that greatly improves the maintainability of information systems, with respect to security, in comparison to more commonly deployed ad hoc implementations of security requirements [4]. Security policies describe constraints on the usage of the information system, that the underlying system must implement. The integration of dedicated security mechanisms in the system that interpret the policies ensures compliance with the security requirements from which the policy specification was derived. This strict separation of concerns means that any change in the security requirements will be reflected in the evolution of the policy and does not require any modification to the implementation of the system.

Building on the policy-based management approach, we consider policies themselves to be dynamic. By dynamic policies, we mean policies that define decisions on the basis of the system state or changes in the system state. These changes are triggered by the system, or as side-effects of access requests. Dynamic policies therefore address anticipated changes in policies that are part of the policy specification. In contrast to this is the evolution of policies, which addresses the change of a policy due to changing requirements, which are triggered through administrative processes, that replace or modify existing policies.

The security policies themselves are an invaluable asset to an organization as they concisely express the underlying protection requirements and represent a cornerstone in any security assessment of the information system. In security critical applications, viz. applications where the compromise of the information system endangers the life or livelihood of people, it is paramount that the language to express policies has a sound and formal basis that can be used for the analysis and proof of properties such as conflicts, completeness or information-flow.

Sometimes requirements refer to the execution history [5], this means that a policy decision can depend on the system behaviour that was observed in the past. This has been shown to be more expressive than traditional mechanisms such as stack inspections in [5], and is now supported by many policy languages, see, e.g. [3] for an overview. A standard example of such a stateful policy is the Chinese Wall Policy [6] in which access to a resource depends on choices made earlier.

Whilst formal approaches to the modelling of security policies have a long history [7–10], many of today's policy languages [11–13] are developed using good software engineering practices alone. Logic and the formal semantics of policy languages play an essential role in providing the level of assurance that is needed for a core component of the security infrastructure that we are putting in charge with the protection of privacy of millions of citizens [14, 15]. This assurance can be obtained by formally verifying security and safety properties of the security policy itself and the mechanisms enforcing the policy [16–19].

Often formal specification and verification methods [20, 21] do not scale up to the size of real-world systems in which thousands of policy rules govern the access to corporate resources. To make matters worse, often policy languages require security professionals to translate high-level security requirements into the relatively low-level syntax of the policy language.

The objective of our work are therefore:

(i) To provide a high-level specification language that closes the gap between informal security requirements and executable policies.
(ii) To provide a formal semantics of the policy language, such that properties of policies can be verified.
(iii) To maintain a compositional approach to specification and verification that addresses the complexity of today's security requirements.

The contribution of this research is 3-fold:

First, history-based requirements are captured in the premise of policy rules using a high-level temporal description of a system behaviour that triggers the rule. The ability to express dynamic policies using descriptions of behaviours removes a level of abstraction that is needed when dynamic policies are defined in other well-known languages such as UCON [22] or in [19], that introduce additional mutable attributes in the policy to capture policies that depend on the history of the system's execution.

Secondly, policies can be composed sequentially and thus change over time and on the occurrence of events. The mixture between declarative specifications that is traditionally used in rule-based policy systems to express a policy and the ability to define sequential changes of these policies provides policy authors with more flexibility to express their requirements. We present in this paper a medium-sized case-study showing the benefits of our policy specification approach.

Thirdly, we present a compositional specification approach that allows for the decomposition of specification and verification tasks into smaller sub-tasks, making the approach modular and more manageable. Using the underlying formal semantics of policies, we also contribute in this paper a set of proof rules that are useful for often recurring verification tasks. Automated verification approaches frequently suffer from the problem of state-explosion and do not scale to the size of large policy specifications. We developed compositional proof rules as a technique to decompose the verification problem so that automated approaches can be more effectively applied to smaller sub-problems. We show the application of proof rules using the policy specification developed for the case-study, where we show how safety, liveness and information flow properties can be proved in a compositional manner. In particular, we are concerned with the notion of *Allowed Information Flows*, that is, flows of information caused by a sequence of accesses that are permitted under a given dynamic access control policy.

The work presented here is an extension of earlier work on compositional policy specifications [23, 24], that removes some major restrictions in the formulae that can be used to describe behaviour. Concretely, this means that we now allow for the negation of subformulae in the premise of policy rules, previously only state-formulae could be negated. We changed the semantics of policy rules to allow for more concise specifications of behaviours that trigger access control decisions and improved on the way free variables are bound in policy rules. Most importantly, we provide a set of compositional proof rules that allow one to prove safety, liveness and information flow properties of policies and show how these can be applied using a case study.

The remainder of this paper is structured as follows. We review related work in Section 2 and compare existing work with our approach and contributions. In Section 3, we provide an informal description of the underlying computational model on which the specification of our policies is based and put the model into the context of the well-known policy decision point (PDP)/policy enforcement point (PEP) model. As part of the preliminaries we also present an introduction to interval temporal logic (ITL). In Section 4, we introduce the SANTA policy language and provide examples of policy rules and policy compositions. In Section 5, we give a formal semantics to SANTA and define the concept of allowed information flow. The policy language is then used in Section 6 to define access control requirements of an electronic paper submission (EPS) system. In Section 7, we show how safety, liveness and information flow properties are expressed and can be verified. We give examples that demonstrate the compositional verification of policies. We conclude the paper in Section 8 where we also discuss some of the future work.

2. RELATED WORK

Policy-based management is increasingly used in the administration of distributed information systems. Several industrial strength frameworks such as Ponder [12, 13], XACML [11] and KAOS [25] are available. The role that formal modelling and logic plays in the development of these languages is widely recognized in the field, e.g. [14, 21, 26, 27]. It is even argued that the development of any policy language should be based on a formal model to avoid ambiguities and contradictions within the language [15]. The selection of the formalism used is tightly linked with the desired expressiveness of the language and the computational model of the system in which the language is applied to. Uszok *et al.* [28] place their policy language in the semantic web domain and opted for a description logic semantics for KAOS to analyse relationships between entities of their policies.

There is an agreement in the policy community that today's complex protection requirements require policies to be stateful, viz. policy decisions depend on the current state of the system. For some systems, this state is an explicit part of the trusted computing base, e.g. stack-based models [29], role-based access control [1] or multi-level security [30]. All these approaches define a static security state, viz. one that is changed only by active participation of an administrator. For others state can be defined in form of mutable attributes [31] as part of access control policies [22, 32].

Other models base policy decisions on the execution history [5], where policy decision can depend on the system behaviour that was observed in the past. This has been shown to be more expressive than traditional mechanisms such as stack inspections in [5], and is now supported by many policy languages, see, e.g. [3] for an overview. A standard example of a stateful policy is the Chinese Wall Policy [6]. More recently, the emphasis on stateful policies has been reinforced by work on *dynamic* policies. Similar to the notion of mutable attributes in UCON here the policy can depend on the state of the protected system [18, 19].

Policies constrain the behaviour of reference monitors in information systems. Reference monitors are a widely known model for access control [33, p. 25] which are also a compulsory part of the Trusted Computer System Evaluation Criteria (TCSEC) [34]. More precisely, *access control* policies determine the choice of the reference monitor to permit or deny the execution of a request. A complete specification of the reference monitor can be given in form of an access control matrix [35], which fully determines the access rights at any point in time during the system execution. As we are interested in history-based access control [5] in order to express dynamic policies, this matrix will depend not only on the current state of the information system, but also on the history of execution.

The potential of temporal logic specifications has been noted by Calo *et al.* [27] and has successfully been applied in areas where dynamic properties are relevant, e.g. [36]. Also the UCON model has been formalized by Zhang [37] using temporal logic of actions (TLA) [38] and revisited in [39]. We use ITL instead of TLA, as TLA specifications are at a comparatively low, operational level (essentially defining an automaton), whereas ITL constructs are more specification oriented. In particular, the 'chop' operator is needed to express sequential phases which would result in complex TLA formulae due to its level of abstraction. Formalizations of UCON using TLA still needed to augment the specification with mutable attributes to capture history dependent requirements [37, 39] and explicitly maintain this as part of the system state. In our approach, this is not necessary as these information can be implicitly encoded as part of the policy-rule specification. Temporal logic specifications generally have the advantage that temporal properties of access control policies, e.g. describing effects that sequences of access have on access control decisions, can be expressed succinctly. Other approaches utilize constraint Datalog [40] to specify and reason about access control policies, e.g. [19, 20, 26] or process algebra [41].

Other work [42, 43] has recognized the need for more expressive access control policies to capture the temporal dimension of access control, however, these models lack compositionality. By compositionality, we mean that the overall security policy can be composed of smaller policies. Compositional specification and composition of policies originating from various domains has been widely addressed [26, 44–47]. The focus here is predominantly to detect and remove conflicts between the composed policies. In comparison, our approach adds the sequential composition of policies and addresses the problem of how policy changes triggered by system events or due to administrative changes impact on the system security. The novelty of the presented model is that it allows for the sequential composition to account for the dynamic change of policies over time and on the occurrence of events.

Other policy languages, e.g. [13], treat policies themselves as managed objects and allow them to be enabled and disabled using obligation rules. This requires a much more detailed understanding of the interaction between policies than is required in our compositional specification. To achieve the same effect with an (de-)activation-based model the policy would have to contain additional obligations rules that disable and enable policies, thus increasing the number of rules contained in the policy in comparison. Alternatively activations can be included in the specification at rule level as part of the premise of policy rules. This approach, however, has the disadvantage that all rules reside in a single, large policy that is difficult to understand and maintain. An analogy to programming would be the use of functions or methods. Similar to functions, policies are logical constructs that are defined by the rules they contain and carry a meaning at a higher level of abstraction. Of course, any programme can be written without the use of functions by replacing every function call with the statements contained in that function. However, the opportunity for reuse and the ease with which such programs can be understood is limited. By providing sequential policy composition as part of our policy language, we employ a similar strategy as is used in procedural programming. This allows policy authors to decide whether to specify policies declaratively, i.e. rules that all apply at the same time, or procedural, i.e. policies that change sequentially over time.

The specification using sequential composition can easily be translated into activation and deactivation rules. However, it expresses the mutual exclusion of both policies at a higher abstraction level and is much closer to the original requirement from which we derived the policy specification. We believe therefore that our approach considerably reduces the potential for specification errors related to policy activations and deactivations.

We adopted the approach of Jajodia *et al.* [26] and specify policies as hybrid policies, i.e. policies that contain both positive authorizations (permissions) and negative authorizations (denials). Naturally conflicts between the two types can occur, when both positive and negative rules apply in an access control decision. These conflicts need to be detected and resolved. The detection of conflicts also has been the subject of study in [48–50]. This requires the analysis of the policy rules to establish if any two rules can 'fire' (be applied) at the same time, which can be expressed as a safety property. We show in Section 1 how safety properties can be verified. In contrast to their work the premises of rules presented in this paper are capturing history-based policies and therefore require the analysis of behaviours rather than exclusion checks on Boolean expressions. Our contribution focuses on a compositional approach to verification of policy properties, such as safety, liveness and information flow. Whilst the development of fully automated verification algorithms is feasible, these do not scale well with the number of policy rules—in particular, when these are history dependent. Instead, we advocate a two-stage approach that decomposes the verification problem using the compositional proof-rules we present in Section 7 and applies the automated approach Cau *et al.* (2012, submitted) to the smaller sub-problems, e.g. simple policies as presented in Section 4.

To resolve conflicts, Woo *et al.* [51] proposed *default rules* as a way to provide complete specifications. The drawback of this approach is that default rules might not be conclusive. As a consequence, the model can lead to a situation in which an authorization request has no answer. Logic-based approaches [52–55] restrict the policy language to a variant of Datalog [56] to overcome this problem using the *closed world assumption* [17, 57], which says that if a positive literal cannot be proved to be true then it is assumed to be false. This approach to conflict resolution has been widely explored (e.g. [21, 26] and is implemented amongst others in [11]). We adopt this notion of default rules, in that we show in Section 5 that a policy can be automatically rewritten into a complete specification, in which every access decision is defined, by adding default rules. We also show that the rewritten policy is a formal refinement, i.e. that the rewritten policy is stronger than the original specification.

During the verification of policies we also consider information flow. Information flow analysis is an important step to check and verify the information security for a given system [9, 58–60].

Language-based information flow analysis [61], analyses the potential flows of the information occasioned by the execution of source operations. Using such techniques, it is possible to determine whether such a flow violates a given information flow security policy. The aim of the information flow analysis is to infer the information dependencies and the relationship between the programme variables. Information flow analysis is usually used to check or verify that the target programme is free of *all* undesired information flows, i.e. that the programme is secure with respect to a given information flow policy.

Static information flow analysis does not require the system to be executed or operated. The majority of information flow analysis approaches are static and determine whether a given program 'text' obeys some predefined policy with respect to an information flow without running the programme [61, 62]. Goguen and Meseguer [63] define the notion of non-interference as a way of handling the occurrence of *illegitimate* information flow in a system specification.

Semantic approaches to information flow analysis are concerned with controlling information flow based on semantic security models that control information flow in terms of programme behaviour [59, 64–66]. Leino and Joshi [67] provided a new technique that statically analyses secure information flow based on a semantic notion of programme equality.

Policies control the behaviour of programmes and represent security requirements that have been separated from the system implementation to allow for easier evolution of the requirements, only requiring the policy to be modified, but not the underlying system implementation. In comparison to the related work on information flow analysis, this work considers information flow analysis based on policies that control the system behaviour. Whilst at this level of abstraction no guarantee can be given that the underlying system implementation is free of undesirable information flows, it does allow an information flow analysis based on the constraints specified in the policy. Under the assumption that information flows only take place through read and write actions that are controlled by the access control policy, an analysis at this level can establish the *allowed* information flows with respect to the policy. In combination with the verification of this assumption using the reviewed static analysis techniques this has the advantage that changes in policy do not require a complete information flow analysis of the system, but can be limited to the policies themselves.

3. PRELIMINARIES

This section introduces the computational model which is an abstraction of the well-known PDP/PEP model for policy-based management. We also introduce the reader to ITL, which is the formal foundation of the SANTA policy language.

3.1. Computational model

The computational model describes the entities that comprise the system, their behaviours and interactions. It represents a suitable abstraction for many real-world implementations that use the PDP/PEP architecture [11, 68] to implement policy-based management. For the purpose of specification, verification and analysis of dynamic security policies, the externally observable behaviour of a system, i.e. the sequence of actions it does perform, is sufficient. We therefore refrain from modelling implementation details of the domain-dependent interactions between users and system.

We distinguish three different entities in the system: *subjects*, *objects* and *reference monitors*. *Subjects* are entities, that (pro-)actively perform actions that affect objects. A subject can be a human user, a group or role or a programme acting on behalf of a user. *Objects* are passive entities, that represent shared data-structures in the information system. *Reference monitors* control the access to objects and determine whether a specific action can be performed by a subject or not. The concrete conditions under which a reference monitor *permits* an execution request or *denies* it are specified in the security policy.

The security policy represents an abstract specification of constraints on the interactions between the subjects and objects in the system. The abstract specification is then constructively refined into the behaviour of the reference monitor such that the overall system satisfies the policy. Proving properties of the policy means that these properties are preserved by the system if the implementation of the reference monitor is *correct*, viz. it is not by-passable and adhering to the constraints specified in the policy. We will show in Section 7 a set of compositional proof rules to check properties of the constraints expressed in the policy.

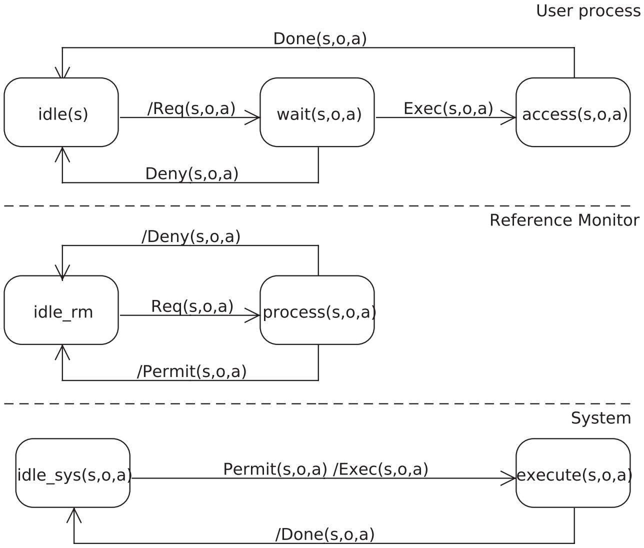


FIGURE 1. Computational model.

The behaviour of a reference monitor and its interaction with the other system components are detailed in Fig. 1 as a Statechart [69]. For simplicity, the reference monitor is mediating at most one request at a time, i.e. all access requests are interleaved. The concurrent enforcement of policies introduces additional complexities related to the atomicity of requests and functions performed by the reference monitor. This is the subject of our ongoing research and out of the scope of this paper, however, we refer the interested reader to initial results presented in [70].

3.1.1. *User model.*

Every subject *s* represents a process (see user process in Fig. 1) acting on behalf of a user. This process can be either in a state *idle*, *wait* or *access*. Initially, we assume a user process *s* to be in its *idle(s)* state.[1] By raising the event *Req*(*s,o,a*), the process *s* indicates that it requests the execution of action *a* on the system object *o* and transitions to the state *wait*(*s,o,a*). It will remain in the waiting state until its is either denied (event *Deny*(*s,o,a*)) or the request is executed (event *Exec*(*s,o,a*)) before transitioning into the state *access*(*s,o,a*).

3.1.2. *Reference monitor model.*

The reference monitor (RM), as depicted in Fig. 1, is a process that is initially in its *idle_rm* state. Upon a user request

[1] Notation: *idle(s)* represents a parametrized state, viz. for every subject *s* there exists an idle state. A similar convention applies to events. *o* ranges of the set of all objects and *a* over the set of actions that can be performed on the object *o*.

	Expressions
$e ::=$	$\mu \mid a \mid A \mid g(e_1, \ldots, e_n) \mid \bigcirc v \mid \mathsf{fin}\ v$
	Formulae
$f ::=$	$p(e_1, \ldots, e_n) \mid \neg f \mid f_1 \wedge f_2 \mid \forall v \bullet f \mid$ $\mathsf{skip} \mid f_1 \,;\, f_2 \mid f^*$

FIGURE 2. Syntax of ITL.

Req(*s,o,a*) the RM transitions into the state *process*(*s,o,a*) in which its behaviour is specified by the policy. If the policy grants access it will raise the event *Permit*(*s,o,a*); if it denies the access, the event *Deny*(*s,o,a*) is raised. The RM subsequently returns to its *idle_rm* state.

3.1.3. System model.

The access to the objects is facilitated by the system process, depicted in Fig. 1. We assume that the system is initially in the state *idle_sys*(*s,o,a*). On the event that the controller permits the execution, it will transition to the state *execute*(*s,o,a*) and raise the event *Exec*(*s,o,a*) that synchronizes the states *access*(*s,o,a*) of the user process and the state *execute*(*s,o,a*) of the system. The concrete behaviour of the user process and the system in these states are not explicitly defined, however, we will assume for the analysis of information flow (Section 7.3) that every pairing of these states can be characterized into the categories *read*, *write* and *read* + *write*.

The computational model represents a simplification of real information systems, where not only subjects can concurrently make requests, but also the reference monitors and the system facilitating access to the shared objects are distributed and can exhibit concurrent behaviour.

3.2. Interval temporal logic

The key notion of ITL [71] is an *interval*. An interval σ is considered to be a (in)finite sequence of states $\sigma_0, \sigma_1 \ldots$, where a state σ_i is a mapping from the set of variables *Var* to the set of values *Val*. The length $|\sigma|$ of an interval $\sigma_0 \cdots \sigma_n$ is equal to n (one less than the number of states in the interval, so a one state interval has length 0).

The syntax of ITL is defined in Fig. 2 where μ is a constant value, a is a static variable (does not change within an interval), A is a state variable (can change within an interval), v a static or state variable, g is a function symbol and p is a predicate symbol.

The informal semantics of the most interesting constructs are as follows:

(i) skip: unit interval (length 1, i.e., an interval of two states).

(ii) $f_1 ; f_2$: holds if the interval can be decomposed ('chopped') into a prefix and suffix interval, such that f_1 holds over the prefix and f_2 over the suffix, or if the interval is infinite and f_1 holds for that interval. Note the last state of the interval over which f_1 holds is shared with the interval over which f_2 holds. This is illustrated in Fig. 3.

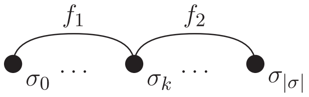


FIGURE 3. Informal semantics of $f_1 ; f_2$.

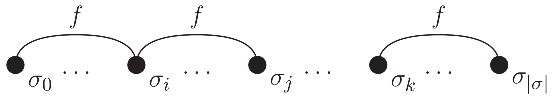


FIGURE 4. Informal semantics of f^*.

(iii) f^*: holds if the interval is decomposable into a finite number of intervals such that for each of them f holds, or the interval is infinite and can be decomposed into an infinite number of finite intervals for which f holds. This is illustrated in Fig. 4.

(iv) $\bigcirc v$: value of v in the next state when evaluated on an interval of length at least one, otherwise an arbitrary value.

(v) $\mathsf{fin}\ v$: value of v in the final state when evaluated on a finite interval, otherwise an arbitrary value.

3.2.1. Derived constructs.

The following lists some of the derived constructs used in the remainder of this paper. The Boolean operators $\vee$ (or) and $\supset$ (implication) are derived as usual.

$\bigcirc f \mathrel{\widehat{=}} \mathsf{skip} \,;\, f$ next f, f holds from the next state. Example: $\bigcirc(X = 1)$: Any interval such that the value of X in the second state is 1 and the length of that interval is at least 1.

$\mathsf{more} \mathrel{\widehat{=}} \bigcirc\mathsf{true}$ non-empty interval, i.e. any interval of length at least 1.

$\mathsf{empty} \mathrel{\widehat{=}} \neg\,\mathsf{more}$ interval, i.e. any interval of length 0 (just one state).

$\mathsf{inf} \mathrel{\widehat{=}} \mathsf{true} \,;\, \mathsf{false}$ infinite interval, i.e. any interval of infinite length.

$\mathsf{finite} \mathrel{\widehat{=}} \neg\,\mathsf{inf}$ finite interval, i.e. any interval of finite length.

$\Diamond f \mathrel{\widehat{=}} \mathsf{finite} \,;\, f$ sometimes f, i.e. any interval such that f holds over a suffix of that interval. Example: $\Diamond X \neq 1$: Any interval such that there exists a state in which X is not equal to 1.

$\Box f \mathrel{\widehat{=}} \neg \Diamond \neg f$ always f, i.e. any interval such that f holds for all suffixes of that interval. Example: $\Box(X = 1)$: Any interval such that the value of X is equal to 1 in all states of that interval.

$\boxed{i}\ f \mathrel{\widehat{=}} \neg(\neg\, f \;;\; \mathsf{true})$ box-i, i.e. any interval such that f holds over all prefix sub-intervals.

$\boxed{a}\ f \mathrel{\widehat{=}} \neg(\mathsf{finite} \;;\; \neg\, f \;;\; \mathsf{true})$ box-a, i.e. any interval such that f holds over all sub-intervals.

$\mathsf{fin}\ f \mathrel{\widehat{=}} \Box(\mathsf{empty} \supset f)$ final state, i.e. any interval such that f holds in the final state of that interval.

$\exists v \bullet f \mathrel{\widehat{=}} \neg\, \forall v \bullet \neg\, f$ Existential quantification.

$$f^n \mathrel{\widehat{=}} \begin{cases} \mathsf{false} & \textbf{if } n < 0, \\ \mathsf{empty} & \textbf{if } n = 0 \\ f \;;\; f^{n-1} & \textbf{if } n > 0. \end{cases} \qquad f \text{ repeats } n \text{ times}$$

$\mathsf{len}(e) \mathrel{\widehat{=}} \mathsf{skip}^e$ holds if the length of the interval is e.

4. SANTA POLICY LANGUAGE

We use a rule-based approach and specify (sets of) access control rights in terms of policy rules and their compositions. A simple access control policy consists of three types of rules: (i) positive authorization rules, (ii) negative authorization rules and (iii) decision rules. These simple access control policies can then be combined into composite policies. In the following, we provide the syntax of access control policies and examples of their usage.

Figure 5 summarizes the syntax of our policy language where e is an expression, *be* a Boolean expression, and *se* a Set expression with their usual operators and semantics. $\mathbf{S}_i$ is a subject variable, where i is a arbitrary name, similarly **O** is an object variable, **A** is an action variable and `pn` is a name for a policy; `rn` is a name for a rule (optional). Let $Subjects$, $Objects$ and $Actions$ be, respectively, the universal set of subjects, objects and actions. These can be used as part of SANTA expressions. Let $cs \in Subjects$ be a subject, $co \in Object$ be an object and $ca(\bar{v}) \in Actions$ be an action with interface $\bar{v}$. The following subsections will explain the syntax informally.

4.1. Policy rules

A rule typically expresses a single security requirement and forms the basic building block of a policy. Rules consists of a *premise* and a *consequence*. The premise describes a set of system behaviours, which lead to the consequence that represents an assertion on the current system state, such as allowing or denying a particular access. The consequence of a rule defines the decision taken by the reference monitor. The set of system behaviours in the premise is matched against the history of the system execution. Rules therefore can refer to sequences of previously observed states in the system execution, allowing for the expression of history-based policies [5] and dynamic separation of duty constraints [72]. Events that can be referred to in the premise of rules are those defined in the computational model (see Fig. 1) or external events that are observable by the RM process.

$$\begin{array}{|ll|}
\hline
\multicolumn{2}{|c|}{\textit{Subjects}} \\
su ::= & \mathsf{S}_i \mid cs \\
\hline
\multicolumn{2}{|c|}{\textit{Objects}} \\
ob ::= & \mathsf{O}_i \mid co \\
\hline
\multicolumn{2}{|c|}{\textit{Actions}} \\
ac ::= & \mathsf{A}_i \mid ca(e_1, \ldots, e_n) \\
\hline
\multicolumn{2}{|c|}{\textit{Premise of rule}} \\
pr ::= & pr_1 \;\textbf{;}\; pr_2 \mid pr_1 \ \textbf{and}\ pr_2 \mid pr_1 \ \textbf{or}\ pr_2 \mid \\
 & \textbf{always}\ pr \mid \textbf{sometime}\ pr \mid \textbf{not}\ pr \mid \\
 & \textbf{if}\ be\ \textbf{then}\ pr_1\ \textbf{else}\ pr_2 \mid \textbf{exists}\ x\ \textbf{in}\ se\ \textbf{:}\ pr \mid \\
 & \textbf{forall}\ x\ \textbf{in}\ se\ \textbf{:}\ pr \mid \textbf{last}(\mathrm{e}) : pr \mid e\ \textbf{:}\ pr \mid be \\
\hline
\multicolumn{2}{|c|}{\textit{Rules}} \\
ru ::= & [\mathrm{rn}\ \textbf{::}]\ \textbf{allow}\ (su, ob, ac)\ \textbf{when}\ pr \mid \\
 & [\mathrm{rn}\ \textbf{::}]\ \textbf{deny}\ (su, ob, ac)\ \textbf{when}\ pr \mid \\
 & [\mathrm{rn}\ \textbf{::}]\ \textbf{decide}\ (su, ob, ac)\ \textbf{when}\ pr \\
\hline
\multicolumn{2}{|c|}{\textit{Policies}} \\
po ::= & ru_1 \ldots ru_n \mid \textbf{policy}\ \mathrm{pn}\ \textbf{::}\ po\ \textbf{end} \mid \\
 & po_1 \;\textbf{;}\; po_2 \mid \textbf{aslongas}\ be\ \textbf{:}\ po \mid \textbf{unless}\ be\ \textbf{:}\ po \mid \\
 & e\ \textbf{:}\ po \mid \textbf{if}\ be\ \textbf{then}\ po_1\ \textbf{else}\ po_2 \mid \textbf{repeat}\ po \\
\hline
\end{array}$$

FIGURE 5. Syntax of SANTA policy language.

Authorization defines the access to resources in the system. With respect to the computational model they define whether the execution of an action is permissible. An *authorization rule* defines the condition under which a *subject* is allowed to perform an *action* on an *object*. We will in the following describe the syntactic elements of the language informally by example.

EXAMPLE 1 (Unconditional authorizations). Everybody can register a paper with the EPS system (*eps*)

`r1` **::** **allow(S**,*eps*, *register*(**O**)) **when** `true`

Here **S** is a subject variable and can represent any subject (everybody). The object of the access control rule is the EPS system which is referred to by its name ($eps \in Objects$). The action is register that has a parameter that is a known object in the system, expressed by an object variable.

Example 1 shows a *positive* authorization rule. *Negative* authorization rules (denials) are analogous with the only difference that the consequence of the rule starts with the keyword **deny**.

EXAMPLE 2 (Conditional on the current state). Only the current owner of a bank account can withdraw money.

allow(S,**O**,`withdrw`()) **when** 0**:** `owner`(**S**,**O**) **and** `account`(**O**)

In Example 2, the condition is checked in the current state of the system. This is forced by the `e`**:** construct that forces the premise to be evaluated over the e long suffix of the execution history. In the case of e = 0, this is the current state of the system.

EXAMPLE 3 (Conditional on history). A subject must not perform an action on the same object twice.

deny(**S**,**O**,**A**) **when sometime done**(**S**,**O**,**A**)

In Example 3, after the action has been executed once, all further requests will be denied. This means that it is not possible to execute more than once. The condition is evaluated over the whole execution history. **sometime** here checks whether in *any* suffix of the history the requested action has been *done*. **done**(**S**,**O**,**A**) refers to the event raised by the system when an authorized action has been successfully performed (see Fig. 1). The use of the same variable names in the consequence and the premise of the rule means that the *same* subject, object, action are referred to. The next example illustrates this further:

EXAMPLE 4 (Conditional on history). A subject is denied to perform an action if this action has already been performed by another subject.

deny($\mathbf{S}_1$,**O**,**A**) **when sometime done**($\mathbf{S}_2$,**O**,**A**) **and** $\mathbf{S}_1$<>$\mathbf{S}_2$

The above rule allows to model exclusive resource access. $\mathbf{S}_1$ and $\mathbf{S}_2$ are free variables that are quantified over the set of all *Subjects* and local to the rule.

EXAMPLE 5 (Invariant). A subject is only allowed to take a loan if (s)he was never bankrupt.

allow(**S**,$\mathbf{O}_{loan}$,take) **when always not** bankrupt(**S**)

Example 5 assumes that the event bankrupt(**S**) is observable by the reference monitor.

EXAMPLE 6 (Choice). For a child younger than 10, both parents need to give consent, otherwise one parent's consent suffices.

allow(**S**,**O**,**A**) **when** $\mathbf{S}_1$<>$\mathbf{S}_2$ **and**
parent($\mathbf{S}_1$, **S**) **and** parent($\mathbf{S}_2$, **S**) **and**
if age(**S**)<10 **then**
sometime done($\mathbf{S}_1$,**O**,consent(**A**)) **and**
sometime done($\mathbf{S}_2$,**O**,consent(**A**))
else sometime done($\mathbf{S}_1$,**O**,consent(**A**)) **or**
sometime done($\mathbf{S}_2$,**O**,consent(**A**))

In Example 6, we consider two distinct subjects $\mathbf{S}_1$ and $\mathbf{S}_2$ to be the parents of child **S**. If the age is <10, the first branch is evaluated, checking whether both parents gave consent before. In the alternative branch only the prior consent of one parent is needed (**or**).

EXAMPLE 7 (Collaboration). The door can only be opened if at least two subjects requested the door to be opened within the last five states.

allow(**S**,door,open) **when** 5**: sometime req**($\mathbf{S}_1$,door,open)
and sometime req($\mathbf{S}_2$,door,open) **and** $\mathbf{S}_1$<>$\mathbf{S}_2$

In Example 7, two (distinct) subjects must collaborate in opening the door, by requesting the door to be opened within the last five system states. Note that in this example the outcome of the previous request is not decisive, i.e. even if both previous requests were denied the condition is met. The order in which these requests were made is arbitrary and the requests could even be made concurrently.

EXAMPLE 8 (Time). A subject should not access the same resource within 10 time units.

deny(**S**,**O**,**A**) **when exists** $\mathbf{t}_{last}$ **in** TIME **:**
(**sometime last**(1) **: done**(**S**,**O**,$\mathbf{A}_1$) **and** $\mathbf{t}_{last}$ = **T**)
and 0**:** $\mathbf{t}_{last}$ + 10 < **T**

We assume that the current system time is available as variable **T**. We treat time as a set TIME and use existential quantification to bind the time when the last access **last**(1)**: done**(**S**,**O**,$\mathbf{A}_1$) has taken place to $\mathbf{t}_{last}$. The comparison between this last access time and the current system time is made in the current state: 0**:** $\mathbf{t}_{last}$+ 10 < **T**. Note that the action of the last access can vary from the current request, i.e. $\mathbf{A}_1$ may or may not be the same as **A**.

EXAMPLE 9 (Cardinality). A subject should not access the same resource more than seven times.

deny(**S**,**O**,**A**) **when sometime last**(7) **: done**(**S**,**O**,$\mathbf{A}_1$)

Combining Example 8 and 9 we can express:

EXAMPLE 10 (Cardinality and time). A subject should not be allowed to make more than 100 access request in 24 time units.

deny(**S**,**O**,**A**) **when exists** $\mathbf{t}_0$ **in** TIME **:**
(**sometime last**(100) **: done**(**S**,**O**,$\mathbf{A}_1$) **and** $\mathbf{t}_0$ = **T**)
and 0**:** $\mathbf{t}_0$ + 24 < **T**

EXAMPLE 11 (Sequential access). An invoice cannot be payed unless it has been received and was authorized by an accountant.

deny(**S**,bank,pay($\mathbf{O}_{inv}$)) **when**
not (**sometime done**(**S**,$\mathbf{O}_{inv}$,receive) **;**
sometime done($\mathbf{S}_A$,$\mathbf{O}_{inv}$,
authorise))
and 0**:** role($\mathbf{S}_A$,accountant) **and** $\mathbf{S}_A$<>**S**

This example enforces a sequence of two actions (receive and authorise) to be successfully performed prior to the invoice $\mathbf{O}_{\mathrm{inv}}$ being payed.

EXAMPLE 12 (Decision rule). The denial of a right takes precedence over the allowance.

```
decide(S,O,A) when
          0:allow(S,O,A) and not deny(S,O,A)
```

Decision rules resolve conflicts that arise from the hybrid policy approach. A decision is derived from positive and negative authorization rules. In the above example, the 'standard' decision rule that denial takes precedence is expressed. Note that this is checked in the current system state. To use the policy as a closed policy, the following decision rule could be used.

EXAMPLE 13 (Closed decision rule). An action is denied unless explicitly allowed.

```
decide(S,O,A) when 0:allow(S,O,A)
```

Similarly for an open policy:

EXAMPLE 14 (Open decision rule). An action is allowed unless explicitly denied.

```
decide(S,O,A) when 0: not deny(S,O,A)
```

Decision rules are usually not history-dependent, however, we do not place any restriction on the use of the rules in our policy language. The following example shows a decision rule, that requires the stability of a *not deny* over a period of 10 states.

EXAMPLE 15 (History-based decision rule). An action is allowed if no negative authorization could be derived in the last 10 time units.

```
decide(S,O,A) when 10: always not
deny(S,O,A)
```

Having given examples of how history-based policy rules can be specified in our policy language, we now show how these can be combined into logical units, referred to as simple policies. In the next section, we present the second contribution of this paper and show how these simple policies can be composed sequentially to describe policy change over time and events.

4.2. Policies and compositions

Policy rules can be combined into *simple policies*:

```
policy pn ::
  allow(S,O_loan,take) when always not bankrupt(S)
  deny(S,O,A) when sometime last(7) : done(S,O,A_1)
  decide(S,O,A) when
        0: allow(S,O,A) and not deny(S,O,A)
end
```

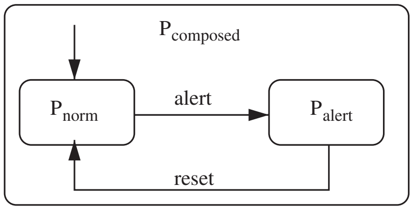


FIGURE 6. Diagrammatic representation of a sequential policy composition.

This example combines some of the rules discussed in Section 4 into a simple policy with the name *pn*, which then can form a building block of further policy compositions. All rules contained in a simple policy apply simultaneously. Typically a simple policy captures the protection requirements for a specific situation or a specific scope of the system. These specific policies are then composed to account for dynamic context changes which we present in the following.

4.2.1. *Sequential compositions.*

Sequential compositions of policies define how policies change over time and on the occurrence of events. Many protection requirements do only apply in certain situations; it is therefore beneficial to allow policy designers to focus on a particular situation when specifying a policy and provide them with tools to compose these individual policies to capture their dynamic nature.

EXAMPLE 16 (Intrusion detection). An intrusion detection system could raise an intrusion alert and automatically trigger a lock-down of some of the systems functions using the following policy composition:

```
policy p_norm :: /* ... */ end
policy p_alert :: /* ... */ end
policy p_composed ::
  repeat ( (unless event_alert(): p_norm) ;
                    (unless event_reset(): p_alert)
         ) end
```

Here, two policies are defined: p_{norm} captures the protection requirements for a normal mode of operation, and p_{alert} a stricter policy that should be enforced if the intrusion detection system raises an alert. The composition defines that the initial policy is p_{norm}, that changes to the policy p_{alert} when the event *event_alert()* is observed. The policy p_{alert} applies unless the alert status is explicitly reset; the composition then repeats with the policy p_{norm}. This can be represented diagrammatically as in Fig. 6.

To achieve the same effect with an (de-)activation-based model, the policy p_{norm} would have to contain an additional obligation rule that disables the policy p_{norm} and enables p_{alert} on the event *alert* whilst the policy p_{alert} requires the obligation that reverses this setting on the event *reset*. Conceptually such a system can be in four distinct states: $\langle\rangle$, $\langle \mathrm{p_{norm}}\rangle$, $\langle \mathrm{p_{alert}}\rangle$, $\langle \mathrm{p_{norm}}+\mathrm{p_{alert}}\rangle$. Only the detailed analysis of the obligation rules can show that the states $\langle\rangle$ and $\langle \mathrm{p_{norm}}+\mathrm{p_{alert}}\rangle$ cannot be reached when starting in state $\langle \mathrm{p_{norm}}\rangle$. Whilst this analysis is not overly complicated for the presented scenario, for more complex policies the number of states increases exponentially leaving any policy designer to be heavily reliant on tool-support to check for undesired interactions between policy activations.

The above Example 16 considered an event for raising the alarm and resetting it to be observable. This could be modelled as part of a system using a status attribute: *alert_status* where the value of $true$ denotes alert, and $false$ no alert. The difference is that the policy mechanism now needs to react to changes in the *alert_status*. This can be achieved by adapting the example:

EXAMPLE 17 (Intrusion detection with attribute). Adaptation of Example 16 using a status attribute.

```
policy $p_{composed}$ ::
  repeat ( (unless alert_status : $p_{norm}$) ;
                    (aslongas alert_status: $p_{alert}$)
          ) end
```

Another example of a policy composition includes conditional policy branches:

EXAMPLE 18 (Procurement). A development company wants to outsource certain parts of their product development. The procurement process includes four phases, *tender*, *contract*, *development*, *acceptance*. During the tender process only high-level information about the product is available to potential contractors under the policy *p-tender*. At the contracting stage more information is made available to the selected contractor under the policy *p-contract*. As subsidiaries are allowed to subcontract the process now differentiates between subsidiaries and external contractors based on the contractor's status. This is expressed in policies *p-sub* and *p-ext*, respectively. At the end of the procurement, the same policy *p-acc* applies to all contractors.

```
policy $p_{composed}$ ::
      (unless contractorSelected() : p−tender) ;
      (unless contractSigned() : p−contract) ;
      (unless developmentComplete() :
         if isSubsidiary(Contractor)
               then p−sub
               else p−ext
      ) ;
      p−acc
end
```

Note that the above policy does not repeat and the subpolicies would reflect contractor access to part of the organizational assets that are germane to the development work.

A composition of policies can consist of other composed policies, to maximize reuse of the logical building blocks that policies represent.

4.2.2. *Policy union, intersection and difference.*

Policies can also be combined in parallel, i.e. multiple policies can be enforced at the same time (see, e.g. [21, 26, 44, 73–75]). A full discussion of the parallel composition of policies, however, exceeds the scope of this paper. In the following, we therefore outline how this can be achieved and point out considerations for such compositions.

Simple policies can be combined similarly to [26, 74] by merging their rules.

EXAMPLE 19 (Merging of simple policies (union)). Let policy $\mathrm{p_{open}}$ be an open policy only stating denials, allowing anybody to perform the action a on the object o, by default.

```
policy $p_{open}$ ::
  decide(S,O,A) when 0: not deny(S,O,A)
end
```

Similarly, let policy p_{closed} be a closed policy only stating permissions, by allowing the action a on the object o explicitly.

```
policy $p_{closed}$ ::
  allow(S,o,a) when true
  decide(S,O,A) when 0: allow(S,O,A)
end
```

Merging the two policies yields:

```
policy $p_{merged}$ ::
  allow(S,o,a) when true
  decide(S,O,A) when 0: allow(S,O,A)
  decide(S,O,A) when 0: not deny(S,O,A)
end
```

Indeed, as we will show in Section 5, the above policy is equivalent to:

```
policy $p_{merged}$ ::
  allow(S,o,a) when true
  decide(S,O,A) when 0:allow(S,O,A) or not deny(S,O,A)
end
```

Note, however, that the merging of rules weakens the policy. For the merged policy to decide to authorize an access at least one of the merged policies must authorize the access. Whilst the merging of two simple policies preserves the intuition of both specifications other set operators such as intersection and difference are typically defined on a syntactic basis (e.g. by removing rules that are not in the other set). We believe that these operations are only of limited use, as a policy can contain rules that are semantically equivalent, but have a different syntax. For example:

EXAMPLE 20 (Rule syntax vs. semantics).

```
p1:: allow(S,O,A) when true end
p2:: allow(S,O,A) when 0:true end
```

The intersection of the two simple policies, is empty (based on syntax), however, semantically both policies are the same, i.e. in the intersection should be p1, which is equivalent to p2.

It is preferable to have operators for policy composition capturing the meaning of the policies. To facilitate this, we propose the construction of policy compositions, where each policy is evaluated as a self-contained unit. The composition of policies (e.g. union, intersection, difference) is then defined by a decision rule that takes into account the outcome of both policies. The following example illustrates this:

Example 21 (Policy composition (intersection)). To intersect two policies p1 and p2 we use the ternary operator **par**

```
p1:: allow(S,O,A) when true end
p2:: allow(S,O,A) when 0:true end
p3:: p1 par p2 deconflict { decide(S,O,A) when
                 p1.decide(S,O,A) and p2.decide(S,O,A) }
```

Here policy p3 is defined by the outcomes of policy p1 and p2. p3 defines the intersection of the two policies as an authorization decision is only made if *both* policy p1 and p2 agree on the outcome.

The advantage of this approach is that the meaning of the component policies as a hybrid combination of positive, negative and decision rules is preserved. The **par** operator locally captures the semantics of the component policy, which can then be referred to in the deconfliction policy that defines the decisions made by the policy composition. The composition preserves the semantics of the component policies.

To remove the need to explicitly specify a deconfliction, we allow the omission of the **deconflict** dp part of the **par** construct, which then defaults to the rule stated in Example 21.

This approach can be used to compose component policies that can themselves be policy compositions. For example:

Example 22 (Policy composition). Policy P1 will be effective after 10 states, if policy P2 becomes effective.

```
P1 :: /* ... */ end
P2 :: /* ... */ end
P−EMPTY :: end
P2 par (10: P−EMPTY ; P1)
```

Here, policy p3 is defined by the outcomes of policy p1 and p2. p3 defines the intersection of the two policies as an authorization decision is only made if *both* policy p1 and p2 agree on the outcome.

As the contribution of this paper is the specification of history-based policies, the sequential composition of policies and compositional proof rules for their verification, we will refrain from giving the semantics of the **par** construct in this paper.

5. FORMAL SEMANTICS OF SANTA

As SANTA is using a compositional approach to the specification of policies, its underlying formalism should therefore also express specifications of system behaviours compositionally. The following introduces ITL and then provides the formal semantics for SANTA.

5.1. SANTA semantics

We first give the semantics for rules and then we define the semantics for policies.

5.1.1. Semantics of rules.

Policy rules define the behaviour of the access control variables. The consequence of a rule determines the type of the rule and the subjects, objects and actions the rule applies to. The operator *always-followed-by* [74] is used to capture the relation between the premise of a rule and its consequence. Let us first define the semantics of a premise.

The syntax that is used in the premise is actually a subset of ITL formulae. The semantics of a rule premise is then as follows:

$$\begin{aligned}
[\![pr_1 \,;\, pr_2]\!] &\mathrel{\widehat{=}} [\![pr_1]\!] \,;\, [\![pr_2]\!] \\
[\![pr_1 \ \textbf{and}\ pr_2]\!] &\mathrel{\widehat{=}} [\![pr_1]\!] \wedge [\![pr_2]\!] \\
[\![pr_1 \ \textbf{or}\ pr_2]\!] &\mathrel{\widehat{=}} [\![pr_1]\!] \vee [\![pr_2]\!] \\
[\![\textbf{not}\ pr]\!] &\mathrel{\widehat{=}} \neg [\![pr]\!] \\
[\![\textbf{sometime}\ pr]\!] &\mathrel{\widehat{=}} \Diamond [\![pr]\!] \\
[\![\textbf{always}\ pr]\!] &\mathrel{\widehat{=}} \Box [\![pr]\!] \\
[\![\textbf{if}\ be\ \textbf{then}\ pr_1\ \textbf{else}\ pr_2]\!] &\mathrel{\widehat{=}} (be \wedge [\![pr_1]\!]) \vee (\neg\, be \wedge [\![pr_2]\!]) \\
[\![\textbf{exists}\ x\ \textbf{in}\ se : pr]\!] &\mathrel{\widehat{=}} \exists x \bullet x \in se \wedge [\![pr]\!] \\
[\![\textbf{forall}\ x\ \textbf{in}\ se : pr]\!] &\mathrel{\widehat{=}} \forall x \bullet x \in se \supset [\![pr]\!] \\
[\![\textbf{last}(e)\ pr]\!] &\mathrel{\widehat{=}} ((\mathsf{empty} \wedge pr) \,;\, \mathsf{skip}; \\
&\qquad \Box \neg(\mathsf{empty} \wedge pr))^{n} \\
[\![e \,\mathbf{:}\, pr]\!] &\mathrel{\widehat{=}} \mathsf{finite} \,;\, (\mathsf{len}(e) \wedge [\![pr]\!])
\end{aligned}$$

The semantics of $e \,\mathbf{:}\, pr$ includes finite; to obtain the 'history' of length e from the point where w holds. The operator *always-followed-by* ($\mapsto$) is defined as follows:

$$f \mapsto w \mathrel{\widehat{=}} \boxed{i}\, (f \supset \mathsf{fin}\,(w)) \qquad (1)$$

The intuition of the operator is that whenever f holds for a prefix interval then w holds in the last state of that interval. For example, if f holds only over the prefix intervals indicated in Fig. 7, then $f \mapsto w$ determines that w is true in states σ_1, σ_2, σ_6 and σ_8. The value of w in any of the other states is not determined.

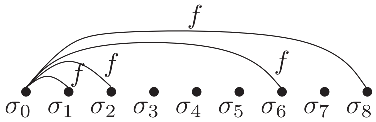


FIGURE 7. The operator *always-followed-by* ($\mapsto$).

The semantics of individual rules is then defined as follows:

$$
\begin{aligned}
&[\![\textbf{allow}\ (su, ob, ac)\ \textbf{when}\ pr]\!] \mathrel{\widehat{=}}\\
&\quad \forall vs \in Subjects \bullet \forall vo \in Objects \bullet \forall va \in Actions \bullet\\
&\qquad [\![pr]\!] \mapsto \mathsf{Aut}^+(su, ob, ac)\\
&[\![\textbf{deny}\ (su, ob, ac)\ \textbf{when}\ pr]\!] \mathrel{\widehat{=}}\\
&\quad \forall vs \in Subjects \bullet \forall vo \in Objects \bullet \forall va \in Actions \bullet\\
&\qquad [\![pr]\!] \mapsto \mathsf{Aut}^-(su, ob, ac)\\
&[\![\textbf{decide}\ (su, ob, ac)\ \textbf{when}\ pr]\!] \mathrel{\widehat{=}}\\
&\quad \forall vs \in Subjects \bullet \forall vo \in Objects \bullet \forall va \in Actions \bullet\\
&\qquad [\![pr]\!] \mapsto \mathsf{Aut}(su, ob, ac)
\end{aligned}
$$

Let $vs \in frees(r)$; $vo \in freeo(r)$; and $va \in freea(r)$ be the free variables (subject, object and action, respectively) in the rule r. The propositional state variable $\mathsf{Aut}^+(su, ob, ac)$ captures positive authorizations. If its value is true the policy defines a positive authorization for the subject su to perform action ac on object ob. Similarly $\mathsf{Aut}^-(su, ob, ac)$ captures negative authorizations. The propositional state variable $\mathsf{Aut}(su, ob, ac)$ defines the access control decision taken by the reference monitor.

5.1.2. Semantics of policies

We first define the semantics of a policy that consists of a collection of rules. The implication, in the semantics of an individual rule, $f \supset \mathsf{fin}\ w$ means that w can be *true* in a state even if f does not hold in the prefix of that interval. Policies (at semantic level), define the access decision in every state of the reference monitor and are important for its verification.

We adopt an refinement approach using the '*strong followed-by*' operator denoted by $\leftrightarrow$, to obtain a complete policy specification.

$$f \leftrightarrow w \mathrel{\widehat{=}} \boxed{\scriptstyle\mathrm{i}}\ (f \equiv \mathsf{fin}\ w) \tag{2}$$

Unlike the operator always-followed-by ($\mapsto$), a rule of the form $f \leftrightarrow w$ determines in any state the value of the state formula w. If f holds in the prefix of the reference interval, then w must hold in that state otherwise w must not hold in that state. (Cf. Fig. 7.)

The motivation of using a refinement approach is that we can show that a system that satisfies $f \leftrightarrow w$ also satisfies $f \mapsto w$. Thus, by rewriting the policy specification using the algorithm presented below we strengthen the specification by adding *default rules* such that the specification is *complete*. By this, we mean that the specification defines the value of $\mathsf{Aut}^+(s, o, a)$, $\mathsf{Aut}^-(s, o, a)$ and $\mathsf{Aut}(s, o, a)$ in each state of the system and thus can be enforced by the reference monitor.

The semantics of a policy of the form $ru_1 \ldots ru_n$ is a *semantically completely specified* formula, i.e. the following formula:

$$
\bigwedge_{\substack{s \in Subjects\\ o \in Objects\\ a \in Actions}}
\begin{array}{l}
(f(s, o, a) \leftrightarrow \mathsf{Aut}^+(s, o, a)) \wedge\\
(g(s, o, a) \leftrightarrow \mathsf{Aut}^-(s, o, a)) \wedge\\
(h(s, o, a) \leftrightarrow \mathsf{Aut}(s, o, a)),
\end{array}
\tag{3}
$$

where, for each $s \in Subjects$, $o \in Objects$ and $a \in Actions$,

(i) $f(s, o, a) \mathrel{\widehat{=}} \bigvee_{i=1}^{l} [\![pr_i]\!]$ and pr_i appears as a premise in an **allow** rule of $ru_1 \ldots ru_n$. If there are no **allow** (s, o, a) rules in $ru_1 \ldots ru_n$, then $f(s, o, a) = \mathsf{false}$.
(ii) $g(s, o, a) \mathrel{\widehat{=}} \bigvee_{i=1}^{m} [\![pr_i]\!]$ and pr_i appears as a premise in a **deny** rule of $ru_1 \ldots ru_n$. If there are no **deny** (s, o, a) rules in $ru_1 \ldots ru_n$, then $g(s, o, a) = \mathsf{false}$.
(iii) $h(s, o, a) \mathrel{\widehat{=}} \bigvee_{i=1}^{k} [\![pr_i]\!]$ and pr_i appears as a premise in a **decide** rule of $ru_1 \cdots ru_n$. If there are no **decide** (s, o, a) rules in $ru_1 \ldots ru_n$, then $h(s, o, a) = \mathsf{false}$.

For each triple $(s, o, a) \in Subjects \times Objects \times Actions$, the formula $[\![ru_1 \ldots ru_n]\!]$ contains exactly one rule of the form $f(s, o, a) \leftrightarrow \mathsf{Aut}^+(s, o, a)$, one rule of the form $g(s, o, a) \leftrightarrow \mathsf{Aut}^-(s, o, a)$ and one rule of the form $h(s, o, a) \leftrightarrow \mathsf{Aut}(s, o, a)$. Therefore, it fully determines the value of $\mathsf{Aut}(s, o, a)$ at each state of the system.

Default rules are automatically provided. For example, if the policy po does not contain a rule for $\mathsf{Aut}^+(s, o, a)$, for some subject s, object o and action a, then it defaults to a rule of the form $\mathsf{false} \leftrightarrow \mathsf{Aut}^+(s, o, a)$ in $[\![ru_1 \ldots ru_n]\!]$. Similarly for $\mathsf{Aut}^-(s, o, a)$ and $\mathsf{Aut}(s, o, a)$ if there are no explicit rules for them in $ru_1 \ldots ru_n$. As such, $[\![ru_1 \ldots ru_n]\!]$ grants every right granted by $ru_1 \ldots ru_n$ and denies everything else.

Lemma 5.1(a) false in the premise of a policy rule does not constrain an access control decision. Furthermore, you can combine the conjunct of two policy rules into a single rule that has as a premise the disjunction of the other rules' premises—provided the rules have the same consequence.

LEMMA 5.1 (Tautologies).

(a) $\mathsf{false} \mapsto w$,
(b) $((f_1 \vee f_2) \mapsto w) \equiv ((f_1 \mapsto w) \wedge (f_2 \mapsto w))$.

LEMMA 5.2 (Refinement). $(f \leftrightarrow w) \supset (f \mapsto w)$.

Lemma 5.2 establishes that the operator *strong always-followed-by* is a refinement (subset relation on sets of intervals) of the operator weak always-followed-by.

Proof.

$$\begin{array}{lcl}
(f \leftrightarrow w) & \equiv & \{\text{by definition}\} \\
\boxed{i}\ (f \equiv \mathsf{fin}\ w) & \equiv & \{\text{ITL reasoning}\} \\
\boxed{i}\ ((f \supset \mathsf{fin}\ w) \wedge (\mathsf{fin}\ w \supset f)) & \equiv & \{\text{distribution of} \\
(\boxed{i}\ (f \supset \mathsf{fin}\ w)) \wedge & & \quad \boxed{i}\ \text{over} \wedge\} \\
\quad (\boxed{i}\ (\mathsf{fin}\ w \supset f)) & \supset & \{\text{ITL reasoning}\} \\
\boxed{i}\ (f \supset \mathsf{fin}\ w) & \equiv & \{\text{by definition}\} \\
f \mapsto w. & &
\end{array}$$

□

The following theorem states that a complete policy specification is a refinement of the collection of individual 'weak' rules.

Theorem 5.1. $[\![ru_1 \ldots ru_n]\!] \supset \bigwedge_{i=1}^{n} [\![ru_i]\!]$.

Proof. Semantics of a policy is a refinement of the conjunction of the rules contained in the policy.

(1) For each possible consequence of a rule $conseq \in \{\mathsf{Aut}^{-}(s, o, a), \mathsf{Aut}^{+}(s, o, a), \mathsf{Aut}(s, o, a)\}$ where $s \in Subjects, o \in Objects, a \in Actions$ we add a default rule of the form $\mathsf{false} \mapsto conseq$ to the conjunct. As this is a tautology (Lemma 5.1(a)), the meaning of the conjunct $\bigwedge_{i=1}^{n} [\![ru_i]\!]$ is not changed.

(2) Using Lemma 5.1(b), we combine any two rules in $\bigwedge_{i=1}^{n} [\![ru_i]\!]$ that have the same consequence without changing the meaning of the rule. The resulting conjunct is similar to Equation (3), albeit every rule is using the operator always-followed-by ($\mapsto$).

(3) Lemma 5.2 shows that the operator strong always followed by ($\leftrightarrow$) is a refinement of the operator always followed by ($\mapsto$). Replacing in every rule the operator always-followed-by ($\mapsto$) with its refinement ($\leftrightarrow$), yields the policy semantics given in Equation (3).

□

The semantics of the other policy construct is as follows: Let *Subjects*, *Objects*, *Actions* be, respectively, the universal set of subjects, objects and actions. Note the SANTA construct **policy** pn **:** *po* **end** gives policy *po* a name pn, i.e. it acts as an abbreviation for *po* so we do not need to give a semantics to this construct.

$$\begin{array}{rl}
[\![po_1 \mathbin{\textbf{;}} po_2]\!] & \mathrel{\widehat{=}} [\![po_1]\!] \mathbin{;} [\![po_2]\!] \\
[\![\textbf{aslongas}\ be \mathbin{\textbf{:}} po]\!] & \mathrel{\widehat{=}} ((([\![po]\!] \wedge \Box be) \mathbin{;} \mathsf{skip}) \wedge \\
 & \quad\ \mathsf{fin}\ \neg\, be) \vee (\mathsf{empty} \wedge \neg\, be) \\
[\![\textbf{unless}\ be \mathbin{\textbf{:}} po]\!] & \mathrel{\widehat{=}} \neg[\![\textbf{aslongas not}\ be \mathbin{\textbf{:}} po]\!] \\
[\![e \mathbin{\textbf{:}} po]\!] & \mathrel{\widehat{=}} \mathsf{len}(e) \wedge [\![po]\!] \\
[\![\ \textbf{if}\ be\ \textbf{then}\ po_1 \textbf{else} po_2]\!] & \mathrel{\widehat{=}} (be \wedge [\![po_1]\!]) \vee (\neg\, be \wedge [\![po_2]\!]) \\
[\![\textbf{repeat}\ po]\!] & \mathrel{\widehat{=}} [\![po]\!] \mathbin{;} ([\![po]\!])^{*}
\end{array}$$

The sequential composition $po_1 \,;\, po_2$ does not determine the exact state in which the policy change does occur. In this sense the change is non-deterministic. Using the operators **unless** and **aslongas**, as well as an explicit timing e **:** po the duration over which a policy holds is specified.

The semantics of **aslongas** states that during the interval in which $p0$ holds the condition be remains invariant. In the last state of the interval be is false.

The semantics of policies is used to reason about *allowed information flows* under a given policy. In the following we define information flow with respect to policies.

5.2. Policy-level information flow analysis

We analyse information flows that are permitted by a given access control policy. The analysis provides answers as to whether information can be transferred between subjects and objects in the system under a given access control policy. The analysis excludes any transfer of information that is passed outside of the control of the reference monitor enforcing the policy and does not include the analysis of covert channels. To give such guarantees the approach must be augmented with well-established language-based information flow analysis techniques [61]. The analysis presented here is, however, helpful in analysing the impact of policy changes, albeit limited to the extend that information flow can be controlled using access control mechanisms.

The information flow analysis at the level of policies is based on two categories of actions: *read* actions and *write* actions. A read action is an action that can leak information from the object to the subject that performs the action on the object. For example, checking the balance of a bank account or reading a file leaks information from the bank account or the file to the subject that exercises the action. In contrast, a write action allows information to flow from the subject that exercises the action to the object recipient. For example, crediting a bank account or appending to a file. Actions that belong to neither of these categories are ignored. However, some actions may belong to both categories. In the sequel we denote by $Actions_r$ and by $Actions_w$ the subset of *Actions* of all read actions and all write actions, respectively.

Definitions 5.1 and 5.2 define our notion of *allowed direct information flow* from a subject to an object and from an object to a subject, respectively.

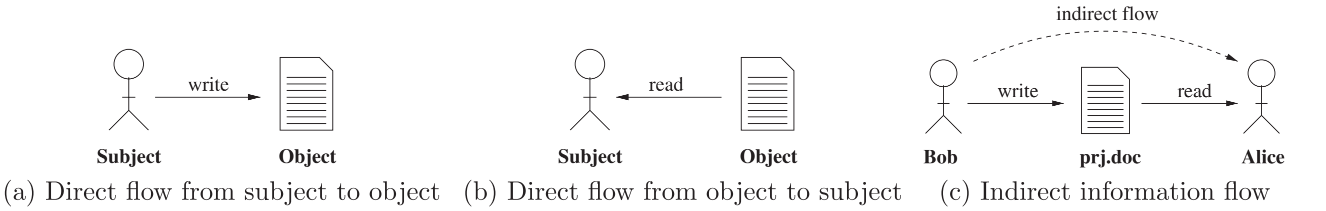
Definition 5.1. *We say that there is an allowed direct information flow from a subject s to an object o if the subject s is allowed to perform a write action on the object o, viz*

$$s \rightsquigarrow o \mathrel{\widehat{=}} \bigvee_{a \in Actions_w} \mathsf{Aut}(s, o, a)$$

Figure 8a illustrates a direct flow from a subject to an object.

Definition 5.2. *We say that there is an allowed direct information flow from an object o to a subject s if the subject s is allowed to perform a read action on the object o, viz*

$$o \rightsquigarrow s \mathrel{\widehat{=}} \bigvee_{a \in Actions_r} \mathsf{Aut}(s, o, a)$$



(a) Direct flow from subject to object (b) Direct flow from object to subject (c) Indirect information flow

FIGURE 8. Direct and indirect information flow. (**a**) Direct flow from subject to object. (**b**) Direct flow from object to subject. (**c**) Indirect information flow.

Figure 8b illustrates a direct flow from an object to a subject.

As such, we have defined a relation $\rightsquigarrow$ over the set $\mathit{Entities} = \mathit{Subjects} \cup \mathit{Objects}$. Note that $v \rightsquigarrow v'$ is a state formula, for any $v, v' \in \mathit{Entities}$. For example the formula $\bigcirc(v_1 \rightsquigarrow v_2)$ holds for an interval if there is an allowed direct information flow from entity v_1 to entity v_2 in that interval's second state. Yet another example is the formula $\Diamond(v_1 \rightsquigarrow v_2)\,;\Diamond(v_2 \rightsquigarrow v_3)$, that holds for an interval if information can flow from entity v_1 to entity v_2 at some point t in time, and from entity v_2 to another entity v_3 at some later time $t' \geq t$. This illustrates an implicit (possible) leakage of information from entity v_1 to entity v_3. For this reason, it is necessary to compute the transitive closure of the relation $\rightsquigarrow$ that lists all possible flows of information allowed by an access control policy. The information flow transitive closure is formalized in Definition 5.3.

DEFINITION 5.3. *Information can flow from $v \in$ Entities to $v' \in$ Entities if there exists an allowed direct information flow from v to v' at some point in time or information can flow from v to some entity $u \in$ Entities and from u to v' later on, i.e.*

$$v \rightsquigarrow^{+} v' \mathrel{\widehat{=}} \Diamond(v \rightsquigarrow v') \vee \bigvee_{u \in \mathit{Entities}} ((v \rightsquigarrow^{+} u)\ ;\ (u \rightsquigarrow^{+} v')).$$

Figure 8c gives an example of transitive information flow from Bob to $Alice$ via the file $prj.doc$. The information flow analysis assumes that flows between objects, e.g. the copying of a file ‘a.txt’ to a file ‘b.txt’ is facilitated by a subject, e.g. the copy process. In this case the copy process ‘reads’ from file ‘a.txt’ and writes to file ‘b.txt’ and the analysis would determine whether the policy allows the ‘read’ and the ‘write’ action to occur in this sequence.

6. CASE STUDY

To show the flexibility of the proposed model, we develop the access control policy for an EPS system. The focus of the case study is on the access-control requirements and the sequential composition of policies and demonstrates how policies can be developed in a modular fashion and then be composed to express the overall protection requirements of the system.

Of particular interest to the contributions of this paper are the history-based requirements in the submission phase (version control) and the review phase (no reviews after a decision was made). The case study was chosen based on the natural occurrence of ‘phases’ in IT systems that support business processes. These phases are used to sequentially compose the policy and illustrate the compositional verification rules presented in Section 7.

6.1. System description

We split the description of the EPS system into the different phases of the process (Fig. 9).

6.1.1. Submission phase.
As the EPS system is available via a web-interface, everybody can *register* a paper. The person registering the paper is referred to as the principal author of that paper and this will be denoted by the predicate $author(A, P)$, where A is the author and P the paper. The principal author can *add* coauthors for a registered paper, denoted by the predicate $coauthor(A, P)$, where A is the coauthor and P is the paper. The author and coauthors can *upload* a new version of the paper and *download* the most recent version from the web-interface. The EPS system implements a basic form of version control, in that uploading a version of the paper cannot overwrite changes made by others without reviewing them. Coauthors can only be *remove*d from a paper by the principal author or themselves.

6.1.2. Review phase.
After the submission period no paper may be registered or uploaded and the coauthors cannot be changed anymore. The committee *assigns* referees to the papers, this is denoted by the predicate $referee(R, P)$, where R is the referee and P is the paper. Referees can also be *withdrawn* by the committee. The system ensures, that no one referees a paper of which he

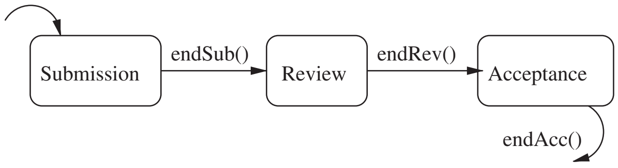


FIGURE 9. Phases in the EPS system.

is a (co-)author or of which an author is working in the same institution. The affiliated institution of a subject S is denoted by the function $institute(S)$.

Referees can download the most recent version of a paper they are assigned to and *write* a review for this paper. We capture the relation between a review and its associated paper with the predicate $review(V, P)$, where V is the review and P is the paper. The relation between a referee and a review is denoted by the predicate $owner(R, V)$, where R is the referee and V is the review. Reviewers can *read* only their own reviews, whilst the committee is allowed to *read* all reviews. No one else is allowed to *read* reviews during the review period. At the end of the review period, the committee scores the paper and thus decides whether to *accept* or *reject* it. It is possible to revise this decision, but referees cannot change their reviews after the decision has been taken by the committee.

6.1.3. *Acceptance phase.*

After the review phase the (co-)authors can read the reviews of their papers. A (co-)author of accepted papers can download and upload revised versions of the paper to take the referees' reviews into account and to provide a camera ready copy of the paper. The same integrity checks (version control) as during the submission phase are made.

6.2. Functional specification

Functional and security requirements cannot be viewed in isolation. They are highly dependent. We split the system description in three phases in order to describe the behaviour of the system. However, it is important to note that the systems functionality remains the same over the phases, whilst the access to these functions is changing according to the phase and events.

We will not detail the functional specification of the system in this paper, but focus on the specification of the dynamically changing access control policy. Table 1 provides a summary of the systems functions and their effect, that is reflected in the predicates. We will additionally use the events $\mathbf{done}(S, O, A)$ to denote the successful termination of the action A on object O by subject S. This links with the computational model presented earlier in Fig. 1 where this event is raised by the *System* upon the execution of action A on the object O.

Note that the functional specification does not cover the access rights and therefore the division in different phases, as this will be defined by the dynamically changing access control policy.

6.3. Authorization policy specification

The policy specification reflects the informal description of the EPS system (Tables 1 and 2). We define the policy as a sequence of simple-policies that correspond to the requirements of each of the submission phases (see Fig. 9).

TABLE 1. Actions in the EPS system.

Action	Effect	Description
$\mathbf{done}(A, eps, register(P))$	$author(A, P)$	Register paper
$\mathbf{done}(A, P, add(A'))$	$coauthor(A', P)$	Add coauthor
$\mathbf{done}(A, P, remove(A'))$	$\neg\, coauthor(A', P)$	Remove coauthor
$\mathbf{done}(A, P, upload(Doc))$	$doc(Doc, P)$	Upload document
$\mathbf{done}(S, P, download)$		Download doc.
$\mathbf{done}(R, P, write(V))$	$review(V, P) \wedge owner(R, V)$	Upload review
$\mathbf{done}(S, P, read(V))$		Download review
$\mathbf{done}(S, P, assign(R))$	$referee(R, P)$	Assign referee
$\mathbf{done}(S, P, withdraw(R))$	$\neg\, referee(R, P)$	Remove referee
$\mathbf{done}(S, P, accept)$	$accepted(P)$	Accept paper
$\mathbf{done}(S, P, reject)$	$\neg\, accepted(P)$	Reject paper

TABLE 2. Events in the EPS system.

Event	Description
endSub()	End submission
endRev()	End review
endAcc()	End acceptance

The objective of the case study is to show how policies are composed sequentially to yield a structured specification against which the compositional verification approach (Section 7) can be applied. The policy rules that are history-based in this example are $\mathbf{R}_5$ and $\mathbf{R}_{12}$.

6.3.1. *Submission policy.*

We model the submission policy as a simple policy containing the rules $\mathbf{R}_1$ to $\mathbf{R}_6$. In the following we formalize the access-control requirements as authorization policies considering only the submission phase. The variable **O** is used in the following to refer to a concrete *paper*.

$\mathbf{R}_1$: *Registration*
Everybody can register a paper [with the EPS system].
allow (**S**,eps,register(**O**)) **when** true

$\mathbf{R}_2$: *Coauthors*
The principal author can add coauthors to a registered paper. Coauthors can only be removed from a paper by the principal author or themselves.
allow (**S**,**O**,add(**S**author)) **when** 0: author(**S**,**O**)
allow (**S**,**O**,remove(**S**author)) **when** 0: author(**S**,**O**)
allow (**S**,**O**,remove(**S**)) **when** 0: coauthor(**S**,**O**)

$\mathbf{R}_3$: *Upload*

The principal author or coauthors can upload a version of the paper.

```
allow (S,O,upload(Ov1)) when
  0: (author(S,O) or coauthor(S,O))
```

R$_4$: *Download*
The principal author or coauthors can download the most recent version from the web-interface.

```
allow (S,O,download) when
  0: (author(S,O) or coauthor(S,O))
```

R$_5$: *No Update Conflict (H)*
The system implements a basic form of version control, in that uploading a paper cannot overwrite changes made by others without reviewing them.

```
deny (S,O,upload(Ov1)) when
  exists x in subjects : (
  x <> S and done(x,O,upload(Ov2)) and
  always not done(S,O,download))
```

The rules **R**$_1$ to **R**$_5$ together with the following decision rule:

R$_6$: *Denial Takes Precedence*
Access is only granted if there is a positive authorization and no negative authorization.

```
decide (S,O,A) when
  0: (allow(S,O,A) and not deny(S,O,A))
```

form the simple policy for the submission phase.

P$_1$: *Submission Policy*

The submission policy contains the rules **R**$_1$, **R**$_2$, **R**$_3$, **R**$_4$, **R**$_5$ *with the decision rule* **R**$_6$

6.3.2. *Review policy.*

The Review policy is also defined as a simple policy. The following rules are included in the policy:

R$_7$: *No Registration, Upload and Author change*
No paper may be registered or uploaded and the additional authors cannot be changed anymore.

```
deny (S,eps,register(O)) when true
deny (S,O,upload(Ov)) when true
deny (S,O,add(Sauthor)) when true
deny (S,O,remove(Sauthor)) when true
```

R$_8$: *Committee*
The committee assigns referees to the papers. This implies their authorization. Referees can also be removed again [by the committee]. The committee is allowed to read all reviews.

```
allow (cmt,O,assign(S)) when true
allow (cmt,O,withdraw(S)) when true
allow (cmt,O,read(Orev)) when 0: review(Orev,O)
```

R$_9$: *Referee not Author*
The system ensures, that no one referees a paper of which he is the (co-) author or of which the author is working in the same institution.

```
deny (S,O,assign(Sreviewer)) when
  0: (author(Sreviewer,O) or
coauthor(Sreviewer,O))
```

```
deny (S,O,assign(Sreviewer)) when
  0: (exists y in subjects : (author(y,O) and
  institute(y) = institute(Sreviewer)))
```

R$_{10}$: *Referees*
Referees can download papers they are assigned to and write a review for this paper. They can also read their own reviews.

```
allow (S,O,download) when 0: referee(S,O)
allow (S,O,write(Orev)) when 0: referee(S,O)
allow (S,O,read(Orev)) when 0: owner(S,Orev)
and review(Orev,O)
```

R$_{11}$: *Paper Acceptance*
The committee can accept or reject papers.

```
allow (cmt, O, accept) when true
allow (cmt, O, reject) when true
```

R$_{12}$: *No Reviews after Decision (H)*
Referees cannot change their review after a decision has been made by the committee.

```
deny(S,O,write(Orev)) when
  sometime (done(cmt,O,accept) and referee(S,O))
```

```
deny(S,O,write(Orev)) when
  sometime (done(cmt,O,reject) and referee(S,O))
```

P$_2$: *Review Policy*

The policy for the review phase is a simple policy containing the rules **R**$_4$, **R**$_7$ *to* **R**$_{12}$ *with the decision rule denial takes precedence* **R**$_6$.

6.3.3. *Acceptance policy.*

The policy for the acceptance phase is again defined as a simple policy. The rules that need to be defined for this policy are:

R$_{13}$: *(Co-)Authors Review*
(Co-) Authors can read the reviews of their paper.

```
allow(S,O,read(Orev)) when
  0: ((author(S,O) or coauthor(S,O)) and
      review(Orev,O))
```

R$_{14}$: *(Co-)Authors upload*
(Co-) Authors of accepted papers can upload updated versions of their paper.

```
allow (S,O,upload(Ov)) when
  0: ((author(S,O) or coauthor(S,O)) and accepted(O))
```

P$_3$: *Acceptance Policy*

The acceptance policy contains the rules **R**$_{??}$ *and* **R**$_{??}$.

Additionally the rule $\mathbf{R}_4$ *that allows authors and coauthors to download the newest version of their paper; the rule* $\mathbf{R}_5$ *to prevent update conflicts and rule* $\mathbf{R}_6$ *to give precedence to denials in the policy, are included.*

Having defined the policies for the different phases, we will now show how these can be composed, to capture the dynamic policy change that is described in the scenario.

6.3.4. *Composing the policies.*

Initially the submission policy applies, this is followed by the review policy and then the acceptance policy. We can conveniently use the operator *unless* and the *sequential composition.*

$\mathbf{P_4}$: *Composed Policy*

This policy defines the dynamic change of policies at the transition of one phase in the EPS to the next.

unless `endSub`(): $\mathbf{P}_1$;
unless `endRev`(): $\mathbf{P}_2$;
unless `endAcc`(): $\mathbf{P}_3$

Note: To simplify the proof of properties we assume that each unless 'phase' is finite and has at least two states, i.e. $\llbracket \textbf{unless}\ be_i : P_i \rrbracket \supset \mathsf{more} \wedge \mathsf{finite}$ for $i = 1, 2, 3$. Informally this means that none of the events *endSub(), endRev() and endAcc()* occur concurrently and that the events *endSub(), endRev()* eventually occur.

The advantage of using dynamically changing policies is that new phases can be easily introduced—without modification of the systems functionality. To add for example an additional phase that allows (co-)authors to comment on the reviews, to clarify questions in the reviews and influence the scoring of their paper would require only a minimum of changes to the system.

Obviously data structures in the system must be modified, to allow to store comments on reviews. We express this relation as a predicate $cmt(C, V)$, where C is the comment of the review V. The functions in the Table 3 are added to the system.

Assuming that the access control requirements for this phase are captured in the policy $\mathbf{P_5}$:Comment Policy, then we can define the overall policy that includes the Comment Phase as:

$\mathbf{P_6}$: *Composed Policy with Comment Phase*

TABLE 3. Additional functions EPS system.

Action	Effect	Description
$done(S, V, writecmt(C))$	$cmt(C, V)$	Write comment
$done(S, V, readcmt(C))$		Read comment
Event		Description
endCmt()		End comment

To incorporate the Comment Phase we include the policy $\mathbf{P}_5$

unless `endSub`(): $\mathbf{P}_1$;
unless `endRev`(): $\mathbf{P}_2$;
unless `endCmt`(): $\mathbf{P}_5$;
unless `endAcc`(): $\mathbf{P}_3$

The above policy composition reflects the natural phases of the EPS system. In the following we will make use of these compositions to break down verification tasks into smaller sub-problems and present compositional proof-rules for safety, liveness and information flow properties.

7. VERIFICATION

This section describes the verification of properties and show how the compositional specification of policies can be exploited using compositional proof-rules that simplify the verification tasks. The following definition states when a policy satisfies a property.

Definition 7.1. *We say that a policy po satisfies a property* f *if and only if* $\llbracket po \rrbracket \supset f$ *is valid.*

The following subsections first present the proof rules followed by examples of proving information flow, safety and liveness properties and make use of the proof rules given below. The proofs are based on the semantics of the policies in the EPS system that was introduced in Section 6. For convenience we have included a mapping from the policies to their semantics in Appendix 1.

7.1. Proof rules

The following compositional proof rule splits the proof of a safety property for a sequential composed policy to proofs of its component policies.

Proof Rule 1.

$$\frac{\llbracket po_1 \rrbracket \supset \Box w, \quad \llbracket po_2 \rrbracket \supset \Box w}{\llbracket po_1 ; po_2 \rrbracket \supset \Box w}$$

Proof.

$$\begin{array}{ll}
 & \llbracket po_1 ; po_2 \rrbracket \\
\equiv & \{ \text{ SANTA semantics } \} \\
 & \llbracket po_1 \rrbracket \ ; \ \llbracket po_2 \rrbracket \\
\supset & \{ \text{ rule assumptions } \} \\
 & \Box w \ ; \ \Box w \\
\supset & \{ \text{ ITL reasoning } \} \\
 & \Box w
\end{array}$$

□

The following rule is a compositional proof rule of an unless policy. Note: the formula $\mathsf{more} \wedge \mathsf{finite}$ expresses that the unless 'phase' is at least two states but finite, which was an assumption made in the specification of the EPS in Section 6. Without this assumption, the proofs would follow the same proof-outline, however, include additional case analyses for the concurrent occurrence of events and non terminating phases.

PROOF RULE 2.

$$\frac{(((\llbracket po \rrbracket \wedge \Box \neg \llbracket be \rrbracket) \; ; \; \mathsf{skip}) \wedge \mathsf{finite} \wedge \mathsf{fin}\,(\llbracket be \rrbracket)) \;\supset\; prop}{(\llbracket \textbf{unless}\; be \;\textbf{:}\; po \rrbracket \wedge \mathsf{more} \wedge \mathsf{finite}) \;\supset\; prop}$$

Proof.

$$\begin{array}{ll} & \llbracket \textbf{unless}\; be \;\textbf{:}\; po \rrbracket \wedge \mathsf{more} \wedge \mathsf{finite} \\ \equiv & \{\text{ SANTA semantics }\} \\ & (((\llbracket po \rrbracket \wedge \Box \neg \llbracket be \rrbracket) \; ; \; \mathsf{skip}) \wedge \mathsf{more} \wedge \mathsf{finite} \wedge \mathsf{fin}\,(\llbracket be \rrbracket)) \\ \supset & \{\text{ Assumption, } f \; ; \; \mathsf{skip} \;\supset\; \mathsf{more}\} \\ & prop \end{array}$$

□

The following compositional proof rule splits the proof of a property for a complete specification of rules to proofs of individual weak rules. This rule is used when the weak rules have enough information to deduce the property.

PROOF RULE 3.

$$\frac{\llbracket ru_1 \rrbracket \;\supset\; prop, \ldots, \llbracket ru_n \rrbracket \;\supset\; prop}{\llbracket ru_1 \ldots ru_n \rrbracket \;\supset\; prop}$$

Proof.

$$\begin{array}{ll} & \llbracket ru_1 \ldots ru_n \rrbracket \\ \supset & \{\text{ Theorem 5.1 }\} \\ & \bigwedge_{i=1}^{n} \llbracket ru_i \rrbracket \\ \supset & \{\text{ Assumptions }\} \\ & \bigwedge_{i=1}^{n} prop \\ \equiv & \{\text{ Predicate reasoning }\} \\ & prop \end{array}$$

□

In case the weak rules do not have enough information, we can use the following stronger rule.

PROOF RULE 4. Let $f(s, o, a)$, $g(s, o, a)$ and $h(s, o, a)$ be defined as in Section 5.1.

$$\frac{\displaystyle\bigwedge_{\substack{s \in Subjects \\ o \in Objects \\ a \in Actions}} \begin{pmatrix} (f(s, o, a) \leftrightarrow \mathsf{Aut}^{+}(s, o, a)) \wedge \\ (g(s, o, a) \leftrightarrow \mathsf{Aut}^{-}(s, o, a)) \wedge \\ (h(s, o, a) \leftrightarrow \mathsf{Aut}(s, o, a)) \end{pmatrix} \supset\; prop}{\llbracket ru_1 \cdots ru_n \rrbracket \;\supset\; prop}$$

Proof. Immediately from the definition of $\llbracket ru_1 \ldots ru_n \rrbracket$. □

Lemma 7.1 is used frequently in the proofs of the case-study and allows us to replace the $\boxed{i}$ in $\leftrightarrow$ by $\Box$ and $\equiv$ when the premise is of length zero.

LEMMA 7.1.
$(\mathsf{finite} \; ; \; (\mathsf{len}(0) \wedge w_1)) \leftrightarrow w_2 \equiv \Box(w_1 \equiv w_2)$

Proof. $(\mathsf{finite} \; ; \; (\mathsf{len}(0) \wedge w_1)) \leftrightarrow w_2$

$$\begin{array}{ll} \equiv & \{\text{by definition}\} \\ & \boxed{i}\; ((\mathsf{finite} \; ; \; (w_1 \wedge \mathsf{empty})) \equiv \mathsf{fin}\; w_2) \\ \equiv & \{(\mathsf{finite} \; ; \; (w_1 \wedge \mathsf{empty})) \equiv \mathsf{fin}\; w_1\} \\ & \boxed{i}\; (\mathsf{fin}\; w_1 \equiv \mathsf{fin}\; w_2) \\ \equiv & \{\text{ITL reasoning}\} \\ & \Box(w_1 \equiv w_2) \end{array}$$

□

7.2. Safety and liveness

Safety and liveness properties have been formally defined by Alpern and Schneider [76]. A liveness property states that something good does eventually happen (i.e. $\chi \mathrel{\widehat{=}} \Diamond f$); while a safety property asserts that something bad never happens (i.e. $\psi \mathrel{\widehat{=}} \Box f$).

We will give a specific safety and liveness property for the EPS system and prove that these properties are valid. Let $authors(A, P)$ denote the predicate $author(A, P) \vee coauthor(A, P)$ in the following.

PROPERTY 1 (Safety). It is never the case that someone modifies a paper without being one of the authors of the paper.

Let $\psi \mathrel{\widehat{=}} \Box(\mathsf{Aut}(A, P, upload(D)) \;\supset\; authors(A, P)))$ denote Property 1.

In the following we will not make explicit the universal quantifiers of variables typeset in uppercase to enhance the readability of the proof outlines. By convention, all these variables are universally quantified, unless explicitly stated otherwise.

A compositional proof that P_4 satisfies ψ can be done using Proof Rule 1, i.e. by proving that each of the P_1, P_2 and P_3 satisfy property ψ.

Proof Rule 2 states that each unless 'phase' has to be finite and has at least two states. But in Section 6 we already made that assumption, i.e. we know $\llbracket \textbf{unless}\; be_i \;\textbf{:}\; P_i \rrbracket \;\supset\; \mathsf{more} \wedge \mathsf{finite}$ for $i = 1, 2, 3$.

We can use now Proof Rule 2 to prove each of the 'unless' phases.

(1) $\llbracket \textbf{unless}\; endSub() \;\textbf{:}\; P_1 \rrbracket \;\supset\; \psi$. We have to prove that $(((\llbracket P_1 \rrbracket \wedge \Box \neg\, endSub()) \, ; \mathsf{skip}) \wedge \mathsf{finite} \wedge \mathsf{fin}\,(endSub())) \;\supset\; \psi$.

Here $\llbracket P_1 \rrbracket$ denotes the subset of the policy containing only rules that can affect the property, viz. rules for $Subjects$; $Objects$ and $\{update(o) | o \in Objects\}$. Other rules cannot cause a violation of the property, as they are independent.

(1.1) Rule $\mathbf{R}_3$ states:
$(\mathsf{finite}; (\mathsf{len}(0) \wedge authors(A, P))) \leftrightarrow \mathsf{Aut}^{+}(A, P, upload(D))$.

Using Lemma 7.1 yields $\Box(authors(A, P) \equiv \mathsf{Aut}^+(A, P, upload(D)))$.

(1.2) Rule $\mathbf{R}_6$ states:
$(\mathsf{finite} \,;\, (\mathsf{len}(0) \wedge \mathsf{Aut}^+(S,O,A) \wedge \neg\mathsf{Aut}^-(S,O,A))) \leftrightarrow \mathsf{Aut}(S,O,A)$. Instantiate it with $A, P, upload(D)$ yields: $(\mathsf{finite} \,;\, (\mathsf{len}(0) \wedge \mathsf{Aut}^+(A, P, upload(D)) \wedge \mathsf{Aut}^-(A, P, upload(D)))) \leftrightarrow \mathsf{Aut}(A, P, upload(D))$. Using Lemma 7.1 yields $\Box((\mathsf{Aut}^+(A, P, upload(D)) \wedge \mathsf{Aut}^-(A, P, upload(D))) \equiv \mathsf{Aut}(A, P, upload(D)))$. This can be simplified to $\Box(\mathsf{Aut}(A, P, upload(D)) \supset \mathsf{Aut}^+(A, P, upload(D)))$. Combining this with (1.1) yields ψ.

(2) $[\![\textbf{unless } endRev() : P_2]\!] \supset \psi$. We have to prove that $((([\![P_2]\!] \wedge \Box\neg endRev()); \mathsf{skip}) \wedge \mathsf{finite} \wedge \mathsf{fin}\,(endRev())) \supset \psi$.

(2.1) Rule $\mathbf{R}_7$ states:
$\mathsf{false} \leftrightarrow \mathsf{Aut}^+(A, P, upload(D))$. Using Lemma 7.1 yields $\Box(\mathsf{false} \equiv \mathsf{Aut}^+(A, P, upload(D)))$. Using (1.2) yields $\Box(\mathsf{false} \equiv \mathsf{Aut}(A, P, upload(D)))$. Using Predicate logic yields ψ.

(3) $[\![\textbf{unless } endAcc() : P_3]\!] \supset \psi$. We have to prove that $((([\![P_3]\!] \wedge \Box\neg endAcc()); \mathsf{skip}) \wedge \mathsf{finite} \wedge \mathsf{fin}\,(endAcc())) \supset \psi$.
(3.1) Rule $\mathbf{R}_{14}$ states:
$(\mathsf{finite} \,;\, (\mathsf{len}(0) \wedge authors(A, P) \wedge accepted(P))) \leftrightarrow \mathsf{Aut}^+(A, P, upload(D))$. Using Lemma 7.1 yields
$\Box((authors(A, P) \wedge accepted(P)) \equiv \mathsf{Aut}^+(A, P, upload(D)))$. Using (1.2) yields ψ.

PROPERTY 2 (Liveness). Eventually a referee is explicitly allowed the right to download a paper he/she is assigned to.

Let $\chi \mathrel{\hat{=}} \Diamond(referee(R, P) \supset \mathsf{Aut}(R, P, download))$ denote Property 2.

A compositional proof that P_4 satisfies χ is given below under the assumption that the paper submission phase is finite, i.e. the event $endSub()$ eventually occurs.

(1) P_2 satisfies the property $\Box(referee(R, P) \supset \mathsf{Aut}(R, P, download))$.

(1.1) Rule $\mathbf{R}_9$ states:
$(\mathsf{finite} \,;\, (\mathsf{len}(0) \wedge referee(R, P)) \vee (\mathsf{finite} \,;\, (\mathsf{len}(0) \wedge authors(R, P)))) \leftrightarrow \mathsf{Aut}^+(R, P, download))$. Using Lemma 7.1 yields $\Box((referee(R, P) \vee authors(R, P)) \equiv \mathsf{Aut}^+(R, P, download))$. Using Predicate logic yields $\Box(referee(R, P) \supset \mathsf{Aut}^+(R, P, download))$.

(1.2) Rule $\mathbf{R}_{10}$ states:
$\mathsf{false} \leftrightarrow \mathsf{Aut}^-(R, P, download)$. Using Lemma s7.1 yields $\Box(\mathsf{false} \equiv \mathsf{Aut}^-(R, P, download))$. Using Predicate logic yields $\Box\neg\mathsf{Aut}^-(R, P, download)$.

(1.3) Rule $\mathbf{R}_6$ states:
$(\mathsf{finite} \,;\, (\mathsf{len}(0) \wedge \mathsf{Aut}^+(S,O,A) \wedge \neg\mathsf{Aut}^-(S,O,A))) \leftrightarrow \mathsf{Aut}(S,O,A)$. Instantiate it with R, P and $download$ yields $(\mathsf{finite} \,;\, (\mathsf{len}(0) \wedge (\mathsf{Aut}^+(R, P, download) \wedge \neg\mathsf{Aut}^-(R, P, download)))) \leftrightarrow \mathsf{Aut}(R, P, download)$. Using Lemma 7.1 yields $\Box((\mathsf{Aut}^+(R, P, download) \wedge \neg\mathsf{Aut}^-(R, P, download)) \equiv \mathsf{Aut}(R, P, download))$. This simplifies to $\Box((\mathsf{Aut}^+(R, P, download) \wedge \neg\mathsf{Aut}^-(R, P, download)) \supset \mathsf{Aut}(R, P, download))$. Using (1.1) and (1.2) yields $\Box(referee(R, P) \supset \mathsf{Aut}(R, P, download))$.

(2) P_4 satisfies Property 2, provided that the paper submission phase is finite, i.e. $(\mathsf{finite} \wedge \mathsf{more} \wedge \Box\neg endSub())\,;\,(endSub() \wedge \neg endRev())$ holds. $[\![P_4]\!] \wedge (\mathsf{finite} \wedge \mathsf{more} \wedge \Box\neg endSub())\,;\,(endSub() \wedge \neg endRev())$. By definition $\Diamond[\![\mathbf{P}_2]\!]\,;\,\mathsf{true}$. Using (1) yields $\Diamond\Box(referee(R, P) \supset \mathsf{Aut}(R, P, download)); \mathsf{true}$. Definition of $\Diamond$ yields $\Diamond(referee(R, P) \supset \mathsf{Aut}(R, P, download))$.

7.3. Policy-based information flow

A generic information flow property of a policy P can be expressed as: $[\![P]\!] \supset e_1 \leadsto^+ e_2$ where $e_1, e_2 \in Subjects \cup Objects$ are entities between which information can flow.

We consider the following information flow property of the EPS system policy.

PROPERTY 3 (Information flow). Information can eventually flow from a referee of a paper to its authors.

This property is important as the authors need to access the referees reviews in order to incorporate them in a revised version of the paper. We construct the proof by decomposing the proof according to the different phases that are defined by the policy. In addition to showing that such a flow is possible, we also derive the assumptions that ensure that this flow takes place.

Proof outline. Show that there is information flow from a reviewer r to the author or coauthor a of a paper: $[\![P_4]\!] \supset r \leadsto^+ a$. By definition 5.3 this means that either there is a direct flow from r to a step (1), **or** there is an indirect flow step (2). Let $authors(A, P)$ denote the predicate $author(A, P) \vee coauthor(A, P)$ in the following.

(1) Direct Flow $[\![P_4]\!] \supset r \leadsto a$: Is not possible as entities cannot interact directly.

(2) Indirect Flow $[\![P_4]\!] \supset \bigvee_{u \in Entities}((r \leadsto^+ u)\,;\,(u \leadsto^+ a))$: Show that there is an object u to which information can flow from the reviewer step (2.1) *and* from which subsequently information can flow to the authors step (2.2).

(2.1) Information flows from reviewers:
Show that information can flow at some point in time from the reviewer to an object in the system: $r \rightsquigarrow^{+} u$. The relevant phase is the *review phase* defined by Policy $\mathbf{P}_2$. Again this flow can be direct or indirect.

(2.1.1) Direct Flow $r \rightsquigarrow u$: The action $write(v)$ on a paper u is a write action. From Definition 5.1 it follows that r must be authorized to perform $write(v)$ on a paper, viz. $[\![P_2]\!] \supset \Diamond\mathsf{Aut}(r, u, write(v))$

— Rule $\mathbf{R}_6$ states:
$(\mathsf{finite}\ ;\ (\mathsf{len}(0) \wedge \mathsf{Aut}^{+}(S,O,A) \wedge \neg\mathsf{Aut}^{-}(S,O,A))) \leftrightarrow \mathsf{Aut}(S,O,A)$. Applying Lemma 7.1 yields: $\Box(\mathsf{Aut}^{+}(r, u, write(v)) \wedge \neg\,\mathsf{Aut}^{-}(r, u, write(v)) \equiv \mathsf{Aut}(r, u, write(v)))$.

— Rule $\mathbf{R}_{10}$ states:
$\Box(\mathsf{Aut}^{+}(r, u, write(v)) \equiv referee(r, u))$. Therefore a necessary condition for information flow from r to u is that $\Diamond referee(r, u)$ during the review phase.

— Rule $\mathbf{R}_{12}$ states:
$(\Diamond\mathrm{done}(cmt, u, accept) \vee \Diamond\mathrm{done}(cmt, u, reject)) \wedge referee(r, u) \leftrightarrow \mathsf{Aut}^{-}(r, u, write(v))$.
The referee is denied to write a review *after* a decision has been made. Therefore information flow from r to u is permissible under the assumption that at some point in the review phase r is a reviewer of the paper u and no decision has been taken by the committee. We formally express this assumption as:
$\psi_1 \mathrel{\widehat{=}} \Diamond\!\!\!\!{\scriptstyle i}\;(\mathsf{more} \wedge \mathsf{keep}\,(\neg\,\mathrm{done}(cmt, u, accept) \wedge \neg\,\mathrm{done}(cmt, u, reject)) \wedge \Diamond referee(r, u))$
where $\Diamond\!\!\!\!{\scriptstyle i}\; f \mathrel{\widehat{=}} f\ ;\ \mathsf{true}$ and $\mathsf{keep}\ f \mathrel{\widehat{=}} \boxed{a}\ (\mathsf{skip} \supset f)$.

— P_2 satisfies the property $r \rightsquigarrow u$ and consequently also the property $r \rightsquigarrow^{+} u$ under the assumption ψ_1.

(2.1.2) Indirect Flow $\bigvee_{u' \in Entities}((r \rightsquigarrow^{+} u')\ ;\ (u' \rightsquigarrow^{+} u))$: *omitted, since direct flow is established by step* (2.1.1)

(2.2) Information flows to authors:
Show that information can flow at some point to the author or coauthors of a paper. The proof step is similar to the step (2.1).

(2.2.1) Direct Flow: The *read* action that can be performed by authors a on a paper u is $read(v)$. The relevant phase is the acceptance phase (Policy $\mathbf{P}_3$), viz. $[\![P_3]\!] \supset u \rightsquigarrow a$.

— Show that: $\Diamond\mathsf{Aut}(a, u, read(v))$ in the acceptance phase. The decision rule $\mathbf{R}_6$ is the same as in step (*2.1.1*).

— Rule $\mathbf{R}_{13}$ states a positive authorization if in the beginning of the review phase the subject is author or coauthor of the paper and v is a review of the paper. Information flow from u to a is therefore permissible with the following necessary assumption on the acceptance phase:
$\psi_2 \mathrel{\widehat{=}} \mathsf{more} \wedge authors(a, u) \wedge review(v, u)$.

— As policy $\mathbf{P}_3$ does not contain any negative authorizations for the action *read* $\mathsf{Aut}^{-}(a, u, read(v))$ is always false.

— P_3 satisfies the property and $u \rightsquigarrow a$ consequently also the property $u \rightsquigarrow^{+} a$ under the assumption ψ_2.

(2.2.2) Indirect Flow: *omitted, since direct flow is established by step* (2.2.1)

(2.3) Clearly from the definition of Policy $\mathbf{P}_4$ the policy P_2 and P_3 are in sequence, thus indirect flow $\bigvee_{u \in Entities}((r \rightsquigarrow^{+} u)\ ;\ (u \rightsquigarrow^{+} a))$ is permissible provided the assumptions ψ_1 and ψ_2 hold in the respective phases.

It remains to show that policy P_2 and P_3 eventually hold, viz. the review phase and the acceptance phase take place. This can be expressed by the following assumption:
$\psi \mathrel{\widehat{=}} (\mathsf{finite} \wedge \mathsf{keep}\,(\neg\, endSub()))\ ;\ (\mathsf{finite} \wedge endSub() \wedge \mathsf{keep}\,(\neg\, endRev()) \wedge \psi_1)\ ;\ (endRev() \wedge \psi_2)$. Here the subformula $(\mathsf{finite} \wedge \mathsf{keep}\,(\neg\, endSub())$ expresses that there is a finite, but not empty, submission phase, which is followed by the review phase initiated through the event $endSub()$. Similarly for the review phase.

The policy P_4 does not satisfy the property $r \rightsquigarrow^{+} a$, viz. $[\![P_4]\!] \supset r \rightsquigarrow^{+} a$ is not valid. However, constraining the EPS with the assumption ψ means that $([\![P_4]\!] \wedge \psi) \supset r \rightsquigarrow^{+} a$ is indeed valid. ψ is a sufficient assumption. As we did not consider all possible information flows in the proof, we cannot claim that ψ is a minimal assumption for the property to be valid.

8. CONCLUSION AND FUTURE WORK

We presented SANTA, a compositional policy language for history-based access control. SANTA can be used to specify a system behaviour in the premise of authorization rules that trigger an access control decision. SANTA supports hybrid policies, viz. policies that contain positive and negative authorizations as well as decision rules to resolve conflicts during the evaluation. The ability to express behaviours as part of the specification removes the need to explicitly model state for the execution history as is, e.g. the case in UCON [37] or [19].

We presented a compositional specification approach for history-based access control policies that allows policies to dynamically change over time and on the basis of events. The key contribution is that policy authors are able to divide the specification and verification of their policies based on specific situations under which the policies apply and then define the transition between policies on the basis of events. This leads to smaller individual policies that are easier to comprehend and to analyse. Although the focus of this paper is the sequential composition, parallel composition as, e.g. in [21] is feasible and an approach has been outlined in Section 4.2.

We also developed a set of compositional proof-rules that can be used to decompose complex verification problems into smaller subproblems for which fully automated verification becomes feasible. Although, we have not provided an automated verification algorithm, the rules can be encoded in, e.g. PVS or for the propositional case in Prover9 and FLCheck (`http://www.tech.dmu.ac.uk/STRL/ITL/`). We showed how these rules can be applied in the verification of safety, liveness and information flow properties in the context of a case-study.

We showed in the EPS system case study how access control requirements are identified and formalized from a given natural language specification. We used the phases that are often a natural element of system descriptions to guide the composition of the system's access control policy. For each phase, we developed a simple policy and subsequently composed the policies to obtain the overall policy. The example of an EPS system has been used by others [20, 77], their specifications consist of a single set of rules with the 'phase' captured as an additional predicate in the rules' premises. In comparison to our approach, this monolithic view complicates policy analysis as additional information is encoded in the premises of rules. Furthermore, the formal semantics of the policy model allows us to reason about the effect changes will have.

In our future work, we will further address issues that arise when composing policies sequentially and in parallel, in that additional constructs such as quantification at policy composition level and policy scoping, viz. the ability to limit the application of a policy to a subset of Subjects, Objects and Actions. With respect to the EPS example this would allow us to specify policies such that they apply to individual submissions as they are used in journal submission processes. Here, the scope of a sequentially composed policy would be reduced to a single article, its authors, and the various actions involved in a journal submission. All of these policies would then be composed in parallel, allowing the individual review processes to be independent.

We are currently implementing the presented proof rules in an automated verification system based on the FLCheck tool which requires an equivalent policy encoding in Fusion Logic, which is a syntactically restricted, but semantically equivalent version of propositional ITL.

We have also developed a runtime validation library ITL-Tracer (http://www.tech.dmu.ac.uk/ heljanic/software.shtml) that provides an efficient evaluation of ITL formulae against recorded system traces. We plan to adapt this technology for the enforcement of SANTA policies.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their supportive and insightful comments.

FUNDING

This work has been partly sponsored by the Data and Information Fusion Defence Technology Centre (DIF-DTC).

REFERENCES

[1] Sandhu, R.S., Coyne, E.J., Feinstein, H.L. and Youman, C.E. (1996) Role-based access control models. *IEEE Comput.*, **29**, 38–47.

[2] Wang, L., Wijesekera, D. and Jajodia, S. (2004) A Logic-Based Framework for Attribute Based Access Control. *Proc. 2004 ACM Workshop on Formal Methods in Security Engineering*, New York, NY, USA, October FMSE'04, pp. 45–55. ACM.

[3] Bandara, A.K., Lupu, E.C. and Sloman, M. (2007) Policy-Based Management. In Burgess, M. and Bergstra, J. (eds), *Handbook of Network and System Administration*. Chapter Policy Based Management. Elsevier, Amsterdam, The Netherlands.

[4] Sloman, M. (1994) Policy driven management for distributed systems. *J. Netw. Syst. Manage.*, **2**, 333–360.

[5] Abadi, M. and Fournet, C. (2003) Access Control Based on Execution History. *10th Annual Network and Distributed System Symp. (NDSS'03)*, Reston, VA, USA, February, pp. 1–15. The Internet Society.

[6] Brewer, D. and Nash, M. (1989) The Chinese Wall Policy. *IEEE Symp. on Research in Security and Privacy*, Oakland, CA, USA, May, pp. 206–214. IEEE.

[7] Bell, D. and Lapadula, L. (1975) Secure Computer System Unified Exposition and Multics Interpretation. Technical Report MTR-2997. MITRE, Bedford, MA.

[8] Harrison, M.A., Ruzzo, W.L. and Ullman, J.D. (1976) Protection in operating systems. *Commun. ACM*, **19**, 461–471.

[9] Denning, D.E. (1976) A lattice model of secure information flow. *Commun. ACM*, **19**, 236–243.

[10] Biba, K.J. (1977) Integrity Considerations for Secure Computer Systems, TR-3153. Technical Report. Mitre Cooperation, Bedford, MA.

[11] OASIS (2005) eXtensible Access Control Markup Language (XACML) Version 2.0.

[12] Damianou, N., Dulay, N., Lupu, E. and Sloman, M. (2001) The Ponder Policy Specification Language. In Sloman, M., Lobo, J. and Lupu, E. (eds), *POLICY*, Bristol, January, Lecture Notes in Computer Science, Vol. 1995, pp. 18–38. Springer.

[13] Twidle, K.P., Lupu, E., Dulay, N. and Sloman, M. (2008) Ponder2—A Policy Environment for Autonomous Pervasive Systems. *POLICY*, Palisades, NY, June, pp. 245–246. IEEE Computer Society.

[14] Abadi, M. (2003) Logic in Access Control. *Proc. 18th Annual Symp. on Logic in Computer Science (LICS'03)*, Ottawa, Canada, June, pp. 228–233. IEEE Computer Society Press.

[15] Becker, M.Y., Fournet, C. and Gordon, A.D. (2006) SecPAL: Design and Semantics of a Decentralized Authorisation Language. Technical Report. Microsoft Research, Roger Needham Building 7 J.J. Thompson Avenue, Cambridge, CB3 0FB, UK.

[16] Alpern, B. and Schneider, F.B. (1987) Recognizing Safety and Liveness. *Distrib. Comput.*, **2**, 117–126.

[17] Halpern, J. and Weissman, V. (2003) Using First-Order Logic to Reason About Policies. *Proc. Computer Security Foundations Workshop (CSFW'03)*, Pacific Grove, CA, USA, July, pp. 187–201. IEEE Computer Society Press.

[18] Becker, M. and Nanz, S. (2007) A Logic for State-Modifying Authorization Policies. In Biskup, J. and López, J. (eds), *Computer Security—ESORICS 2007*, Lecture Notes in Computer Science, Vol. 4734, pp. 203–218. Springer, Berlin/Heidelberg.

[19] Becker, M.Y. (2009) Specification and Analysis of Dynamic Authorisation Policies. *Proc. 2009 22nd IEEE Computer Security Foundations Symp.*, Washington, DC, USA, July, pp. 203–217. IEEE Computer Society.

[20] Dougherty, D.J., Fisler, K. and Krishnamurthi, S. (2006) Specifying and Reasoning about Dynamic Access-Control Policies. In Furbach, U. and Shankar, N. (eds), *IJCAR*, Berlin, August, Lecture Notes in Computer Science, Vol. 4130, pp. 632–646. Springer.

[21] Wijesekera, D. and Jajodia, S. (2003) A propositional policy algebra for access control. *ACM Trans. Inf. Syst. Secur.*, **6**, 286–325.

[22] Park, J. and Sandhu, R.S. (2004) The UCON$_{ABC}$ usage control model. *ACM Trans. Inf. Syst. Secur.*, **7**, 128–174.

[23] Janicke, H., Cau, A., Siewe, F., Zedan, H. and Jones, K. (2006) A Compositional Event & Time-based Policy Model. *Proc. POLICY2006*, London, ON, Canada, June, pp. 173–182. IEEE Computer Society.

[24] Janicke, H., Cau, A., Siewe, F. and Zedan, H. (2007) Deriving Enforcement Mechanisms from Policies. *POLICY*, Bologna, Italy, June, pp. 161–172. IEEE Computer Society.

[25] Moreau, L., Bradshaw, J., Breedy, M., Bunch, L., Hayes, P., Johnson, M., Kulkarni, S., Lott, J., Suri, N. and Uszok, A. (2005) Behavioural Specification of Grid Services with the Kaos Policy Language. *Proc. 5th IEEE Int. Symp. on Cluster Computing and the Grid (CCGrid'05)—Vol. 2*, Washington, DC, USA, May CCGRID'05, pp. 816–823. IEEE Computer Society.

[26] Jajodia, S., Samarati, P., Sapino, M.L. and Subrahmanian, V.S. (2001) Flexible support for multiple access control policies. *ACM Trans. Database Syst.*, **26**, 214–260.

[27] Calo, S. and Lobo, J. (2006) A Basis for Comparing Characteristics of Policy Systems. *7th IEEE Int. Workshop on Policies for Distributed Systems and Networks (POLICY2006)*, London, ON, Canada, June, pp. 183–192. IEEE Computer Society.

[28] Uszok, A., Bradshaw, J.M., Jeffers, R., Suri, N., Hayes, P.J., Breedy, M.R., Bunch, L., Johnson, M., Kulkarni, S. and Lott, J. (2003) Kaos Policy and Domain Services: Toward a Description-Logic Approach to Policy Representation, Deconfliction, and Enforcement. *POLICY*, Lake Como, Italy, June, pp. 93. IEEE Computer Society.

[29] Gong, L., Ellison, G. and Dageforde, M. (2003) *Inside Java 2 Platform Security: Architecture, API Design and Implementation* (2nd edn). Addison-Wesley Professional, Boston, MA, USA. ISBN: 0201787911.

[30] Bell, D. and LaPadula, L. (1973) Secure Computer Systems: Mathematical Foundations. Technical Report. MITRE Corporation, Massachusetts.

[31] Park, J., Zhang, X. and Sandhu, R.S. (2004) Attribute Mutability in Usage Control. In Farkas, C. and Samarati, P. (eds) *Proc. IFIP TC11/WG 11.3 18th Annual Conf. on Data and Applications Security*, Sitges, Catalonia, Spain, July, pp. 15–29. Kluwer.

[32] Lazouski, A., Martinelli, F. and Mori, P. (2010) Usage control in computer security: A survey. *Comput. Sci. Rev.*, **4**, 81–99.

[33] Anderson, J.P. (1972) Computer Security Technology Planning Study. Technical Report. Deputy for Command and Management System, HQ Electronic Systems Division (AFSC), Bedford, MA, USA.

[34] 5200.28 (1985) *Department of Defense Trusted Computer System Evaluation Criteria*. Department of Defense.

[35] Lampson, B.W. (1974) Protection. *SIGOPS Oper. Syst. Rev.*, **8**, 18–24.

[36] Mossakowski, T., Drouineaud, M. and Sohr, K. (2003) A Temporal-Logic Extension of Role-Based Access Control Covering Dynamic Separation of Duties. *10th Int. Symp. on Temporal Representation and Reasoning/4th Int. Conf. on Temporal Logic (TIME-ICTL 2003)*, Cairns, QLD, Australia, July, pp. 83–90. IEEE Computer Society.

[37] Zhang, X., Parisi-Presicce, F., Sandhu, R.S. and Park, J. (2005) Formal model and policy specification of usage control. *ACM Trans. Inf. Syst. Secur.*, **8**, 351–387.

[38] Lamport, L. (1994) The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, **16**, 872–923.

[39] Janicke, H., Cau, A. and Zedan, H. (2007) A Note on the Formalisation of UCON. *Proc. 12th ACM Symp. on Access Control Models and Technologies*, New York, NY, USA, June SACMAT'07, pp. 163–168. ACM.

[40] Li, N. and Mitchell, J.C. (2003) DATALOG with Constraints: A Foundation for Trust Management Languages. *Proc. 5th Int. Symp. on Practical Aspects of Declarative Languages*, London, UK, January PADL'03, pp. 58–73. Springer.

[41] Collinson, M. and Pym, D. (2010) Algebra and logic for access control. *Form. Asp. Comput.*, **22**, 83–104. doi:10.1007/s00165-009-0107-x.

[42] Barker, S. and Stuckey, P.J. (2003) Flexible access control specification with constraint logic programming. *ACM Trans. Inf. Syst. Secur.*, **6**, 501–546.

[43] Bertino, E., Bonatti, P.A. and Ferrari, E. (2001) TRBAC: A temporal role-based access control model. *ACM Trans. Inf. Syst. Secur.*, **4**, 191–233.

[44] Bonatti, P., Vimercati, S. and Samarati, P. (2002) An Algebra for composing access control policies. *ACM Trans. Inf. Syst. Secur.*, **5**, 1–35.

[45] Backes, M., Duermuth, M. and Steinwandt, R. (2004) An Algebra for Composing Enterprise Privacy Policies. *Proc. 9th European Symp. on Research in Computer Security (ESORICS)*, Berlin, September, Lecture Notes in Computer Science, Vol. 3193, pp. 33–52. Springer.

[46] Bruns, G., Dantas, D.S. and Huth, M. (2007) A Simple and Expressive Semantic Framework for Policy Composition in Access Control. *FMSE'07: Proc. 2007 ACM Workshop on Formal Methods in Security Engineering*, New York, NY, USA, November, pp. 12–21. ACM.

[47] Mohan, A. and Blough, D.M. (2010) An Attribute-Based Authorization Policy Framework with Dynamic Conflict Resolution. *Proc. 9th Symp. on Identity and Trust on the Internet*, New York, NY, USA, April IDTRUST'10, pp. 37–50. ACM.

[48] Lupu, E.C. and Sloman, M. (1999) Conflicts in Policy-Based Distributed Systems Management. *IEEE Trans. Softw. Eng.*, **25**, 852–869.

[49] Montangero, C., Reiff-Marganiec, S. and Semini, L. (2008) Logic-based conflict detection for distributed policies. *Fundam. Inf.*, **89**, 511–538.

[50] Craven, R., Lobo, J., Ma, J., Russo, A., Lupu, E. and Bandara, A. (2009) Expressive Policy Analysis with Enhanced System Dynamicity. *Proc. 4th Int. Symp. on Information, Computer, and Communications Security*, New York, NY, USA, March ASIACCS'09, pp. 239–250. ACM.

[51] Woo, T.Y.C. and Lam, S.S. (1993) Authorization in distributed systems: A new approach. *J. Comput. Secur.*, **2**, 107–136.

[52] Jajodia, S., Samarati, P., Subrahmanian, V.S. and Bertino, E. (1997) A Unified Framework for Enforcing Multiple Access Control Policies. *SIGMOD'97: Proc. 1997 ACM SIGMOD Int. Conf. on Management of Data*, New York, NY, USA, May, pp. 474–485. ACM Press.

[53] DeTreville, J. (2002) Binder, A Logic-Based Security Language. *Proc. IEEE Symp. on Security and Privacy*, Oakland, CA, USA, May, pp. 95–105. IEEE Computer Society.

[54] Jim, T. (2001) SD3: A Trust Management System with Certified Evaluation. *Proc. 22th IEEE Symp. on Security and Privacy*, Oakland, CA, May, pp. 106–115. IEEE Computer Society.

[55] Li, N., Grosof, B.N. and Feigenbaum, J. (2003) Delegation logic: a logic-based approach to distributed authorization. *ACM Trans. Inf. Syst. Secur.*, **6**, 128–171.

[56] Garcia-Molina, H., Ullman, J.D. and Widom, J. (2002) *Database Systems: The Complete Book*. Prentice Hall, NJ.

[57] Ceri, S., Gottlob, G. and Tanca, L. (1989) What you always wanted to know about datalog (and never dared to ask). *IEEE Trans. Knowl. Data Eng.*, **1**, 146–166.

[58] Denning, D.E. and Denning, P.J. (1977) Certification of programs for secure information flow. *Commun. ACM*, **20**, 504–513.

[59] Pottier, F. and Conchon, S. (2000) Information flow inference for free. *SIGPLAN Not.*, **35**, 46–57.

[60] Smith, G. (2001) A New Type System for Secure Information Flow. *Proc. 14th IEEE Workshop on Computer Security Foundations*, Washington, DC, USA, June CSFW'01, pp. 115. IEEE Computer Society.

[61] Myers, A.C. (1999) Jflow: Practical Mostly-Static Information Flow Control. *Proc. 26th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, New York, NY, USA, January POPL'99, pp. 228–241. ACM.

[62] Banerjee, A. and Naumann, D.A. (2004) History-Based Access Control and Secure Information Flow. *Proc. 2004 Int. Conf. on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, Berlin, Heidelberg, March CASSIS'04, pp. 27–48. Springer.

[63] Goguen, J.A. and Meseguer, J. (1982) Security Policies and Security Models. *IEEE Symp. on Security and Privacy*, Oakland, CA, USA, April, pp. 11–20. IEEE Computer Society Press.

[64] Agat, J. (2000) Transforming Out Timing Leaks. *Proc. 27th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, New York, NY, USA, January POPL'00, pp. 40–53. ACM.

[65] Sabelfeld, A. and Sands, D. (2000) Probabilistic Noninterference for Multi-Threaded Programs. *Proc. 13th IEEE Workshop on Computer Security Foundations*, Washington, DC, USA, July CSFW'00, pp. 200. IEEE Computer Society.

[66] Sabelfeld, A. and Sands, D. (2001) A per model of secure information flow in sequential programs. *High. Order Symbol. Comput.*, **14**, 59–91.

[67] Joshi, R. and Leino, K.R.M. (2000) A semantic approach to secure information flow. *Sci. Comput. Program.*, **37**, 113–138.

[68] ISO/IEC (2006) ISO/IEC 10181-3:1996 Information technology—Open Systems Interconnection—Security frameworks for open systems: Access control framework.

[69] Harel, D. (1987) Statecharts: a visual formalism for complex systems. *Sci. Comput. Program.*, **8**, 231–274.

[70] Janicke, H., Cau, A., Siewe, F. and Zedan, H. (2008) Concurrent Enforcement of Usage Control Policies. *Proc. 2008 IEEE Workshop on Policies for Distributed Systems and Networks*, Washington, DC, USA, June POLICY'08, pp. 111–118. IEEE Computer Society.

[71] Cau, A., Moszkowski, B. and Zedan, H. (2011) The ITL homepage: http://www.cse.dmu.ac.uk/STRL/ITL. Technical Report. Software Technology Research Laboratory, De Montfort University, The Gateway, Leicester LE19BH, UK.

[72] Sandhu, R. (1988) Transaction Control Expressions for Separation of Duties. *Aerospace Computer Security Applications Conf., 1988, 4th*, Washington, DC, USA, December, pp. 282–286. IEEE Conference Publications.

[73] Bertino, E., Catania, B., Ferrari, E. and Perlasca, P. (2003) A logical framework for reasoning about access control models. *ACM Trans. Inf. Syst. Secur.*, **6**, 71–127.

[74] Siewe, F. (2005) A compositional framework for the development of secure access control systems. PhD Thesis, Software Technology Research Laboratory, Department of Computer Science and Engineering, De Montfort University, Leicester.

[75] Janicke, H.T. (2007) The development of secure multi-agent systems. PhD Thesis, De Montfort University.

[76] Alpern, B. and Schneider, F.B. (1985) Defining liveness. *Inf. Process. Lett.*, **21**, 181–185.

[77] Qunoo, H. and Ryan, M. (2010) Modelling Dynamic Access Control Policies for Web-Based Collaborative Systems. *Proc. 24th Annual IFIP WG 11.3 Working Conf. on Data and Applications Security and Privacy*, Berlin, Heidelberg, June DBSec'10, pp. 295–302. Springer.

APPENDIX 1. SEMANTICS OF EPS POLICIES P_1, P_2 AND P_3

The following is a mapping from the SANTA policy language used to express the EPS policies into their formal ITL semantics. The proofs in Section 7 are using the semantic representation of policies.

Semantics of P_1

$$[\![P_1]\!] \equiv [\![R_1 \ldots R_6]\!] \equiv \bigwedge_{\substack{s \in \mathit{Subjects} \\ o \in \mathit{Objects} \\ a \in \mathit{Actions}}} \begin{pmatrix} (f_1(s,o,a) \leftrightarrow \mathsf{Aut}^+(s,o,a)) \wedge \\ (g_1(s,o,a) \leftrightarrow \mathsf{Aut}^-(s,o,a)) \wedge \\ (h_1(s,o,a) \leftrightarrow \mathsf{Aut}(s,o,a)) \end{pmatrix}$$

where $f_1(s, o, a)$ is defined as

$f_1(s, o, a)$	(s, o, a)	R
true	$(S, eps, register(O))$	1
⟦0: author(**S**,**O**)⟧	$(S, O, add(Sauthor))$	2
⟦0: author(**S**,**O**)⟧	$(S, O, remove(Sauthor))$	2
⟦0: coauthor(**S**,**O**)⟧	$(S, O, remove(S))$	2
⟦0: (author(**S**,**O**) **or** coauthor(**S**,**O**))⟧	$(S, O, upload(Ov1))$	3
⟦0: (author(**S**,**O**) **or** coauthor(**S**,**O**))⟧	$(S, O, download)$	4
false	otherwise	

and $g_1(s, o, a)$ is defined as

$g_1(s, o, a)$	(s, o, a)	R
⟦**exists** x **in subjects :(** x <> **S and done**(x,**O**,upload(**O**v2)) **and always not done(S**,**O**,download))⟧	$(S, O, upload(Ov2))$	5
false	otherwise	

and $h_1(s, o, a)$ is defined as

$h_1(s, o, a)$	(s, o, a)	R
⟦0: (**allow**(**S**,**O**,**A**) **and not deny**(**S**,**O**,**A**))⟧	(S, O, A)	6

Semantics of P_2

$$[\![P_2]\!] \equiv [\![R_4, R_6, R_7 \dots R_{12}]\!] \equiv \bigwedge_{\substack{s \in Subjects \\ o \in Objects \\ a \in Actions}} \begin{pmatrix} (f_2(s, o, a) \leftrightarrow \mathsf{Aut}^+(s, o, a)) \wedge \\ (g_2(s, o, a) \leftrightarrow \mathsf{Aut}^-(s, o, a)) \wedge \\ (h_2(s, o, a) \leftrightarrow \mathsf{Aut}(s, o, a)) \end{pmatrix},$$

where $f_2(s, o, a)$ is defined as

$f_2(s, o, a)$	(s, o, a)	R
⟦0: (author(**S**,**O**) **or** coauthor(**S**,**O**))⟧	$(S, O, download)$	4
true	$(cmt, O, assign(Srev))$	8
true	$(cmt, O, withdraw(Srev))$	8
⟦0: review(**O**rev,**O**)⟧	$(cmt, O, read(Orev))$	8
⟦0: referee(**S**,**O**)⟧	$(S, O, download)$	10
⟦0: referee(**S**,**O**)⟧	$(S, O, write(rev))$	10
⟦0: owner(**S**,**O**rev) **and** review(**O**rev,**O**)⟧	$(S, O, read(Orev))$	10
true	$(cmt, O, accept)$	11
true	$(cmt, O, reject)$	11
false	otherwise	

and $g_2(s, o, a)$ is defined as

$g_2(s, o, a)$	(s, o, a)	R
true	$(S, EPS, register(O))$	7
true	$(S, EPS, upload(Ov))$	7
true	$(S, EPS, add(Sauthor))$	7
true	$(S, EPS, remove(Sauthor))$	7
⟦0: (author(**S**x,**O**) **or** coauthor(**S**x,**O**))⟧ ∨ ⟦0: (**exists** y **in subjects : (** author(y,**O**) **and** institute(y) = institute(Sx)))⟧	$(S, O, assign(Sx))$	9
⟦**sometime done**(cmt,**O**,accept) **and** referee(**S**,**O**)⟧ ∨ ⟦**sometime done**(cmt,**O**,reject) **and** referee(**S**,**O**)⟧	$(S, O, write(Orev))$	12
false	otherwise	

and $h_2(s, o, a)$ is defined as

$h_2(s, o, a)$	(s, o, a)	
⟦0: (**allow**(**S**,**O**,**A**) **and not deny**(**S**,**O**,**A**))⟧	(S, O, A)	6

Semantics of P_3

$$[\![P_2]\!] \equiv [\![R_4, R_5, R_6, R_{13}, R_{14}]\!] \equiv \bigwedge_{\substack{s \in Subjects \\ o \in Objects \\ a \in Actions}} \begin{pmatrix} (f_2(s, o, a) \leftrightarrow \mathsf{Aut}^+(s, o, a)) \wedge \\ (g_2(s, o, a) \leftrightarrow \mathsf{Aut}^-(s, o, a)) \wedge \\ (h_2(s, o, a) \leftrightarrow \mathsf{Aut}(s, o, a)) \end{pmatrix}$$

where $f_2(s, o, a)$ is defined as

$f_2(s, o, a)$	(s, o, a)	R
⟦0: (author(**S**,**O**) **or** coauthor(**S**,**O**))⟧	$(S, O, download)$	4
⟦(author(**S**,**O**) **or** coauthor(**S**,**O**)) **and** review(**O**v,**O**)⟧	$(S, O, read(Ov))$	13
⟦(author(**S**,**O**) **or** coauthor(**S**,**O**)) **and** accepted(**O**)⟧	$(S, O, upload(Ov))$	14
false	otherwise	

and $g_2(s, o, a)$ is defined as

$g_2(s, o, a)$	(s, o, a)	R
⟦**exists** x **in subjects :(** x <> **S and done**(x,**O**,upload(**O**v1)) **and always not done(S**,**O**,download))⟧	$(S, O, upload(Ov2))$	5
false	otherwise	

and $h_2(s, o, a)$ is defined as

$h_2(s, o, a)$	(s, o, a)	
⟦0: (**allow**(**S**,**O**,**A**) **and not deny**(**S**,**O**,**A**))⟧	(S, O, A)	6