RZ 3662 (# 99682) 10/16/2006
Computer Science 20 pages

Research Report

From Regulatory Policies to Event Monitoring Rules: Towards Model-Driven Compliance Automation

Christopher Giblin, Samuel Müller, and Birgit Pfitzmann

IBM Research GmbH
Zurich Research Laboratory
8803 Rüschlikon
Switzerland

From Regulatory Policies to Event Monitoring Rules: Towards Model-Driven Compliance Automation

Christopher Giblin, Samuel Müller, and Birgit Pfitzmann
IBM Research GmbH, Zurich Research Laboratory, 8803 Rüschlikon, Switzerland

Abstract - The complexity and costs of conforming to regulatory objectives in large enterprises has drastically heightened the need for consistent and automated approaches to managing compliance. To uniformly describe and manage compliance policies in distributed and heterogeneous IT environments, we have proposed a compliance metamodel for formally capturing regulatory requirements and managing them in a systematic lifecycle. A key aspect in automating compliance involves the monitoring of application events to determine whether business processes and applications operate within the parameters set forth in formal compliance policies. We show how subsets of the regulations, industry guidances or best practices that are expressed in terms of the metamodel can be (semi-)automatically transformed into event monitoring rules with the help of temporal rule patterns. Using examples of regulatory requirements, we demonstrate their formalization in compliance policies and their automated transformation into event correlation rules.

1 Introduction

Meeting regulatory requirements poses a unique set of challenges for the management of IT systems. Regulations can be complex and in part vague, requiring interpretation. Since corporate compliance is linked to other important business concerns such as strategy, compliance policies need to be integrated in the terms of the business. The technology scope for implementing regulatory objectives may be quite broad, impacting a variety of systems such as storage retention, business processes and applications. In addition, many regulations and compliance standards call for independent review by auditors to assess the effectiveness of control procedures. Regulations and their respective interpretation evolve over time, as do organizations, business conditions, and IT systems. Finally, the complexity in implementing compliance objectives rapidly increases with the number of regulations and the size of an organization. In responding to these requirements, enterprises are seeking to reduce costs through a systematic and, as far as possible, automated approach to compliance.

Regulations Expressed As Logical Models (REALM) [9] is a metamodel for compliance that addresses the above challenges by capturing a wide variety of regulations in a single language and semantics. Using REALM, regulatory requirements can be formalized, in part, directly from regulatory text and compliance guidelines in technology-independent, business terms. We provide a succinct introduction to REALM in Section 2.

While regulatory requirements can be captured in technology-agnostic terms using our metamodel, eventually compliance objectives still have to be implemented by concrete systems using specific technology. In this context, event-monitoring technology allows for monitoring distributed and heterogeneous IT systems in a loosely-coupled manner without the need to integrate disparate systems and components. Accordingly, event monitoring plays an important role with respect to the effective implementation, monitoring, and enforcement of regulatory requirements. Specifically this holds for event correlation, which is software technology designed to observe, classify, and correlate message streams and to inform human or technology recipients of system events so that the recipients can react in an appropriate way.

Bridging this gap, abstract policies expressed as high-level REALM models can, at least partially, be automatically refined into low-level implementation technologies such as correlation rules with the help of model transformation. This allows for efficiently addressing similar classes of requirements and dealing with them in an automated manner.

In this paper, we demonstrate a model-driven approach to automated rules generation for runtime monitoring of compliance objectives. In particular, low-level monitoring rules are derived from certain abstract policy expressions using model transformation and parameterizable temporal rule patterns. Event-oriented middleware capabilities such as event standards and event correlation are exploited in developing components to monitor applications and business processes.

1.1 On Monitoring using Correlation

Compliance rules may be either enforceable or non-enforceable in a certain environment. Enforceable compliance rules can typically also be monitored, whereas non-enforceable compliance rules are partitioned into monitorable and non-monitorable ones. While violations of non-enforceable but monitorable compliance rules cannot be prevented, they can at least be detected. Thus, only the residual subset of non-enforceable and non-monitorable compliance rules require additional measures to attain compliance, which are typically external to the system. A similar classification scheme for usage control of access control obligations has been proposed by Basin et al. [4]. In this paper, we focus on monitorable compliance rules.

In principle, all monitorable compliance rules can be implemented using event-based monitors. Specifically the monitoring of loosely coupled, unstructured work flows, implemented across distributed and heterogeneous IT systems, suggests implementations based on an event architecture and correlation. In this context, a correlation engine is a software component that detects, classifies, and correlates message streams and reacts on predefined patterns as specified in correlation rules. In particular, the monitoring of temporal constraints as frequently occurring in regulations can benefit from such technology. This is true even if such constraints can also be enforced by other technology such as state-of-the-art business process execution engines as suggested in [1]. Here, event-based monitoring can generate early warnings by observing timeouts and thresholds or assure the proper functioning of the enforcement mechanism.

Event-based monitoring and correlation requires the instrumentation of relevant systems (e.g., a business process execution engine) to publish events at predefined state transitions during their execution (e.g., a certain activity is started or terminated, a data object changes its state, a timeout occurs, etc.). The use of event monitoring is particularly adequate when existing systems can be easily configured to emit events in standardized formats or when such systems can be easily extended to incorporate such a capability.

In principle, monitoring only discovers the violation of a formula when it happens. While this might be too late, one may still want to detect it or even be required to monitor by law (e.g., logging for audits). In such cases, one would usually raise an early warning, e.g., by transforming original timeouts into shorter timeouts and thresholds into smaller thresholds.

While some compliance rules may be useful to monitor, a correlation engine may not always be the best way to do so. For instance, where data are entered manually, once would typically enforce their correctness directly at the user interface and reject non-compliant data. Also, instead of monitoring events, a direct evaluation on a system or database state may sometimes be more useful. This holds in cases where it is inefficient to transform all data included in the compliance rules into events. Further, while one main benefit of event correlation with state over pure state evaluation is to be triggered by an event, there are also other triggering mechanisms that need to be considered, e.g., database triggers.

Furthermore, depending on the timing requirements and system parameters, event-basing may not be useful, thereby also rendering the use of correlation impractical. Evaluation at short regular intervals may be sufficient in many cases (i.e., ‘online’ vs. ‘offline’ processing for audit purposes). Especially for audit purposes, real-time monitoring generates an unnecessary overhead. Here, for instance the use of standard SQL queries to evaluate system logs may lead to more efficient results. What is more, even in an event-based system there is a delay between the event in the source system and when the rules have been evaluated on it.

1.2 Content

The remainder of this paper is structured as follows. In the subsequent section, we provide a summary of REALM, our compliance metamodel. We present a simple example to demonstrate the basic idea behind REALM and then focus on temporal REALM rule patterns, which we use as starting point for our model transformations. In Section 3, we briefly explain event-based architectures and monitoring. We then introduce Active Correlation Technology (ACT), a specific instantiation of an event-based correlation system. Next, in Section 4, we describe how temporal REALM rule patterns are automatically transformed into such low-level correlation rules and how such a transformation can be implemented. Finally, we summarize our work and end with a conclusion.

2 The REALM Metamodel for Compliance

Regulations Expressed As Logical Models (REALM) [9] is a metamodel and method being developed by IBM Research, which encompasses requirements engineering of regulations and builds on their rigorous formalization as formal compliance policies. Regulatory requirements are formalized from regulation text as sets of compliance rules in a real-time temporal object logic over concept models.

The REALM approach is based on three central pillars:

- *Concept model.* The domain of discourse of a regulation is captured by a concept model expressed as UML class diagrams using a specific UML profile developed for REALM. The concept model thus plays the role of a complex type system.

- *Compliance rule set.* Regulatory requirements are formalized as a set of logical formulae expressed in a real-time temporal object logic. The logic is a combination of Alur and Henzinger's Timed Propositional Temporal Logic [3] and many-sorted first-order logic. Each sort used in a formula corresponds to a type defined in the concept model.

- *Metadata.* Meta-information about the structure of the legal source (e.g., locator to regulation source such as Act and Section numbers) as well as life-cycle data (e.g., enactment and expiration dates) are captured as separate metadata. The metadata allow annotating the models and individual model elements of both the concept model and the compliance rule set to ensure traceability from regulatory requirements to model elements and vice versa.

REALM provides the basis for subsequent model transformations, deployment, and monitoring of compliance policies. A REALM-based compliance management lifecycle starts with the determination of the *scope* of a given regulation including an evaluation of the respective impact on the enterprise. Next, relevant regulatory requirements are formalized into an *immediate* REALM model, which is then substantiated into a *refined* REALM model. After a thorough *as-is analysis* of the enterprise has resulted in models of relevant parts in scope of the regulation, the impact of the regulation is assessed by comparing the REALM model with the as-is situation. Informed by this *gap analysis*, the REALM model is *transformed* and *deployed* into the identified target systems. Figure 1 depicts the transformation of high-level REALM models into technologies supporting compliance such as storage, processes, and correlation. After a REALM model has been deployed into a target system, compliance is sometimes ensured automatically. However, due to the inherent possibility of human or system error, compliance must almost always also be *monitored* and *enforced* in real time.

In this paper, we focus on the automatic transformation of high-level REALM compliance rules into low-level event monitoring rules (i.e., correlation rules). Next, we present a simple example to illustrate the basic idea behind REALM. We then present some temporal rule patterns that we found in real regulations and which we utilize to select suitable formulae for the automatic transformations described in Section 4.

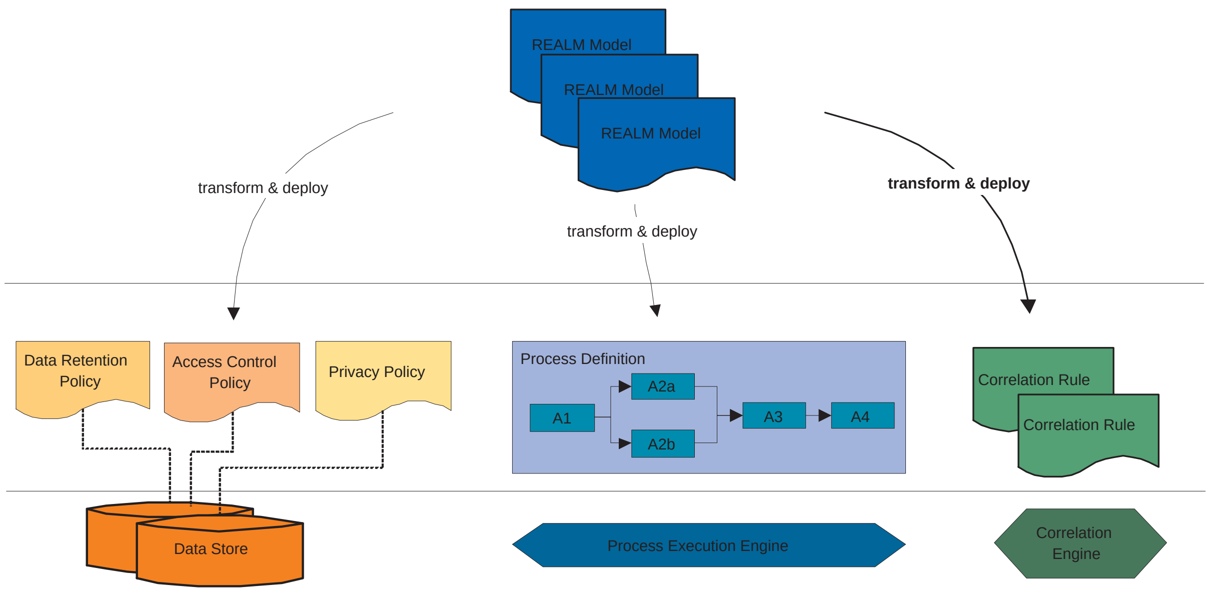

Figure 1: Transformation and Deployment of REALM Models.

2.1 REALM Example

To make the subsequent discussion more tangible, we present a simple example of how regulatory requirements are captured in a REALM model. Later, in Section 4, we explain how parts of such a model that are suitable for event monitoring can be automatically transformed into a correlation rule.

In Figure 2, we present a simplified version of the regulation text in 31 CFR 103.121, which clarifies Section 326 of the USA Patriot Act. We focus on the portion printed in boldface type.

Banks must implement procedures for verifying the identity of each customer; these procedures must ensure that the bank knows the true identity of each customer.
The bank must further implement procedures that specify the identifying information that will be obtained from each customer. At a minimum, the bank must obtain the following information prior to opening an account: name; date of birth; residential address; identification number.
The bank must verify the identity of each customer, using the information obtained in accordance with the above requirements, within a reasonable time after the account is opened.

Figure 2: Simplified regulation text inspired by 31 CFR 103.121.

Figure 3 shows the REALM concept model corresponding to this regulation passage. It captures the high-level concepts that occur in the regulation such as *Bank*, *Customer*, *Account*, *Record*, and the two Action types *Open* and *Verify*.

The requirement *'The bank must verify the identify of each customer, using the information obtained in accordance with the above requirements, within a reasonable time after the account is opened'* is captured in Formula (1). As explained in [9], REALM formulae build upon the Timed Propositional Temporal Logic from [3], which provides a good balance between expressiveness and complexity [2]. Instead of atomic propositions, however, we add an objectmodel, the specialization to actions with lifetimes, and syntactic abbreviations such as the use of time

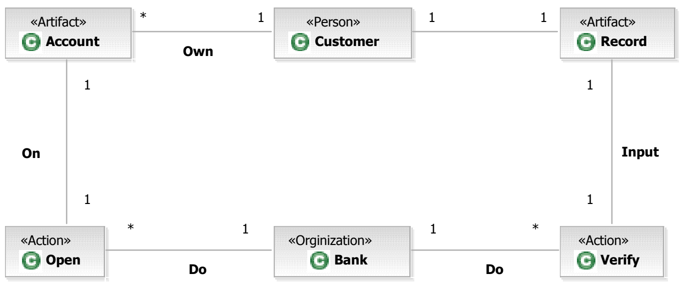

Figure 3: Concept Model for 31 CFR 103.121.

with different units and the combination of state predicates.

In addition, REALM provides several predefined relations over predefined types. For example, *Do* is a predefined abstract relation between a person or organization and an action. We write it $Do(a, b)$, where a is an instance (i.e., a bound variable) of a subtype of *Person* or *Organization* and b an instance of a subtype of *Action*. This predicate evaluates to true if a executes action b at the point in time where the predicate is evaluated. Other important examples of predefined relations are $On(a, b)$, $Input(a, b)$, and $Output(a, b)$ where a is an action and b an artifact; they have the natural meanings. As many time constraints in laws refer to the beginning or the end of an action, we use subscripts '$_S$' and '$_F$' and attach them to the relations, e.g., $Do_S(a, b)$ and $Do_F(a, b)$ evaluate to true only at the point in time where the execution of action b *starts* or *finishes*. If we talk about actions only, we allow attaching these subscripts directly to the action variable. We may then write b_S instead of $Do_S(b)$.

Temporal formulae are built with the usual non real-time temporal modalities such as $\Box$ (always), $\Diamond$ (eventually), and $\blacklozenge$ (once), as well as freeze quantifiers and relations on times. A freeze quantifier corresponds to the introduction of a variable for a point in time. For instance, a formula term $\Diamond t.\phi$ means that eventually formula ϕ will hold, and we introduce the time variable t for this point in time. Time relations within ϕ can refer to t, e.g., to express that t is at most 2 days later than some other time t', introduced similarly with a freeze quantifier.

$$\begin{aligned}&\forall open \in Open, \exists verify \in Verify\\ &\Box t_{open}.\ (open_F \Rightarrow \Diamond t_{verify}.\ (verify_F \wedge\ open.account.customer.record = verify.record\\ &\qquad\qquad\qquad\qquad \wedge\ t_{verify} - t_{open} \leq\ 2_{[day]}))\end{aligned} \tag{1}$$

Hence, Formula (1) states that whenever a bank has opened a new account, eventually it has to finish the verification of the identity of the customer, based on the data collected in the process of opening the account. The formula further states that the verification needs to be completed within 2 days, which we used as the interpretation of "reasonable time" by our example bank. The boolean state predicate $open.account.customer.record = verify.record$ ensures that for every newly opened account there is also a dedicated verification of the customer's identity. Because the formula is only used in the context of one bank, we may write $open_F$ and $verify_F$ instead of $Do_F(bank, open)$ and $Do_F(bank, verify)$.

Formula (1) represents an instance of a temporal REALM pattern. Such patterns predefine a certain temporal structure that often occurs in regulatory settings, while allowing for dedicated parameterization of well-defined points. We now present a number of such patterns and explain how they are used.

2.2 Temporal REALM Rule Patterns

The focus of this paper is the automated transformation of REALM models into correlation rules and their subsequent monitoring. As a transformation into correlation rules is necessary only when we consider such monitoring appropriate, we introduce temporal REALM patterns as a selection tool for suitable REALM compliance rules.

2.2.1 y_Within(c)_After_x

The first pattern we consider can be derived from the regulation text presented in Figure 2 and the corresponding formalization in Formula (1), respectively. The characteristic temporal structure of this pattern is visualized in a time constraint diagram in the upper part of Figure 4. An abstract representation of the pattern is given in Formula (2).

$$\forall x \in X, \exists y \in Y \Box\, t.(x_{\{S,F\}} \Rightarrow \Diamond\, s.(y_{\{S,F\}} \wedge\ pred(x,y) \wedge\ s - t \leq c_{[unit]})) \tag{2}$$

Pattern 'y_Within(c)_After_x' reads as follows: Whenever an instance of some action type X starts or terminates execution, an instance of some action type Y must also start or finish executing within an interval c of time unit *unit*. While the temporal structure is predefined, the exact subtypes of the actions X and Y, as well as the size and unit of the time window can be parameterized. Also the action predicates (i.e., the subscript '$_S$' or '$_F$') can be chosen. In addition, the pattern requires the definition of an abstract boolean state predicate $pred(x, y)$, which is used for correlation.

The example from Formula (1) represents an instance of this pattern, where the first action type X is set to *Open* and the second action type Y is set to *Verify*. The state predicate $pred(x, y)$ is instantiated as $open.account.customer.record = verify.record$, which formalizes the constraint that every instance of *Open* needs a corresponding instance of *Verify* operating on the same *Record* instance, i.e., belonging to the same customer. Finally, the time window c is set to 2 and the time *unit* is defined as *day*.

We call such a temporal REALM formula "pattern" as it occurs in different variations in real regulations. By defining patterns with a predefined temporal structure and "configurable" action types X and Y, a predicate $pred(x, y)$ and time window c and time unit *unit*, we can define automated transformations from parameterized patterns to executable correlation rules.

When temporal REALM rule patterns are transformed into event correlation rules, the correlating event monitor needs some information on which events to react upon and correlate. This information comes from two sources: Predicates $x_{\{S,F\}}$ and $y_{\{S,F\}}$ are transformed into so-called event selectors (e.g., XPath expressions that are evaluated by a correlation engine upon receiving an event message) defining which events are of interest to the correlation engine. Furthermore, events are correlated using information contained in the abstract state predicate $pred(x, y)$. We explain this transformation in more detail in Section 4.

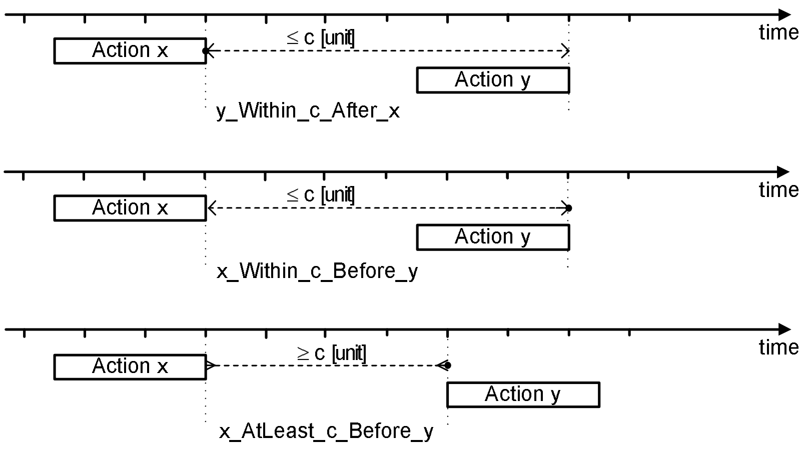

Figure 4: Some temporal REALM patterns.

2.2.2 x_Within(c)_Before_y

The following passage from the Sarbanes-Oxley Act of 2002, Sec. 302, (a)(4)(C), represents another temporal pattern that frequently occurs in regulations:

> *"[...] the signing officers [...] (C) have evaluated the effectiveness of the issuer's internal controls as of a date within 90 days prior to the report."*

The middle part of Figure 4 visualizes the temporal structure inherent in this requirement. Formula (3) formally captures this.

$$\forall y \in Y, \exists x \in X \; \Box t.(y_{\{S,F\}} \Rightarrow \blacklozenge s.(x_{\{S,F\}} \wedge \; pred(x,y) \wedge \; t - s \leq c_{[unit]})) \tag{3}$$

Abstractly speaking, pattern 'x_Within(c)_Before_y' reflects an obligation in the past, which must have been completed within a specific time before the triggering action occurs. The same parts as in the previous pattern can be parameterized, while the temporal structure is fixed.

In the above regulation text, the evaluation of the effectiveness of the issuer's internal controls must be completed within 90 days before they can finish the report. Hence, $x_{\{S,F\}}$ and $y_{\{S,F\}}$ are replaced by some predicates $evaluate_F$ and $report_F$ over instances of action types *Evaluate* and *Report*, respectively. Assuming that reporting and controls effectiveness are only associated with one issuer, the predicate $pred(x, y)$ could be $report.issuer = evaluate.controlsEffectivness.issuer$. Finally, the size and unit of the time window are set to *90* and *day*.

2.2.3 x_AtLeast(c)_Before_y

Finally, we consider the requirements made in SEC17a-4 f.2.i:

> *"If employing any electronic storage media other than optical disk technology (including CD-ROM), the member, broker, or dealer must notify its designated examining authority at least 90 days prior to employing such storage media."*

This is an instance of pattern 'x_AtLeast(c)_Before_y'. A visual representation of its temporal structure is again given by a time constraint diagram in Figure 4.

$$\forall y \in Y, \exists x \in X \ \Box t.(y_{\{S,F\}} \Rightarrow \blacklozenge s.(x_{\{S,F\}} \wedge \ \mathit{pred}(x,y) \wedge \ t - s \geq c_{[unit]})) \qquad (4)$$

In analogy to the previous case, this pattern focuses on the past as seen from the point in time when it was triggered. Applied to the regulation text above, this pattern would be parameterized as follows: Contingent on the actual concept model for this regulation text, $x_{\{S,F\}}$ and $y_{\{S,F\}}$ would be substantiated as predicates employ_S and notify_F over instances of some actions types *Employ* and *NotifyAuthority*, respectively. The predicate $\mathit{pred}(x, y)$ would be used to correlate action instances that relate to the same situations where electronic storage media other than optical disc technology is employed. Assuming the existence of a relation between *Employ* and a dedicated concept *OtherMedia* as well as between *NotifyAuthority* and *OtherMedia* (both with multiplicity 1 in the direction towards *OtherMedia*), this could e.g. be expressed as $\mathit{employ.otherMedia} = \mathit{notify.otherMedia}$.

Having introduced the basic idea behind temporal REALM patterns and their use, in the next section, we briefly review event monitoring technology. In Section 4, we then present the transformation of a REALM pattern into an event correlation rule in detail.

3 Event Monitoring Technology

Informally, we regard an *event* as an occurrence of a phenomenon. An *event message* reports on such an occurrence. Within the context of software systems, event messages are commonly referred simply to as "events" for brevity, abridging the difference between the phenomenon and the report of the phenomenon. Event-oriented programming models accommodate the creation, emitting and reception of such event messages.

The Common Base Event (CBE) [8] is a specification describing a standard event model and format for exchange of event messages between disparate systems. A CBE contains an event type, which indicates what the event is about, and a time-stamp indicating when the event occurred. Additionally, optional event attributes include contextual information such as the component where the event originated, the component reporting the event, software versions and application-specific information. CBEs are emitted by a number of IBM products such as WebSphere Process Server (WPS) and Tivoli Access Manager while IBM development tools such as Rational Software Architecture (RSA) and WebSphere Integration Developer (WID) support programming with CBEs.

A common event model also facilitates the establishment of common infrastructure capabilities for storing and distributing events. These capabilities, in turn, engender new applications which receive, interpret and act on events from various sources. For example, the capability of WPS to emit CBEs at each step of a workflow makes it possible to monitor business process events relating to compliance objectives.

Further examples of IBM products and components providing targeted event-processing capabilities are:

- Common Event Infrastructure (CEI) implements a publish-subscribe event server for persisting and distributing CBEs.
- Common Auditing and Reporting Service (CARS) supports the selection and storage of CBEs for audit reporting.
- Active Correlation Technology (ACT) [6] is an internal component performing event-correlation of CBEs and other types of events.

As ACT appears later in an example, the following section provides an overview of ACT correlation rules.

3.1 Active Correlation Technology

ACT [6] is an IBM Tivoli internal component consisting of a Java-based correlation engine, a rule language, and a rule compiler that translates rule files into a data-structure, the ruleset, which the correlation engine directly evaluates during event-processing.

The ACT component supports several rule types which correspond to common correlation patterns. The UML Class Diagram in Figure 5 illustrates the structure of the ACT ruleset, rule blocks, and rules. As the diagram shows, rules are organized into rule blocks. Rule blocks can be nested. A ruleset is composed of one or more rule blocks.

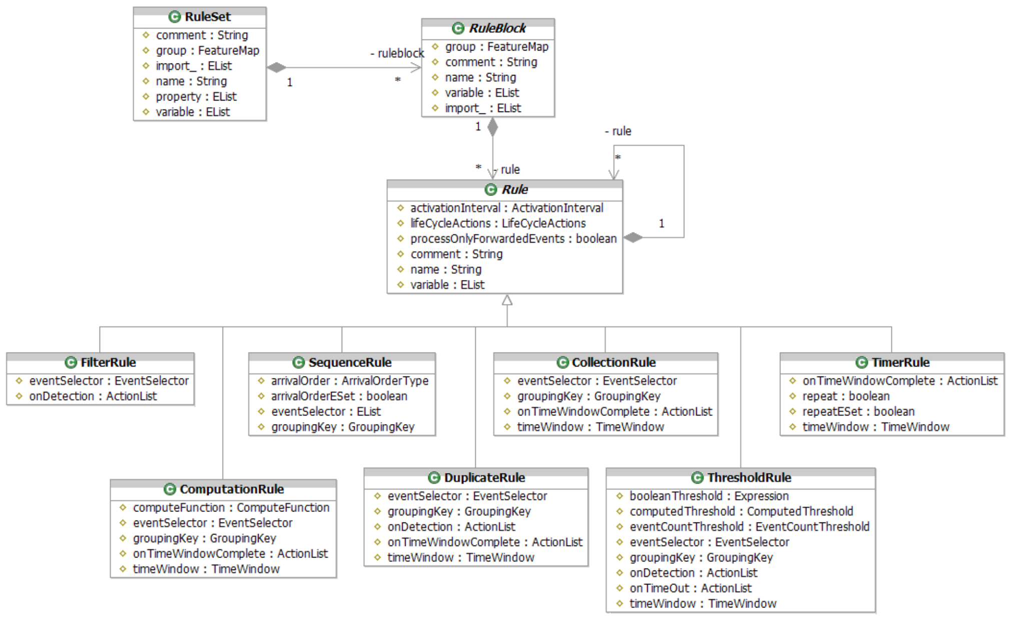
RuleSet
comment : String
group : FeatureMap
import_ : EList
name : String
property : EList
variable : EList
1
- ruleblock
*
RuleBlock
group : FeatureMap
comment : String
name : String
variable : EList
import_ : EList
1
- rule
*
*
rule
Rule
activationInterval : ActivationInterval
lifeCycleActions : LifeCycleActions
processOnlyForwardedEvents : boolean
comment : String
name : String
variable : EList
1
FilterRule
eventSelector : EventSelector
onDetection : ActionList
SequenceRule
arrivalOrder : ArrivalOrderType
arrivalOrderESet : boolean
eventSelector : EList
groupingKey : GroupingKey
CollectionRule
eventSelector : EventSelector
groupingKey : GroupingKey
onTimeWindowComplete : ActionList
timeWindow : TimeWindow
TimerRule
onTimeWindowComplete : ActionList
repeat : boolean
repeatESet : boolean
timeWindow : TimeWindow
ComputationRule
computeFunction : ComputeFunction
eventSelector : EventSelector
groupingKey : GroupingKey
onTimeWindowComplete : ActionList
timeWindow : TimeWindow
DuplicateRule
eventSelector : EventSelector
groupingKey : GroupingKey
onDetection : ActionList
onTimeWindowComplete : ActionList
timeWindow : TimeWindow
ThresholdRule
booleanThreshold : Expression
computedThreshold : ComputedThreshold
eventCountThreshold : EventCountThreshold
eventSelector : EventSelector
groupingKey : GroupingKey
onDetection : ActionList
onTimeOut : ActionList
timeWindow : TimeWindow

Figure 5: UML diagram of ACT ruleset and rule types.

The ACT rule types are briefly summarized below:

Match/Filter identifies whether a single event matches a given characteristic and if so triggers an action.

Collection identifies and collects a number of specific events over a given amount of time. This rule is useful in summarizing events and triggering an action when a situation has occurred n times.

Computation executes a user-defined function upon detection of each event. It is used, for example, to compute running sums, mean values, or statistics.

Duplicate detects duplicate events and suppresses those duplicates during a specifiable time window.

Threshold computes a user-definable function which checks whether a function exceeds or falls below a given threshold.

Sequence looks for a series of events of a given characteristic. The rule allows specification of whether events arrive in a given order and whether the events arrive within a given time interval. If the sequence is detected, a detection action is triggered; if the events do not arrive within the timeout period, a timeout action is triggered.

Timer triggers an action after a specified amount of time. This rule can be used to ensure a certain action is triggered during a given time span after a specific event arrived.

4 Transforming REALM Policies into Monitoring Rules

Model transformation is used to derive executable ACT rule sets from REALM policies. REALM policies are the source model while two artifacts comprise the target models: an ACT rule set and a Java stub. The temporal rule patterns described earlier play an important role in determining the target ACT rule type (Sequential, Duplicate, etc.). The patterns determine the assignment of rule attributes involving time-windows.

The generated Java stub provides a placeholder for execution code called by the ACT rule actions. ACT rule actions are elements of a rule which are executed when the rule triggers. Implemented in a specified language such as Java, action code often contains programmatic considerations specific to a given IT environment. The generation of the stubs allows for custom implementation and separates the maintenance of code from the rule. The transformation can be configured to generate a default implementation, for example, which issues a specific event in response to detecting a compliance violation. Because these stubs implement the actions of a rule, we refer to the stubs hereafter as *action handlers*.

During authoring of a REALM policy, a reference to the regulatory passage(s) to which the policy pertains is optionally included. The transformation looks for this metadata reference and includes it in the target ACT rule set, making the reference available to ACT rule actions. Artifacts created during runtime, such as event messages, can thus bear reference to the policies to which they relate. Uniform Resource Identifier (URI) syntax [5] was chosen as the syntax for these references. URIs lend themselves not only to constructing large namespaces, but also to addressing online resources such as a policy repository.

4.1 Transformation Architecture

The transformation is implemented with the Rational Software Architect (RSA) version 6.0 Transformation Framework [10]. The framework supports Java-based, imperative transformations which can be deployed as RSA plugins. The framework consists of extensible classes implementing the fundamental concepts of source model navigation and target transformation rules. The transformation accepts optional parameters which are entered in an input form prior to execution. An example of a transformation parameter is the name of the Java package for action handler code.

A second framework, the Eclipse Modeling Framework (EMF) [7], is used as a metamodel for the source and target models. EMF provides a Java code-generation facility as well as runtime features which include resource serialization between Java objects and XML.

The transformation architecture, with source and target models, as well as a shared metadata repository for policy, is depicted in Figure 6. The architecture diagram informally layers, from top to bottom, the relationship of metamodel, model, and instance. EMF is seen describing the REALM, Action Handler and ACT rule models. The transformation engine takes instances of REALM policies, shown on the left hand side, and produces event handler and ACT rule instances, on the right hand side. Instances are illustrated with sample content as in the case of the generated sequence rule. The diagram demonstrates how a URI reference, which occurs in the source REALM policy, is mapped to the target ACT rule. The action handler is designed to have access to the context of the rule and can therefore retrieve the regulatory reference URI for use during action execution. In this example, the URIs, in addition to being unique identifiers, point to an online policy repository.

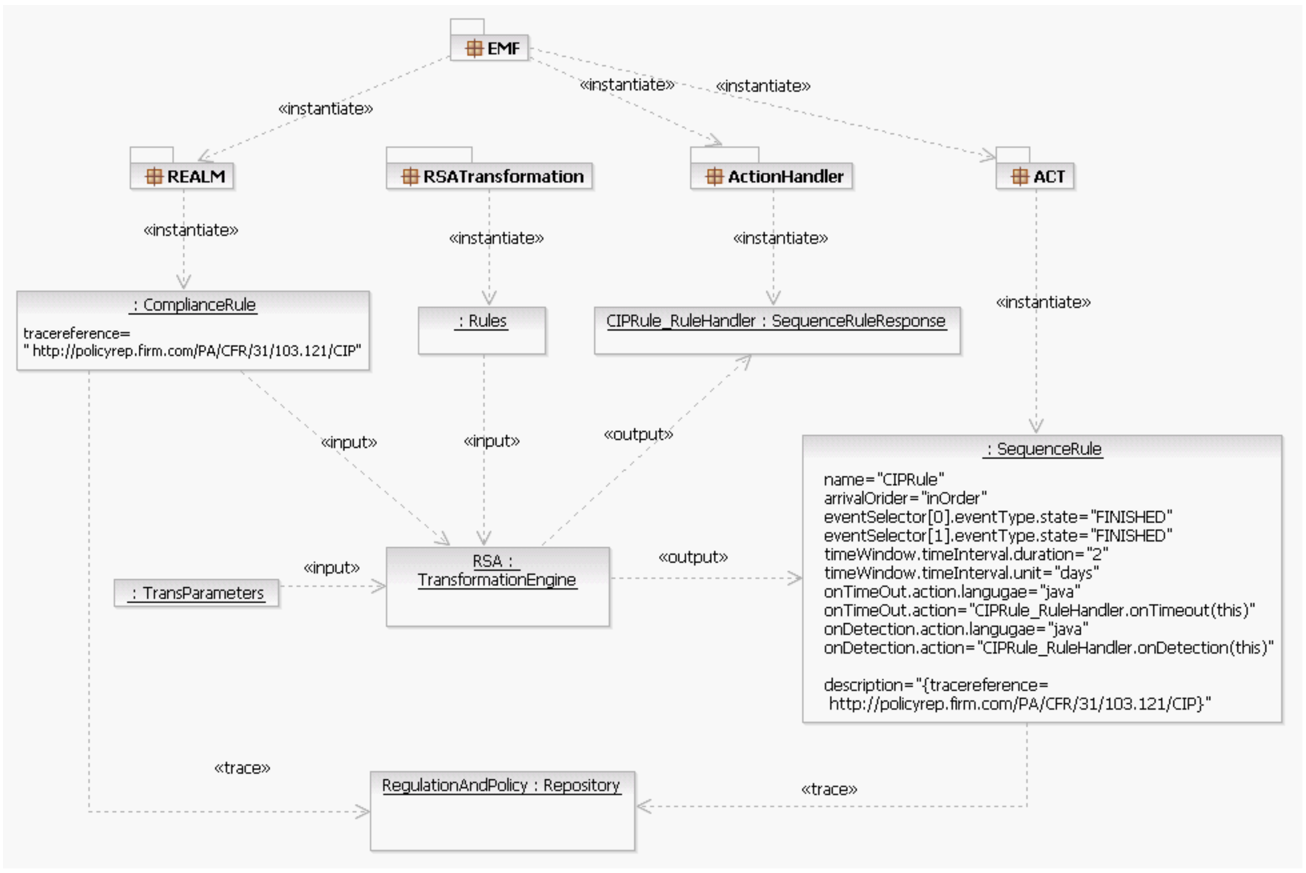
EMF
«instantiate»
«instantiate»
«instantiate»
REALM
RSATransformation
ActionHandler
ACT
«instantiate»
«instantiate»
«instantiate»
«instantiate»
: ComplianceRule
tracereference=
" http://policyrep.firm.com/PA/CFR/31/103.121/CIP"
: Rules
CIPRule_RuleHandler : SequenceRuleResponse
«input»
«input»
«output»
: SequenceRule
name="CIPRule"
arrivalOrder="inOrder"
eventSelector[0].eventType.state="FINISHED"
eventSelector[1].eventType.state="FINISHED"
timeWindow.timeInterval.duration="2"
timeWindow.timeInterval.unit="days"
onTimeOut.action.langugae="java"
onTimeOut.action="CIPRule_RuleHandler.onTimeout(this)"
onDetection.action.langugae="java"
onDetection.action="CIPRule_RuleHandler.onDetection(this)"
description="{tracereference=
http://policyrep.firm.com/PA/CFR/31/103.121/CIP}"
: TransParameters
«input»
RSA : TransformationEngine
«output»
«trace»
RegulationAndPolicy : Repository
«trace»

Figure 6: Transformation Architecture.

4.2 Transformation Mappings

Mappings performed in transforming REALM compliance policies into ACT correlation rules are divided into three categories for discussion: standard mappings, action-handler mappings, and pattern-based mapping. All mappings are implemented with the RSA Transformation Framework.

A pre-requisite to the transformation process is the mapping of REALM's abstract objects, defined in its Concept Model, to existing software versions of the same constructs. Using the example of the U.S. Patriot Act earlier, references to Customer, Record and Account are configured to refer to the corresponding equivalents in an application system such as a REALM Customer being mapped to a given application's Customer class.

The mappings assume the REALM policy has been authored and exists in form of an XMI document, EMF's default XML serialization. As both the REALM policy and ACT rule have XML forms, XPath is used as a notation below for specifying model elements. The transformations, however, operate on the EMF object instances, not XML document elements.

Standard mappings are straightforward copying of element values from source-to-target and are performed for each transformation. Standard mappings are summarized in Table 1.

Source	*Target*	*Comments*
/realm:ComplianceRule	/act:ruleSet/ruleBlock	Root REALM element creates of ACT Rule-Set and child RuleBlock element.
/realm:ComplianceRule/@name	/act:ruleSet/ruleBlock/@name	Compliance rule name mapped to ACT Rule-Block name.
/realm:ComplianceRule/tracereference	/act:ruleSet/ruleBlock/comment	Adds policy URI to ACT rule block comment.

Table 1: Standard REALM-to-ACT mappings.

Action-handler mappings address the naming conventions required for the generated handler code. By partially deriving package and class names from REALM elements, class names reflect the name of the policy or law being implemented. The naming conventions used in generating package and class names are outlined in Table 2. Internally, the action handler is created as an EMF model (.ecore file) from which the handler code is generated using EMF's code generation facility.

Source	*Target*	*Comments*
Transformation parameter	base Java package	e.g., com.example
/realm:ComplianceRule/@name	Java package suffix	Append rule name to base package prefix, e.g., com.example.CIPRule
Transformation parameter	Base handler class name with, "RuleHandler", appended	Appended to full package name, e.g., com.example.CIPRule.CIPRule302_RuleHandler

Table 2: Action handler mappings.

Temporal rule patterns serve as parameterizable templates during transformation. Parameterizable objects are resolved through input parameters to the transformation, source model objects and target model objects. Initially, the pattern identifies the target rule type. Table 3 shows how the temporal rule patterns described earlier map to the ACT rule types. Note that given an event, the two patterns, x_Within(c)_Before_y and x_AtLeast(c)_Before_y, need to assert another event having occurred in the past, implying the need to consult a persistent event store. These patterns therefore implement the Computation rule in order to programmatically invoke the event lookup, presumably to a database.

Pattern	*Target*	*Comments*
y_Within(c)_After_x	/act:ruleSet/ruleBlock/sequenceRule	This pattern maps to an ACT Sequence.
x_Within(c)_Before_y	/act:ruleSet/ruleBlock/computationRule	Look-up into past requires a custom implementation.
x_AtLeast(c)_Before_y	/act:ruleSet/ruleBlock/computationRule	Look-up into past requires a custom implementation.

Table 3: Pattern to ACT rule type mappings.

Temporal patterns also determine the specification of events a rule triggers on. The transformation target for event selection is a combination of ACT's event selector field and its optionally contained filter predicate. The event selector itself is used to specify a primary event and the filter predicate is used for additional logic in deciding whether an event will be processed. The primary event selection is derived, as described in Section 2, from the action predicates, *starts* and *finishes*. The state predicate, indicating a relationship between objects, is mapped to the event selector's filter predicate, where fine-grained comparison logic can be applied.

In the case of the patterns, x_Within(c)_Before_y and x_AtLeast(c)_Before_y, the latter event is taken as the event selector. The filter predicate implements the correlation relationship such as whether two record IDs refer to the same object.

Time is specified as a value in a given unit. Assuming time value and units are specified during policy definition, the mapping into ACT elements is straightforward and shown in Table 4.

Source	*Target*	*Comments*
/realm:ComplianceRule//TimeConstant/@units	/act:ruleSet//timeInterval/@unit	Note that REALM time units are specified when the REALM policy is authored. Translation of REALM time unit names to ACT required.
/realm:ComplianceRule//TimeConstant/@value	/act:ruleSet//timeInterval/@duration	

Table 4: Time unit and value mappings.

4.3 Example

This section describes the policy transformation using the excerpt from the U.S. Patriot Act described in Section 2.1. This example assumes the WebSphere Process Server (WPS) executes a business process which includes activities for applying for an account and verifying customer identity. The REALM objects are mapped to business objects in the application system such as CustomerApplication and Account. The transformation is further configured to map REALM actions to process activities such as AccountOpen and IDVerification. The business process is configured to emit CBEs at the beginning and end of these activities.

The output of the transformation is shown in Figure 7 and Figure 8. The generated ACT Sequence rule in Figure 7 contains a URI reference to the regulation the rule applies to. The event selectors contain XPath expressions identifying the start of account open activity and the finishing of the identity verification activity, corresponding to the respective REALM action predicates. The time window sets a timeout of 2 days. If the event sequence is detected within the allowable time window, the action handler method, *onDetection()*, is invoked. Otherwise the action handler method, *onTimeout()*, will be called. The action handler associated with this rule is shown in Figure 8. The method handling the timeout emits another CBE indicating a compliance violation event while the method for detection of the compliant sequence sends an audit event.

```
<ruleBlock name="CIP_Period">
 <comment>{http://policyrep.firm.com/PA/CFR/31/103.121/CIP}</comment>

 <sequenceRule name="CIPRule">

  <eventSelector alias="AccountOpenActivity">
    <!-- CBE with extensionName "BPE" indicate process event -->
    <eventType type="/CommonBaseEvent/@extensionName="BPE"[activty="AccountOpen" and state="STARTED]"/>
  </eventSelector>

  <eventSelector alias="IdentityVerificationActivity">
    <eventType type="/CommonBaseEvent/@extensionName="BPE"[activty="IDVerification" and state="FINISHED]"/>
  </eventSelector>

   <timeWindow>
     <timeInterval duration="2" unit="days"/>
   </timeWindow>

  <onDetection>
    <action expressionLanguage="java" name="onDetection_">
        CIPRule_RuleHandler.onDetection(this);
    </action>
  </onDetection>

  <onTimeOut>
     <action expressionLanguage="java" name="onTimeout_">
       CIPRule_RuleHandler.onTimeout(this);
     </action>
  </onTimeOut>

 </sequenceRule>
</ruleBlock>
```

Figure 7: ACT rule file.

```
/*
 * If events occur within the required time,
 * the process is within compliance limits.
 * Send message to audit trail.
 */
public void onDetection(Object context) {
    sendToAudit(createAuditEvent(context);
}

/*
 * If a timeout occurs, a compliance violation has occurred.
 * Create a corresponding CBE and forward to appropriate queue.
 */
public void onTimeout(Object context) {

    // create compliance CBE:
    EventFactory ef = EventUtils.getCachedEventFactory("TestFactory");
    CommonBaseEvent event = ef.createCommonBaseEvent(EventUtils.COMPLIANCE_EXT_NAME);

    ComponentIdentification srcComp = createSrcComponent();
    event.setSourceComponentId(srcComp);
    Situation sit = ef.createSituation();
    sit.setCategoryName(Situation.REPORT_SITUATION_CATEGORY);
    event.setSituation(sit);

    // optional:
    event.setReporterComponentId(srcComp);
    event.setMsg("Compliance violation");
    event.setPriority(CommonBaseEvent.PRIORITY_HIGH);
    event.setSeverity(CommonBaseEvent.SEVERITY_CRITICAL);

    // send
    sendToViolationsQueue(event);
}
```

Figure 8: Java snippet from action handler, CIPRule_RuleHandler.

5 Conclusion

In this paper, we have shown how regulatory requirements, which are captured in high-level policies with a compliance metamodel, can be automatically transformed into low-level event correlation rules with the aid of temporal rule patterns. The REALM compliance metamodel supports the expression of temporal logic modalities, thereby addressing a common theme in compliance, namely temporal ordering and time periods.

We have presented three temporal rule patterns which occur in regulations. The rule patterns are used in identifying those subsets of regulations which can be sensibly monitored using event correlation. The patterns, defining a specific temporal structure over parameterizable objects, also guide the transformation of policy expressions into correlation rules. The parameterizable objects can be resolved to input parameters and elements in the source or target models. The pattern defines the target rule type, event selection and time windows generated during transformation.

Two key aspects of the automating compliance-related activities have been addressed. First, model transformation automates the generation of runtime artifacts from high-level policies. As regulations, their interpretations, and technology all change over time, this automation path impacts the development and deployment of policy-based systems. The second form of automation presented points out the role event monitoring and correlation can play in automating compliance control activities and assessment. Key enablers of automating compliance monitoring are event standards, such as the Common Base Event, and event-oriented middleware capabilities such as the Common Event Infrastructure. This common infrastructure for reliably storing and distributing events places enterprises in a better position to integrate applications into to an overall compliance monitoring architecture.

Acknowledgements

We thank Thomas Gschwind and Michael Moser for their valuable comments and contributions on earlier versions of this paper. We are also grateful to Alice Y. Liu, Luke O'Connor, and Naishin Seki for discussions pertaining to temporal rule patterns and their transformations, and to Andreas Wespi for general support.

References

[1] R. Agrawal, C. Johnson, J. Kiernan, and F. Leymann. Taming compliance with sarbanes-oxley internal controls using database technology. In *ICDE*, page 92, 2006.

[2] R. Alur and T. A. Henzinger. Real-time logics: Complexity and expressiveness. *Information and Computation*, 104/1:35–77, 1993.

[3] R. Alur and T. A. Henzinger. A really temporal logic. *Journal of the ACM*, 41/1:181–204, 1994.

[4] D. Basin, M. Hilty, and A. Pretschner. Distributed usage control. *Communications of the ACM*, 2006. to appear.

[5] T. Berners-Lee, R. Fielding, and L. Masinter. RFC 2396 - uniform resource identifiers (URI): Generic syntax, August 1998. Internet Request for Comments.

[6] A. Biazetti and K. Gajda. Achieving complex event processing with active correlation technology, 2005. IBM developerworks, `http://www-128.ibm.com/developerworks/autonomic/library/ac-acact/`.

[7] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. J. Grose. *Eclipse Modeling Framework*. Addison-Wesley Professional, 2003.

[8] David Ogle et al. Canonical situation data format: The common base event v1.0.1, November 2003. Copyright IBM, `http://www.eclipse.org/tptp/platform/documents/resources/cbe101spec/CommonBaseEvent_SituationData_V1.0.1.pdf`.

[9] C. Giblin, A. Y. Liu, S. Müller, B. Pfitzmann, and X. Zhou. Regulations Expressed As Logical Models (REALM). In M.-F. Moens and P. Spyns, editors, *Proceedings of the 18th Annual Conference on Legal Knowledge and Information Systems (JURIX 2005)*, volume 134 of *Frontiers in Artificial Intelligence and Applications*, pages 37–48. IOS Press, December 2005.

[10] P. Swithinbank, M. Chessell, T. Gardner, C. Griffin, J. Man, H. Wylie, and L. Yusuf. *Patterns: Model-Driven Development Using IBM Rational Software Architect*. IBM Redbook, 2005.

Biographies

Christopher Giblin, IBM Research Division, IBM Zurich Research Laboratory, Säumerstrasse 4, 8803 Rüschlikon, Switzerland (cgi@zurich.ibm.com). Christopher Giblin is a Software Engineer at the IBM Zurich Research Lab. His interests are middleware, security engineering and model transformation.

Samuel Müller, IBM Research Division, IBM Zurich Research Laboratory, Säumerstrasse 4, 8803 Rüschlikon, Switzerland (sml@zurich.ibm.com). Mr. Müller obtained a M.S. in Computer Science from the University of Zurich in 2004 and a M.A. in Economics from the University of Zurich in 2006. He joined IBM Research in Zurich in 2004, where he is currently doing research in the area of risk and compliance. In parallel, he is working towards his doctorate as an external Ph.D. student at the Swiss Federal Institute of Technology (ETH) Zurich, where he is a member of the Information Security group. His thesis advisors are Prof. Dr. David Basin and Prof. Dr. Birgit Pfitzmann and his research interests include modal logics, formal methods, modeling methodology, risk and compliance management, game theory and economics.

Birgit Pfitzmann, IBM Research Division, IBM Zurich Research Laboratory, Säumerstrasse 4, 8803 Rüschlikon, Switzerland (bpf@zurich.ibm.com). Birgit Pfitzmann is a senior research staff member at the IBM Zurich Research Lab. She joined IBM in 2001. Since then she was responsible for research in risk and compliance, identity management, web services security, key management, and formal verification of cryptographic protocols. She has served on various taskforces on defining IBM's technology and strategy in security and privacy. Before joining IBM, she was a tenured professor and dean of the Department for Computer Science at the University of Saarland in Saarbrcken. Birgit Pfitzmann is author of more than 100 research papers in security, privacy and cryptography, and regularly serves on the program committees and as program chair of international conferences on these topics. She received a Diploma in Computer Science from the University of Karlsruhe, and a Doctorate from the University of Hildesheim, both in Germany. She is Member of the IACR, where she served on the Board of Directors, and of the ACM and GI, and Senior Member of the IEEE.