An approach to correctness of security and operational business policies☆

V.R. Karimi, D.D. Cowan⁎, P.S.C. Alencar

David R. Cheriton School of Computer Science, University of Waterloo

ARTICLE INFO

Article history:
Received 1 June 2013
Received in revised form 22 January 2014
Accepted 15 May 2014
Available online 12 July 2014

Keywords:
Security policies
REA
Correctness of security policies

ABSTRACT

In this paper we have proposed an approach to describing security and operational business policies and verifying their correctness with respect to a set of properties. The method is based on the REA business modeling language to construct definitions of security and operational business rules. Once the rules are created their representations are combined into policies and policy sets using state machines.

© 2014 Elsevier Inc. All rights reserved.

1. Introduction

One of the fundamental goals of software engineering is to provide systematic and disciplined approaches for the development of real-world software systems. In contrast with ad hoc approaches, these methods can benefit modern organizations in many ways since they offer techniques that can, for example, be used to guarantee that the software meets specific organizational requirements and works correctly with respect to the expectation of organizational stakeholders.

Providing such guarantees becomes a significant challenge when we consider large and complex modern software such as enterprise resource planning (ERP) systems, which have thousands of control requirements that need to be managed (Gal et al.). These controls involve a number of different aspects, some of which are related to the organization's business processes involving access control and proper business operation. Although frameworks such as COSO and CoBIT have been proposed for the evaluation of an organization's internal controls and emphasize the need to manage these controls properly, there is a need for approaches

☆ The authors thank Graham Gal and Malik Datardina for their valuable comments and suggestions, as discussants of an earlier version of this paper presented at the 2013 University of Waterloo Research Symposium on Information Integrity & Information Systems Assurance.

⁎ Corresponding author.

E-mail addresses: vrkarimi@uwaterloo.ca (V.R. Karimi), dcowan@uwaterloo.ca (D.D. Cowan), palencar@uwaterloo.ca (P.S.C. Alencar).

http://dx.doi.org/10.1016/j.accinf.2014.05.004
1467-0895/© 2014 Elsevier Inc. All rights reserved.
that can make the specification of policy-related controls more systematic and support automated control guarantees. In this paper we provide an approach that can be used to guarantee the correctness of business policies involving access control security and organizational process operation. In general terms, this work helps to bridge the gap between the software world supported by formal mathematical models and the corporate accounting world.

Overall, the proposed approach provides a formal method of specifying and evaluating business policies in ERP and similar automated environments. From the standpoint of a real world business perspective, the approach can also benefit practitioners that often need to make sure their access control and operational business policies work as expected. Further, the proposed approach can also help to alleviate problems such as unauthorized access control and dissemination, which are increasingly becoming a threat to modern information systems and can lead to costly and disastrous consequences. In addition, the complexity and large number of policies in real world applications make the use of manual checking not feasible in most cases.

We refer to a policy as a statement that guides decision making and indicates the general direction of an enterprise and is usually in the form of a procedure or protocol. Policies that assist in objective decision-making are usually operational in nature and can be objectively tested. For example a password policy is an objective operational policy.

Corporate operational policies that describe who can perform what actions on what objects are problematic especially when they are implemented in software. Policies incorporated into an enterprise resource planning (ERP) system or across a financial institution can be especially troublesome as a large percentage of policies are buried in software and hidden from human scrutiny. Mergers and acquisitions are also a problem as firms now need to confirm that policies are correct as they are amalgamated.

Testing is one way to have limited confidence in the correctness of a policy's software implementation. However, testing shows that the software passes the test; it does not show that the implementation has overall correct behavior in a given situation. To quote Dijkstra (Dijkstra, 1972) “testing shows the presence of bugs (errors) not their absence.” Even policies that are primarily implemented by people can have incorrect behaviour, although there is the basic safeguard of individuals deciding whether what a policy permits makes sense.

Software engineering practitioners have developed mathematical approaches and tools (sometimes called formal methods) over the last four decades that have allowed them to determine the correctness of portions of a software system. These methods have been applied most often in safety critical systems such as ones controlling aircraft or nuclear power plants. For example, the description of the program or software system is translated into mathematical logic, and then this version of the software is checked to see if it satisfies certain properties. However, these techniques are not readily accessible to the business community as they require advanced mathematical knowledge to apply them and are quite expensive to use in terms of time and expertise.

Recent work by Karimi (Karimi, 2012) on policies related to access control has indicated a direction that looks quite promising in its ability to make a significant subset of these formal methods available to the accounting and business community to check the correctness of operational policies in general. Although there is still some mathematical logic involved its presence in the methods has been substantially reduced through the use of patterns (Karimi, 2012).

This paper outlines the approach by describing the overall model and then applying it to role-based access control (RBAC) (Ferraiolo and Kuhn, 1992). RBAC is a model underlying operational security policies frequently used in business software systems to control access and updates to specific business information. In this case RBAC or similar access control models are used to create the access control rules for a business, and then these rules are combined into business security policies, which are further made into policy sets. For example, a rule is: “a teller may deposit a customer's money into the customer's account” or another rule is: “an account representative may deposit money and may also change personal information.” In contrast a simple policy would be “a teller or account representative may deposit money into a customer's account and an account representative may change personal account information.” A policy set may be: “a manager has all the privileges of an account representative and may also open accounts” along with the previously mentioned policies about tellers and account representatives. Of course policies and policy sets are significantly more complex than these examples.

In this paper we have deliberately kept the examples simple so as not to overcomplicate the methods being demonstrated. For more complex examples the reader can refer to (Karimi, 2012).
2. The approach

The approach just described is based on the resource-event-agent (REA) model (McCarthy, 1982; Hruby, 2006), which was developed by McCarthy and his colleagues in the business and accounting community to describe business systems. Once the rules are described in REA they are formed into policies and a policy set using a state machine, which is a very simple concept. This total model is encoded and provided as input to a software tool called the SPIN model checker (Holzmann, 2004). Access control properties that need to be verified are written in English and then easily translated into a simple form of mathematical logic (Huth and Ryan, 2004). These properties can be classified on the basis of the different kinds of patterns of relevant properties that can be verified against policies and their combinations. The properties are then input to the model checker, and the output from the program will determine if the property is valid. An example of a property is: “a teller can always deposit into a customer account?” Fig. 1 illustrates this approach to checking properties of business policies.

Although the approach and its models are based on formal mathematical concepts, it is relevant to practitioners in the sense that it allows automated checking of access control (i.e., RBAC) properties, which is important in the context of providing automated access control guarantees and rule compliance.

In practice, the access control rules and policies are defined by the designers of the specific access control system. These rules are sometimes provided in formats such as XACML (Organization for the Advancement of Structured Information Standards (OASIS) and Moses, 2005; Organization for the Advancement of Structured Information Standards (OASIS) and Rissanen, 2013). In other cases, these rules can be obtained by analyzing access control mechanisms and systems implemented in software systems such as SAP or Oracle, or can be obtained by a direct analysis of an organization’s policy requirements. In general, the access control policies and rules are created based on specific organizational, system and process requirements, and for this reason, we do not assume that there are “right rules” since rules that are appropriate in the specific case of one firm may not be appropriate in others.

3. Resources, events, agents (REA)

Resources, events, agents (REA) is a model of how an accounting system can be re-engineered for the computer age. REA was originally proposed in 1982 by McCarthy as a generalized accounting model, and is a popular model in teaching accounting information systems. REA can be described by entity relationship models (Chen, 1976) or UML diagrams (Blaha and Rumbaugh, 2005).

The REA model does not use certain standard accounting objects. Many artifacts of the debits and credits double-entry bookkeeping system including general ledger accounts such as accounts receivable or accounts payable are not modeled in the REA approach as persistent objects or database entries since they can be generated by the computer.

REA treats the accounting system as a model of the actual business. In other words, it creates computer objects that directly represent real-world-business objects. The real objects included in the REA model as shown in Fig. 2 are:

- resources - goods, services or money
- events - sales or maintenance activities
- agents - people or other human agencies such as companies that provide or receive events

A REA model as shown in Fig. 2 has a pair of events, linked by a “duality” relation. One of these events usually represents a resource being given away or lost, while the other represents a resource being received or gained.
In the sales process, such as the sale of a book as shown in Fig. 3 (Hruby, 2006) one event would be “BookSale,” where goods are provided, and the other would be “Cash Received,” where cash is provided in exchange for the book. Thus, these two events are linked; a cash receipt occurs in exchange for a sale, and vice versa. The duality relationship can be more complex as in the manufacturing process, where more than two events are involved. These objects contrast with conventional accounting terms such as asset or liability, which are less directly tied to real-world objects.

There is a separate REA pattern for each type of business process in the company or organization. A business process roughly corresponds to a functional department, or a function. Examples of business processes would be sales, purchases, conversion or manufacturing, human resources, and financing. An example of a sales or exchange pattern or process, that is exchanging cash for a commodity, is in the example in Fig. 3. The patterns are extended to encompass commitments (promises to engage in transactions, e.g., a sales order), policies, and other constructs. Two good overviews can be found in (Dunn et al., 2005; Hruby, 2006).

Since REA systems can be modeled using entity-relationship diagrams or UML, they can be implemented as relational or object-oriented databases.

4. Role-based access control

The RBAC model (Ferraiolo and Kuhn, 1992) was introduced in 1992 as a generalized model of access control by adapting existing role-based access control approaches. RBAC (Ferraiolo et al., 2001) introduces roles between users and permissions, and permissions are assigned to roles instead of to users. RBAC is a conceptually simple model in which access to an object such as a bank account is determined by the role of a subject such as teller. For example, Sally (user) is a teller (role), and a teller can deposit funds into a bank account (object). “Deposit funds into a bank account” is an operation (deposit funds into) on an object (bank account) and together they constitute a permission. Therefore a teller can deposit funds into a bank account. Basically management creates roles and permissions on classes of objects and then assigns users to the roles.

This arrangement makes permission assignment easier because permissions related to roles change less frequently than permissions related to users (i.e., people change jobs or are assigned to various roles more often than permissions for roles would be changed). In addition, an estimate indicates that the number of roles is about 3–4% of the number of users (Schaad et al., 2001); and over time each individual (user) can take multiple roles in an organization. RBAC has been used frequently in commercial systems.
Fig. 4 provides a simple picture of the RBAC model. The double arrow connecting users and roles indicates that users can be assigned roles and roles can be assigned to users, in other words there is a two-way relationship between users and roles. The double arrow connecting roles and permissions (PRMS) indicates a two-way relationship between these two entities where roles can have permissions and permissions are assigned to roles. Finally permissions are modeled as operations (OPS) on objects (OBS); again there is a two-way relationship between operations and objects. Additional constraints may be applied as well to the model, and roles can be combined into a hierarchy where higher-level roles subsume permissions owned by lower-level roles. However the simple picture in Fig. 4 is adequate for our explanation.

The next step is translating the RBAC model into the REA modeling language. Each type of role is an agent (A), and each type of object is a resource (R). The event (E) is the occurrence of an operation on a resource. The permission is whether the event related to an object can occur. As well as these three objects we must show that the agents, resources and events are related. A relationship exists between resources and events and between agents and permissions. We can simplify the relationship between agents and permissions to a relationship between agents and events, because the event is part of the permission and is connected to the resource. We will use these relations later in developing our approach further. These relations can be written as Rel(R,E) and Rel(A,E). In RBAC a subject or person holding a role such as a teller has permission to complete an action or event such as deposit money in an account. In this example the account is the resource (R), deposit money is an event (E) and the teller is the agent (A). Thus, there is a mapping from the RBAC policy into the REA modeling language. In the next section we illustrate how to formalize the RBAC policy in terms of REA so that it can be processed by a model checker.

5. Translating rules in English into REA machine-processable statements

Although the rule “a teller has permission to deposit funds into an account” is clear enough and could probably be processed directly by a computer; more complex rules and policies written in English will be impossible to parse and divide into manageable components. Therefore we must have a structured approach to expressing rules and policies. In this section we focus on rules and then in the next section show how these can be combined into policies. We will use the earlier statement about tellers and depositing money into an account to illustrate the approach. We continue to use the simple banking example so that the principles of our approach are clear. For more complex examples see (Karimi, 2012).

Example: A teller can deposit funds into savings accounts.

First let us identify the resource, event and agent and show how we can encode them for machine processing.

- A savings account represents a resource; so we encode it as Resource(savings accounts).
- Deposit is a verb and represents an event and is written as Event(deposit).
- A teller represents an agent, and so we write Agent(teller).

However this first step has only identified the resource, event and agent. We still have to recognize that saving accounts, deposits and tellers are related. There are several possible ways to encode these relationships. We choose to indicate that tellers are related to the event deposit and savings accounts are related to the same event. Since the event deposit is common to both relationships we can infer that tellers are related to savings accounts. We write the relation between agent and event as RELAE(Agent, Event) and the one between resource and event as RELRE(Resource, Event). Thus, the relation between teller (agent) and deposit (event) is written as RELAE(teller, deposit) and between savings account (resource) and deposit (event) as RELRE(savings account, deposit).
We have specified the resource, event and agent and their relationships. However we are still missing one component, namely the statement that deposit access is permitted. This is encoded as DepositAccess(Permit).

We can now write that if the agent is a teller and if the resource is an account and if the event is deposit and there is a relationship between teller and deposit and between savings account and deposit, then deposit is permitted. This statement can be encoded as:

Resource(savings accounts) and Event(deposit) and Agent(teller) and RELRE(savings account, deposit) and RELAE(teller, deposit) implies DepositAccess(Permit)

In mathematical notation the statement would appear as:

Resource(savings accounts) $\land$ Event(deposit) $\land$ Agent(teller) $\land$ RELRE(savings account, deposit) $\land$ RELAE(teller, deposit) $\rightarrow$ DepositAccess(Permit)

In this form “and” has been replaced by “$\land$” and implies by “$\rightarrow$.”

One can view this rule as an assumption followed by a conclusion. The assumption is that we have an agent(teller), resource(savings account) and event(deposit) and the relationships among them. Further if these all hold, then the conclusion is that deposit is permitted (DepositAccess(Permit)).

What we have specified is a general rule for tellers depositing funds into a savings account. We could also specify that tellers could only deposit funds into accounts held at the branch where they work. The rule is meant to specify categories of accounts and to limit the range of actions that can be performed.

6. Policy representations and combinations using state machines

Policies are combinations of one or more rules, and policy sets are combinations of policies. Once a model of a policy or policy set is created it can be queried to see if it has certain properties. This process is called property verification. A property is stated (see Section 8) in much the same way we have described rules, and then the property is verified by running or checking it against the policy set as shown in Fig. 1.

Rules and policies can be organized into policy sets in different ways depending on how the business wishes to operate. In one possible option we can check a property or query such as “can a manager open an account?” In this case we can look through the policy set until we match that query, determine whether it is permitted or denied and then not proceed further. Of course if we never match the query then the property does not exist in the policy set, and we say it is “does not hold.” This organization of rules into policies and policy sets is called “first-applicable,” in that the outcome is determined the first time the query is encountered.

Another possibility exists when a business adds a rule that negates a rule that was permitted earlier in the policy set. In this case we want to examine all rules in a policy to make sure that one rule that is approved is not negated by a later rule. Such an organization of rules and policies is called “ordered-deny-overrides.”

Another further possible organizational strategy is called weak-majority. This scheme states that we must proceed through all the policies and record every time a rule is permitted and every time one is denied. If a rule is permitted more times than it is denied then rule is permitted. In this case the evaluation must record the history of each permit or deny decision.

The state machine concept that is described next supports all ways of organizing rules (Organization for the Advancement of Structured Information Standards (OASIS) and Moses, 2005; Organization for the Advancement of Structured Information Standards (OASIS) and Rissanen, 2013; Wang et al., 2009) that have been reported in the literature including the ones just mentioned. Rules are combined into policies using state machines, and policy sets can be composed from policies also using the same state machine concept. A state machine can be represented as a diagram consisting of nodes (or states) and transitions. Two states, $s_{10}$ and $s_{11}$, are shown in Fig. 5 where the states are represented by rounded rectangles with the names $s_{10}$ and $s_{11}$ inside.

A transition from one node or state to the other is shown by a uni-directional labeled arrow. For instance, a transition from node or state $s_{10}$ to $s_{11}$ with a label “timeout” is shown in Fig. 6. The label is usually interpreted literally; in this case a time has expired and caused the transition from $s_{10}$ to $s_{11}$.

We now provide examples to describe how to represent and combine policies using state machines.

A simple example: A teller can deposit in savings accounts. This rule or simple policy can be viewed in two parts: an assumption or premise and a conclusion. The assumption for this example consists of the fact
that there is a teller, deposit, and savings accounts and there are relationships among them as described in Section 5. The conclusion is whether the event deposit is permitted or denied. In this example access to deposit is permitted. We repeat the rule from the end of Section 5 in two parts.

Assumption or premise (p-rule1): Resource(savings accounts) \( \wedge \) Event(deposit) \( \wedge \) Agent(teller) \( \wedge \) RELRE(savings accounts, deposit) \( \wedge \) RELAE(teller, deposit)

Conclusion or consequence (q-rule1): DepositAccess(Permit)

This policy can be represented as shown in Fig. 7. State s00 in this figure contains the text or pattern that describes the assumption of the rule. When a property (to be described in Section 8) is compared against the premise of the rule it either matches or it does not. If there is a match the next state to be entered is s11 as the premise (p-rule1) is true. If there is no match then p-rule1 is false, and state s10 is entered. For this example, it means that we are dealing with a policy that indicates that an agent is a teller, the resource is savings accounts, the event is deposit and there is a relationship among these three entities.

If there is a transition to state s11 then it will be decided if the conclusion (q-rule1) matches the similar text in the property being examined. In this example, q-rule1 = permit is the outcome as access is allowed that is, a teller is authorized to deposit into savings accounts. Note that there are two transitions from state s11. These transitions go to two final states labeled permit or deny.

We have developed a convention for labeling states. The initial state of the state machine is s00. Once past the initial state the initial digit of the state name is 1 or greater and indicate(s) the rule number. The last digit indicates whether the assumption of that rule holds (i.e., true = 1) or does not hold (i.e., false = 0). For instance,

- s11 the state in which we arrive if rule 1’s assumption holds.
- s10 the state in which we arrive if rule 1’s assumption does not hold.

Each state, with the exception of final states, can have a transition to another intermediate state sij where it is being evaluated further or to a final state which indicates the outcome of the policy that combines the rules. Such final states are shown as circles with an enlarged black center as in Fig. 7. Although we are restricting our diagrams to two policy outcomes (namely permit and deny), in general other policy outcomes can be defined (e.g., not-applicable). Now, we show how to add further rules and thus states to the state machine through another example.

Example: A loan officer can modify loan accounts. In this case the resource is the loan accounts, the event is modify, and the agent is loan officer, and there is a relationship between loan officer and modify and between loan accounts and modify. We can rewrite this rule as shown next.

Assumption or premise (p-rule2): Resource(loan accounts) \( \wedge \) Event(modify) \( \wedge \) Agent(loan officer) \( \wedge \) RELRE(loan accounts, modify) \( \wedge \) RELAE(loan officer, modify)

Conclusion or consequence (q-rule2): ModifyAccess(Permit)

Fig. 8 shows the augmented state machine. In this case the pattern representing p-rule2 is contained in state s10. If the property matches the pattern then p-rule2 is true, and the state machine enters state s21. If the property does not match the pattern then p-rule2 is false, and the state machine enters s20. If the state machine enters state s21 q-rule2 is compared to the pattern of the property. If they match the state machine goes through the transition labeled q-rule2 = permit to the final state permit, otherwise to the final state deny.

Suppose we add a rule that negates a previous rule in the policy set. For example we add the rule:

Assumption or premise (p-rule2): Resource(savings accounts) \( \wedge \) Event(deposit) \( \wedge \) Agent(teller) \( \wedge \) RELRE(savings accounts, deposit) \( \wedge \) RELAE(teller, deposit)

Conclusion or consequence (q-rule2): DepositAccess(Deny).

\footnote{We use q-rule rather than c-rule because of the conventions of our logic representation.}
If we insert this rule later in the policy set using the ordered-deny-overrides combination then we get Fig. 9. In summary, if there is any deny, you stop, and the result is deny, and if there is no deny and at least one permit the result is permit. Note that the evaluation continues reaching the permit-seen state and is shown as transitions from permit-seen to two other states to make sure there is not another rule with a deny result. Of course, if the conclusion of rule 1 is deny, we go from the state $s_{11}$ to deny (not shown in order to make Fig. 9 simpler).

7. Encoding the model in a state machine

The state machines just described in Section 6 can be encoded for the SPIN model checker (Holzmann, 2004) using a language called PROMELA. The PROMELA language strongly resembles the C programming language (Kernighan and Ritchie, 1988). However in this paper we will use a simple version of PROMELA represented as pseudo-code. In other words, programming without all the subtleties that take a long time to learn. The pseudo-code in Fig. 10 shows one possible description of this encoding. In this case, the pseudo-code describes the examples of the state machines in (Figs. 7 and 8). The statement “if premise-rule does not hold” states that the pattern encoded in rule is compared to the property being checked, and if they do not match then we keep trying.

In this case, which encodes the first-applicable (Organization for the Advancement of Structured Information Standards (OASIS) and Moses, 2005; Organization for the Advancement of Structured Information Standards (OASIS) and Rissanen, 2013) combination of rules into the policy set, when a correct policy is found, the procedure is finished, and there is no need to look at other rules. This pseudo-code represents the first applicable state machine which accepts the first property that is correct. Other configurations that involve remembering what has happened as a property is verified can also be encoded. Examples are ordered-permit-overrides (Organization for the Advancement of Structured Information Standards (OASIS) and Moses, 2005; Organization for the Advancement of Structured Information Standards (OASIS) and Rissanen, 2013), ordered-deny-overrides (Organization for the Advancement of Structured Information Standards (OASIS) and Moses, 2005; Organization for the Advancement of Structured Information Standards (OASIS) and Rissanen, 2013), weak-majority and strong-majority (Wang et al., 2009).
8. Properties and checking properties

We have now illustrated how to construct a model that encompasses the rules and policies joined together with a combining scheme. The simple state machines previously shown are examples of such models. Once the model is complete, different questions called properties can be asked and checked against this model. Fig. 1 described earlier is as a pictorial representation of this process.

Just like a database is a model of an organization’s explicit and implicit business data objects and their relationships, our model depicts an organization’s security and operational business rules and their relationships. With a database we can use a language such as SQL (Date, 2003) to specify and ask questions of the model such as show all the accounts receivable that are more than 30 days overdue.

We can ask similar questions about security and operational business rules models. To do this, we can encode a property much like a rule and then provide it as input to the model checker that has been programmed with the REA and state machine representation and combination of the rules, policies and policy sets. An example from earlier in the paper (Section 5) is shown next.

Example assumption or premise (p-rule1): Resource(savings accounts) ∧ Event(deposit) ∧ Agent(teller) ∧ RELRE(savings accounts, deposit) ∧ RELAE(teller, deposit)

Conclusion or consequence (q-rule1): DepositAccess(Permit)

Fig. 8. The combination of policies using state machines with the first applicable combination.

Fig. 9. The combination of policies using state machines with ordered-deny-overrides combination.
We do not know if this property is correct until it has visited one or more states in the state machine representing the model. Therefore we must have a method of specifying this property such that the model checker ensures that it visits all states until the property is either satisfied or not.

Therefore we should rewrite the example as: $\text{Always}(\text{Resource(savings accounts)} \land \text{Event(deposit)} \land \text{Agent(teller)} \land \text{RELRE(savings accounts, deposit)} \land \text{RELAE(teller, deposit)} \rightarrow \text{Eventually(DepositAccess(Permit))})$

These two operators always and eventually are usually written as $\Box$ and $\Diamond$, and are operators in a language called linear temporal logic (LTL) (Huth and Ryan, 2004). LTL is used for the purpose of specifying and querying properties of the models in a manner similar to the approach of creating and processing queries with databases.

Another example of a question or property is “It is always the case if an individual is a teller, then he or she cannot eventually close a loan account.” The property, just stated, without the relationships as described in the previous paragraph can be expressed in LTL as follows:

$$\Box (\text{teller} \land \text{loanaccount} \land \text{close} \rightarrow \Diamond \text{deny}).$$

This property can be converted to a property pattern by substituting agent resource and event into the expression, so it becomes

$$\Box (\text{Agent} \land \text{Resource} \land \text{Event} \rightarrow \Diamond \text{permit}) \lor \Box (\text{Agent} \land \text{Resource} \land \text{Event} \rightarrow \Diamond \text{deny}).$$

Then a property can be expressed as it is always the case that an agent can perform an event on a resource, and this is eventually permitted or denied. There are several other patterns that express properties and that can be used in this way (Karimi, 2012).

These patterns can be seen as a classification of the relevant access control properties that can be verified against policies and their combinations. The proposed classification involves five sub-categories (absence, existence, universality, precedence, and response) based on property patterns (Karimi, 2012). These patterns can help practitioners by providing pre-defined collections of properties that can be reused in order to validate the access control policies. In all patterns such as those shown next, variables like $P$, $Q$, $R$ and $S$ denote expressions based on REA primitive elements (i.e., involving resources, events and actions).
that are evaluated in the context of the state-machine. For example, a universality property can state that a property is valid globally, that is, in all cases and for paths of the associated state machines.

The pre-defined patterns include:

- **Absence patterns**: Patterns asserting that P does not hold globally (i.e., P is false globally); P is false before R; or P is false after R.
- **Universality patterns**: Patterns asserting that P holds globally (i.e., P is true globally); P becomes true before R; or P is true after R.
- **Existence patterns**: Patterns asserting that P becomes true eventually; P becomes true before R; or P becomes true after R.
- **Response patterns**: Patterns asserting that S responds to P globally; S responds to P before R; or S responds to P after Q.
- **Precedence patterns**: Patterns asserting that S precedes P globally; S precedes P before R; or S precedes P after Q.

For more details on the precise specification of these patterns please refer to (Karimi, 2012).

### 9. Related work

Gal et al. (Gal et al., 1991; Gal et al.) use a UML (Fowler, 2003) version of REA and the notion of roles and permissions that also exist in RBAC to specify controls in a business model. A permission is described using the object constraint language (OCL) (Warmer and Kleppe, 2003), a constraint language for object-oriented designs in UML. Part of the specification contains a hierarchical approach that is used to show how an authorization may be delegated. Each permission can put the system into one of several states depending on the degree to which the permission has been violated. Such an approach fits well with assessing the adequacy of internal controls such as required in an audit or when determining whether legislation such as Sarbanes Oxley has been violated. Each rule or policy is treated independently, and there is no discussion of potential interactions among them. This could be a serious drawback when applying the technique to complex software such as typified by an ERP system where as many as 3,000 separate controls could exist (Gal et al.).

In contrast our approach not only models rules but also policies and policy sets which are combinations of rules and policies respectively. Our language is also based on REA concepts to define rules. This language has a formal basis in that the rules are expressed in a mathematical form and then combined to produce policies and policy sets using a mathematical structure called a state machine. The rules and the state machine can be translated into the language of automated tools called model checkers that provide support for policy analysis. In effect the tools and the formal model allow us to check if a policy or policy set satisfies certain properties. In contrast the work described in (Gal et al., 1991; Gal et al.) does not have automated support to examine violations. Although RBAC is used to illustrate our approach, the methods are quite general and should be able to be applied to other business models and policy sets that can be made explicit.

### 10. Conclusions

In terms of compliance, standards such as the ones provided by PCI, E13PA, ISO/IEC 27001/2, and SOC 2 (ISO/IEC, 27001, 2013; Security Standards Council, 2008) address the need to have in place strong access control measures, which allow organizations to permit or deny the use of physical or technical means to access data. Further, these frameworks recommend ensuring that critical data can only be accessed by authorized people, systems and processes, and that procedures are in place to limit access based on factors such as job responsibilities or roles. In this case, access is granted to components and data only to individuals whose job role requires such access, and access control is set to “deny all” unless specifically allowed. In summary, all of the standards assume that in any business or organization in every industry, protecting sensitive and confidential data is a top priority and information security mechanisms involving different forms of access control must be implemented so that related security risks are properly addressed.
In this paper we have proposed an approach to describing security and operational business policies and verifying their correctness with respect to a set of properties. The method is based on the REA business modeling language to construct definitions of security and operational business rules. Once the rules are created their representations are combined into policies and policy sets using state machines.

To check policy set correctness, these policies can then be encoded in such a way that they may be used with a model checker, a software tool that supports automated analysis. This analysis compares a set of desired properties against the model. In order to ensure completeness of the analysis, the properties are expressed in linear temporal logic or LTL. Adding new rules and new properties becomes as simple as adding new states or new properties in LTL.

The provision of such languages as LTL and of tools such as model checkers allows the determination of the correctness of security or operational business policies with respect to a specified set of properties. This is quite advantageous because as the system evolves hundreds of rules and their associated policies can be verified at the push of a button.

Finally, the rigorous nature of the proposed models and the automated support for checking policy correctness can benefit organizations by leading them to: (1) save the time otherwise spent on manual procedures; (2) avoid the costs that could be incurred in case of unauthorized access and dissemination of sensitive and confidential information, especially when the involved risks are high; (3) decrease the risks of security breaches; and (4) boost consumer confidence since when consumers see that the organization is following compliance standards they understand the organization is following the best in terms of business practices. Overall, the proposed approach advances the state-of-the-art by providing theoretical and practical automated methods that can establish the correctness of security and operational business policies.

References


Date CJ. An introduction to database systems. 8th ed. Addison Wesley; 2003.


Organization for the Advancement of Structured Information Standards (OASIS), Moses Tim, editors. eXtensible Access Control Markup Language (XACML), Version 2.0: 2005. [February].

Organization for the Advancement of Structured Information Standards (OASIS), Rissanen Erik, editors. eXtensible Access Control Markup Language (XACML), Version 3.0: 2013. [January].


