A UML Extension for the Model-driven Specification of Audit Rules

Bernhard Hoisl[1,2] and Mark Strembeck[1,2]

[1] Institute for Information Systems and New Media,
Vienna University of Economics and Business (WU Vienna),
Augasse 2-6, 1090 Vienna, Austria
[2] Secure Business Austria Research (SBA Research),
Favoritenstrasse 16, 1040 Vienna, Austria
{bernhard.hoisl,mark.strembeck}@wu.ac.at

Abstract. In recent years, a number of laws and regulations (such as the Basel II accord or SOX) demand that organizations record certain activities or decisions to fulfill legally enforced reporting duties. Most of these regulations have a direct impact on the information systems that support an organization's business processes. Therefore, the definition of audit requirements at the modeling-level is an important prerequisite for the thorough implementation and enforcement of corresponding policies in a software system. In this paper, we present a UML extension for the specification of audit properties. The extension is generic and can be applied to a wide variety of UML elements. In a model-driven development (MDD) approach, our extension can be used to generate corresponding audit rules via model transformations.

Key words: Audit, Model-driven Development, UML

1 Introduction

In information system security, an audit process records and analyzes data about the activities in a software system in order to detect security violations or to identify the cause of such violations (see, e.g., [1, 2, 3, 4, 5]). In this paper, we use the term *audit* for an "independent review and examination of records and activities to assess the adequacy of system controls and ensure compliance with established policies and operational procedures" [6]. Audit requirements not only stem from organization-specific management decisions or cost controlling policies, but also from corresponding laws and regulations, such as the Basel II Accord or the Sarbanes-Oxly Act (SOX) (see [7, 8]).

An audit process may involve different departments or divisions and focus on different assets of an organization, for example, financial records, customer privacy regulations, or access control policies. Nevertheless, all audit processes have in common that they are more and more based on and supported through information systems. For this reason, the software systems of an organization must be able to keep an audit trail of all audit-relevant business processes and activities. However, process modeling languages such as BPMN [9] or UML activity diagrams [10] do not provide native language elements to model such security properties. Thus, in order to properly enforce business-level security concerns in the corresponding software systems we need to integrate these concepts in a modeling language.

In recent years, model-driven development (MDD; see, e.g., [11, 12]) emerged as an approach for the specification of tailored domain-specific software systems. Due to its versatility, MDD can be applied as an approach for the systematic specification of information system security properties (see, e.g., [13, 14, 15, 16]). In the context of MDD, domain-specific languages (DSLs) are tailor-made (computer) languages for a specific problem domain (see, e.g., [17, 18, 19]). In general, a DSL can be defined as a standalone language or as a domain-specific extension to a pre-existing (modeling or programming) language. Such domain-specific extensions are also called "embedded DSLs".

In this paper, we present an approach for modeling system audits. In particular, we present a domain-specific UML extension that provides new language elements for the specification of audit events, audit rules, and notifications (or actions) that are triggered via audit events. The remainder of this paper is structured as follows: in Section 2 we give an overview of our audit modeling approach. Section 3 describes the metamodel, syntax, and semantics of our UML extension. Subsequently, Section 4 gives an example how our extension can be used to describe different audit modeling perspectives. After that, Section 5 summarizes related work and Section 6 concludes the paper. In addition, Appendix A provides a textual concrete syntax for our UML extension.

2 Motivation and Approach Synopsis

For each organization, a number of laws, regulations, and internal rules demand that the organization records certain activities or decisions which have a direct impact on the corresponding information systems (see, e.g., [20, 21, 22]). In particular, audit trails are needed to discharge an organization's reporting duties, for example, to prove the correctness of certain financial transactions (such as the enforcement of the four-eyes-principle for procurement operations). However, software engineers are usually not aware of all legal requirements that must be fulfilled by a software system. Therefore, we need a means to incorporate audit requirements in the respective software models. On the one hand, such a means should support the software engineer to model corresponding audit properties in a standard modeling language (such as the UML), on the other hand it should facilitate the communication between software engineers and domain experts (such as lawyers or experts from a certain business domain).

Moreover, because software systems as well as laws and regulations change over time, an extension for audit modeling should support the integration of audit properties with many different types of (heterogeneous) systems. Synchronous request/reply communication typically results in a strong coupling of interacting components. In contrast to that, a loose coupling of software services helps to integrate many different types of heterogeneous (legacy) systems (see, e.g., [23]).

Event-based communication is an important paradigm to model and implement such loosely-coupled systems—it is asynchronous and inherently decouples interacting system components (see, e.g., [24]). Event-based communication follows a publish/subscribe scheme where software components can produce and consume events. This means, an event producer does neither know the consumers of its events, nor does the producer publish events with the intention to trigger some action in an other component. Therefore, event-based components only have to know how to react on a particular notification and then publish events to "whom it may concern". This allows for a straightforward integration of new components and, thus, directly supports the evolution of event-based systems. Moreover, because event producers and event consumers are almost completely decoupled, event-based components are widely independent of each other which, again, makes these components more easy to adapt and extend.

In this paper, we, therefore, present an approach for the event-based modeling of audit properties. Fig. 1 shows an informal overview for the main conceptual elements of our approach. In essence, we provide a UML extension to model audit properties of software artifacts that can be applied to different types of UML models. We have chosen the UML because it is the de-facto standard for modeling information systems and provides native support for all types of software models as well as for event-based modeling. The audit properties defined via this modeling extension can then be used to generate corresponding audit rules that can be enforced in a software system.

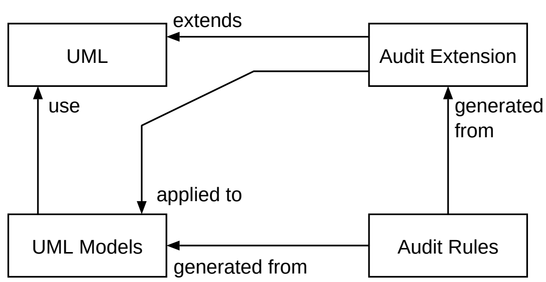


Fig. 1. Audit extension for UML models

Our extension supports the definition of different perspectives, each of which models a particular aspect of system audits (see Fig. 2). Subsequently, model transformations (see, e.g., [25, 26, 27]) can be used to generate different types of software artifacts and audit rules from these models. The generated artifacts then enforce the behavior that was defined on the modeling level. Thereby, our UML extension allows to map audit requirements from the modeling- to the system-level. Because the UML provides an integrated family of modeling notations, a UML extension helps to avoid the semantic gap that could occur if we integrate models that are defined in different modeling languages (see, e.g., [28, 29]).

Our extension is generic and allows to define audit requirements for arbitrary elements in arbitrary UML models. Moreover, it is event-based and thereby

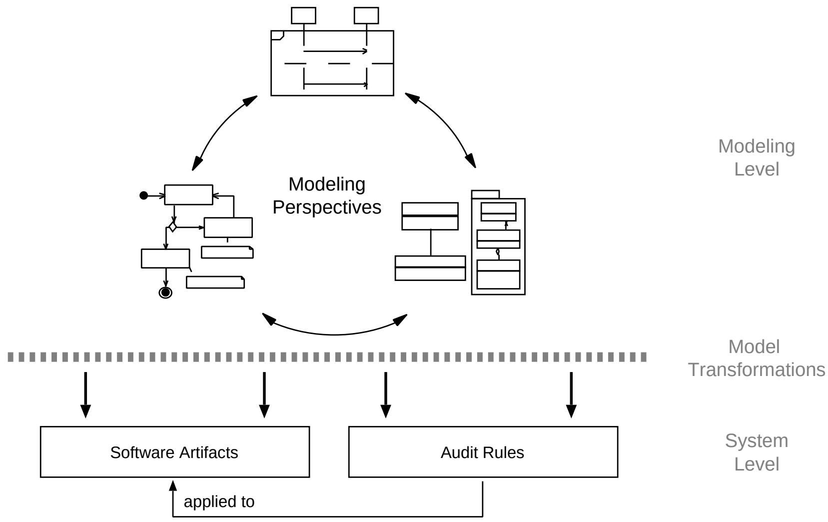


Fig. 2. Modeling-level audit properties are transformed into system artifacts

enables a loose-coupling and a straightforward integration with different types of (heterogeneous) software components.

3 UML Audit Extension

3.1 Metamodel Overview

In this section, we specify a UML extension (see Fig. 3) for modeling event-based audit requirements. In particular, we introduce a new package called *SecurityAudit* as a UML metamodel extension [10]. The package consists of both, a UML stereotype specialization and MOF-based (Meta Object Facility, [30]) extensions.

In general, the UML can be extended in two ways: (1) by using UML profiles [10] or (2) by introducing new modeling concepts on the metamodel level. UML profiles provide a mechanism for the extension of existing UML metaclasses to adapt them for non-standard purposes. However, UML profiles are not a first-class extension mechanism (see [10, page 660]). They extend existing metaclasses of the UML metamodel and the extension defined through a profile must be consistent with the semantics of the extended (original) UML metaclasses. For this reason, more complex extensions are defined on the level of the UML metamodel (see [10, 30]). An extension of the UML metamodel allows to define new and specifically tailored UML elements (defined via new metaclasses), and allows to define a customized notation, syntax, and semantics for the new modeling elements. In our extension, we employ a combination of both methods to take advantage of each mechanism.

In our extension, the `«stereotype» AuditEventSource` extends the UML `Element` metaclass (see Fig. 3). As a specialized `Element` stereotype, it is possible

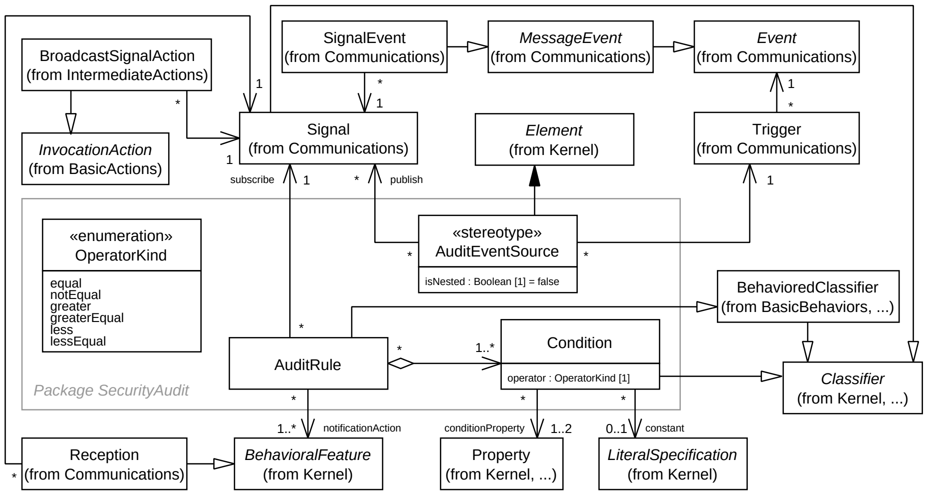


Fig. 3. UML extension for modeling event-based audit requirements

to define any UML element as being the source for an event that may trigger an audit-related behavior execution. In this way, an integration with arbitrary (pre-existing) UML models is possible. The `isNested` attribute defines whether the `AuditEventSource` stereotype is applied to the owned elements of a stereotyped element (e.g. to all nodes in an UML activity). Hence, it is possible to tag the owner element only and recursively apply the `AuditEventSource` stereotype and its properties to all nested elements.

A `Trigger` relates an `Event` to a certain type of `Signal` that is published each time this particular event occurs. A UML `Signal` is a specialized `Classifier` and can carry data which is passed via the corresponding send invocation occurrence. Events are published through a corresponding `BroadcastSignalAction` which transmits a `Signal` instance to all potential target objects in a system (see also Fig. 3 and [10]). We use a `BroadcastSignalAction` in favor of a `SendSignalAction` because events are published independent of the entities (software components) consuming the events (see, e.g., [24]).

Modeling the receipt of a `Signal` instance is done via an `AcceptEventAction` (in behavior diagrams) or via the `Reception` element (in structure diagrams). Either way, a `SignalEvent` represents the receiving of an asynchronous `Signal` instance. The elements modeling the transmission and receipt of `Signal` instances act as the underlying event notification service, which mediates between notification producers and consumers (according to the publish/subscribe pattern; see, e.g., [24, 31]).

An `AuditRule` is defined as a specialized `BehavioredClassifier` and is subscribed to a specific `Signal` (see Fig. 3). Each `AuditRule` consists of one or more `Condition` elements. A `Condition` evaluates a certain attribute of a `Signal` and checks the corresponding attribute value (e.g. by using binary infix operators, as in: “price $<$ 63.50” or “currency $=$ EUR”). In our extension, a

`Condition` can test either two `Properties` against each other, or it can check a `Property` against a pre-defined constant value (a `LiteralSpecification`). A UML `LiteralSpecification` references an instance of a primitive data type[1]. For basic condition matching, the `«enumeration» OperatorKind` specifies an exemplary list of valid self-explanatory operator alternatives. Note, however, that these infix comparison operators can easily be extended to represent other types of operators, for instance, n-ary prefix operators (such as `isInAscendingOrder(...)`, `isInDescendingOrder(...)`, or `includes(...)`).

An `AuditRule` matches an event (resp. the corresponding `Signal`) if all `Conditions` that are associated with this `AuditRule` are fulfilled. In case all `Conditions` of an `AuditRule` are fulfilled, the respective `AuditRule` triggers the execution of a certain `BehavioralFeature` (see Fig. 3). This `BehavioralFeature` implements a notification action that informs another system entity that one of the audit rules was activated and causes a certain behavior (e.g., generating a new log entry in the audit trail).

In general, every stereotype must be included (directly or indirectly) in a profile [10]. For our extension, we define that the `«stereotype» AuditEventSource` is contained in the `AuditEventSourceProfile`. We use the Object Constraint Language (OCL, [32]) to formally specify constraints for our modeling extension:

```
context AuditEventSource inv:
  self.profile.name = 'AuditEventSourceProfile'
```

As this profile is an integral part of our extension, we define that it must be applied to the package `SecurityAudit`:

```
context SecurityAudit inv:
  self.profileApplication->exists(
    appliedProfile.name = 'AuditEventSourceProfile')
```

The relationship of the `SecurityAudit` package, its profile application, and their referenced metamodels are shown in Fig. 4. The profile `AuditEventSource-Profile` references the UML metamodel and is applied to the package `Security-Audit`. As we define the package `SecurityAudit` via a UML metamodel extension, it uses elements from the UML. The MOF is self-describing (through reflection; see [30]) and, therefore, does not need another metamodel for its specification. Furthermore, the MOF specification reuses modeling constructs from the UML infrastructure library (through package imports; see [33]).

3.2 Metamodel Elements' Syntax and Semantics

Table 1 shows the notation elements of the `SecurityAudit` package (see also Section 4). The other UML elements used in our examples correspond to the UML specification (see [10]).

[1] The UML defines six `LiteralSpecification` subtypes: `LiteralNull`, `LiteralBoolean`, `LiteralInteger`, `LiteralReal`, `LiteralString`, and `LiteralUnlimitedNatural` [10]. Due to space limitations these six specializing `LiteralSpecifications` are omitted in Fig. 3.

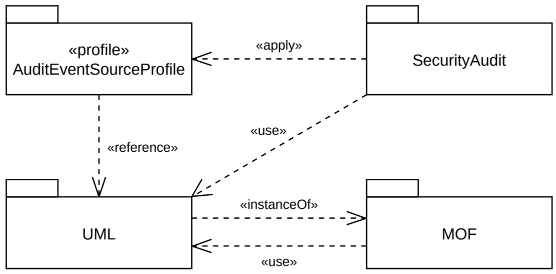


Fig. 4. Dependencies of the `SecurityAudit` package

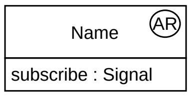
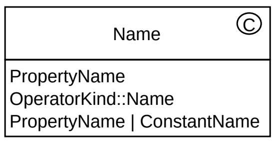
Node type	Notation	Explanation
AuditRule		An `AuditRule` is shown as a rectangle with the encircled characters AR in the upper right corner. The lower compartment states that the `AuditRule` is subscribed to the receipt of a certain `Signal` type (see [10]).
Condition		A `Condition` is shown as a rectangle with the encircled character C in the upper right corner. The lower compartment includes the attributes and the operator that constitute the respective condition. The first attribute is the name of a `Property` which references a certain `Signal` attribute, the second attribute may either be another `Property` or a constant value (i.e. a `LiteralSpecification`), and the operator is of type `OperatorKind` (see Fig. 3). Thereby, a condition consists of an operator that compares two operands (for example "price $<$ 63.50" or "currency $=$ EUR").

Table 1. Modeling elements of the `SecurityAudit` package

In addition to the graphical modeling elements, Appendix A provides a textual syntax for event-based audits that is specified via a variant of the Backus-Naur-Form (BNF). We have chosen this BNF variant as a notation because it is also applied in OMG specifications (e.g., [10, 32]), it is commonly used to formally specify the syntax of computer languages, and it is widely tool-supported (e.g., the Eclipse Xtext notation is very similar to an extended BNF). To model event-based audits, the graphical or the textual syntax can be used separately and equivalently. Moreover, it is also possible to combine the textual and graphical syntaxes (see the example in Section 4).

In addition, to the syntax definitions we specify OCL invariants that ensure the correct semantics of models defined with our UML extension (see Fig. 3). The

AuditEventSource stereotype can be applied recursively to all owned elements of a tagged element (if the isNested attribute is set to true). All stereotype properties of the tagged owner element are inherited, except if a nested element explicitly defines its own Trigger and Signal. In this case, the properties of the tagged owner element are overwritten[2]:

```
context AuditEventSource inv:
  self.isNested implies
    self.base_Element.ownedElement->forAll(oe |
      oe.getAppliedStereotype('AuditEventSourceProfile::
          AuditEventSource') <> null)
```

To be able to evaluate a Condition of an AuditRule, exactly one Property must be a referenced attribute of the subscribed Signal instance:

```
context AuditRule inv:
  self.condition->forAll(c |
    self.subscribe.ownedAttribute->intersection(
      c.conditionProperty)->size() =
    c.ownedAttribute->select(oa |
      oa.name = 'conditionProperty')->first().lowerBound())
```

We define that a Condition can test either two Properties against each other or one Property against a constant (as specified in the metamodel), but not both. Specifying a Condition without matching operands is also not allowed:

```
context Condition inv:
  self.conditionProperty->size() + self.constant->size() =
  self.ownedAttribute->select(oa |
    oa.name = 'conditionProperty')->first().upperBound().
        oclAsType(Integer)
```

Matching Properties against each other or against a LiteralSpecification constant implies that they conform to the same type (e.g., both are of type <Primitive Type> Integer):

```
context Condition inv:
  if self.constant->notEmpty() then
    self.conditionProperty->forAll(p |
      p.type.conformsTo(self.constant.type))
  else
    self.conditionProperty->forAll(p1,p2 |
      p1.type.conformsTo(p2.type))
  endif
```

4 Audit Modeling Perspectives

In this section, we describe an example for audit modeling of a simple event-based system. In order to thoroughly describe a software system, different modeling

[2] Please note that the getAppliedStereotype method is provided by the Eclipse MDT/OCL engine.

perspectives have to be defined. Therefore, we take different viewpoints into account to explain the application of our UML extension to different structural and behavioral models. The perspectives in Fig. 5 are exemplary and can be used interchangeable.

Fig. 5a shows a process-based perspective modeled via a UML activity diagram. Here, the `«AuditEventSource»` stereotype is applied to two `BroadcastSignalActions`. The example models a basic login process to an ERP system that should include audit trails for successful as well as for failed login attempts (indicated via the `«AuditEventSource»` stereotype). Two notes are attached to the actions including the `Trigger` for the audit event and the corresponding `Signal` classifier. However, using this perspective alone, information about the `Signals`, the `AuditRules`, their `Conditions` and `Actions` can not be modeled sufficiently.

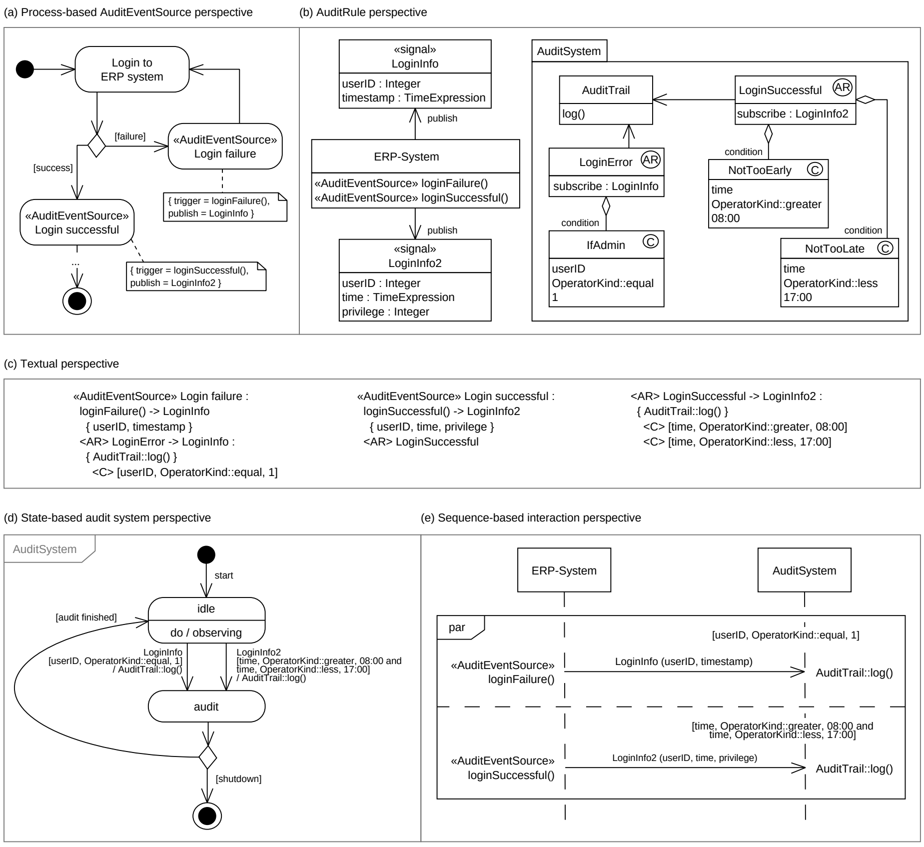


Fig. 5. Modeling event-based audit requirements from different perspectives

Therefore, Fig. 5b presents the `AuditRule` perspective. It shows an `ERP-System` classifier that implements two methods which match the execution operations of the corresponding `BroadcastSignalActions` shown in Fig. 5a. The `«AuditEventSource»` stereotypes bind both, the `«signal» LoginInfo` to the `loginFailure()` method and the `«signal» LoginInfo2` to the `loginSuccessful()` method. Furthermore, Fig. 5b shows two simple `AuditRules LoginError` and `LoginSuccessful` with each having a compartment defining the corresponding subscribed `Signal`. The `AuditRule LoginError` consists of one `Condition` (`IfAdmin`) which checks for failed administrator logins (i.e., if the `userID` included in the corresponding `Signal` instance equals `1`[3]). The second `AuditRule LoginSuccessful` consists of two conditions which check if a login happened outside of normal business hours. If these `Conditions` evaluate to true, the `log()` method of the `AuditTrail` classifier is invoked (as both `AuditRules` reference the same notification action). This perspective, of course, omits all process information.

Fig. 5c shows an example of the textual perspective. The syntax conforms to the grammar defined in Appendix A. The textual syntax is equivalent to the graphical `AuditRule` perspective (see Fig. 5b); i.e. all `AuditRules` and `Conditions` are equally defined. The textual syntax can be used complementary to the graphical representation.

Fig. 5d shows a perspective of the audit system as a UML state machine. The state machine is used to model the receiving `Signal` instances, their `Conditions`, and corresponding actions. As can be seen from the `AuditRule` and the textual perspective, the second `Signal` named `LoginInfo2` serves as the notification message of action `Login successful` in the process-based view. The state machine, for instance, shows the same `Signal`, `Condition`, and action information associated with the corresponding transition. In this perspective, the modeled states and their transitions of an audit system reveal neither process- nor object-specific information.

Finally, Fig. 5e shows a simplified message interaction perspective as a UML sequence diagram. Therein, the sending and receiving events of the two involved systems, together with the interchanged signal messages are shown. Both `«AuditEventSource»` events are defined for parallel execution, i.e. there is no sequential order between these events. The corresponding messages are defined via the respective `Signal` names including their owned attributes. The `Conditions` for invoking audit actions are defined as guards on the lifeline of the `AuditSystem`. This perspective neither shows the process flow nor the detailed structure of the audit rules.

All perspectives presented here are complementary and can be used interchangeable. The combination of perspectives are dependent on the modeled software system (e.g., state-based).

5 Related Work

In [34], Jürjens describes how to model audit security for smart-card payment schemes with UMLSec. The UMLSec extension is defined as a UML profile. Our

[3] For the sake of simplicity, we assume that the administrator of the ERP system has the value `1` for the attribute `userID`.

extension for audit modeling supports the definition of different audit perspectives and complements the UMLSec approach. In general, we extend the UML `Element` metaclass and, thereby, allow to extend a wide variety of UML elements with audit properties. Furthermore, our extension supports event-based modeling and, thus, aims to facilitate the integration of audit properties into pre-existing models for heterogeneous (or legacy) systems.

Rodríguez et al. [35] present a UML profile extension for activity diagrams which aims to support the specification of certain security properties (e.g., access control, integrity, non-repudiation, and privacy). In [35], audits are specified as an additional characteristic for another security property. The audit process is treated as a logging of data, and the logged data must be defined via attributes of the corresponding audited entity. In contrast, our extension is more generic and can be used to model audit rules for arbitrary UML elements. Moreover, our audit extension is integrated with other UML extensions for security modeling (see, e.g., [15, 36, 37, 38, 39])

In [40], Fernández-Medina et al. provide support for modeling access control and audit properties for multidimensional data warehouses with a UML profile definition. Audit requirements are considered by defining audit rules for logging user requests and activities. Audit rules are defined via a custom-made grammar specified in Extended Backus-Naur-Form (EBNF). These audit rules are represented in the form of constraints for a UML class diagram. In contrast, our approach is not specific to a particular application domain and can be integrated with other UML-based approaches.

In [41], an approach for the modeling of security-critical, service-oriented systems is presented. The authors provide a UML profile that defines stereotypes for the extension of class diagrams. Security patterns and protocols are applied to identified security critical use cases. Service composition rules can be defined as post-obligations to be taken into account while (or after) executing a protocol (e.g., auditing). In [41], audit requirements are not defined as specialized modeling elements, but via OCL constraints. Thus, the modeling approach is rather specialized and has a limited expressiveness (for both, syntax and semantics).

6 Conclusion

In this paper we presented a UML extension for modeling system audits. Our extension supports an event-based modeling style and thereby aims to enable the integration of audit properties in a wide variety of different types of UML models. We support the definition of structural and behavioral perspectives to model different aspects of system audits. In addition to graphical model elements, we also provide a fully equivalent textual syntax.

With our extension, each UML element can be defined as an audit event source. Thus, the extension is not limited to a specific type of UML diagram. Moreover, it can be customized to different types of system audits. However, in this paper we do not elaborate on the modeling of an event notification service (i.e., we omit an explicit discussion of `BroadcastSignalActions` and

`AcceptEventActions` in our examples). Furthermore, we neither show an example of nested audit models nor discuss wildcard triggers which invoke a specified audit rule on every event occurrence of an element or nested elements. Application-specific OCL constraints can be used to further refine, for instance, event triggers or audit rules (e.g., pre- and postconditions). The textual syntax of our extension is fully integrated with the graphical perspectives and can be applied either interchangeable or in addition to the graphical models.

In our future work, we will integrate support for the explicit modeling of composite as well as hierarchical audit event types. Moreover, we are working on a tool integration of our extension which will implement both, the graphical and textual syntax.

Acknowledgments

This work has partly been funded by the Austrian Research Promotion Agency (FFG) of the Austrian Federal Ministry for Transport, Innovation and Technology (BMVIT) through the Competence Centers for Excellent Technologies (COMET K1) initiative and the FIT-IT program.

References

1. Garera, S., Rubin, A.: An Independent Audit Framework for Software Dependent Voting Systems. In: Proc. of the 14th ACM Conference on Computer and Communications Security (CCS). (2007) 256–265
2. Hasan, R., Winslett, M.: Efficient Audit-based Compliance for Relational Data Retention. In: Proc. of the 6th ACM Symposium on Information, Computer and Communications Security. (2011) 238–248
3. King, J., Smith, B., Williams, L.: Modifying Without a Trace: General Audit Guidelines are Inadequate for Open-source Electronic Health Record Audit Mechanisms. In: Proc. of the 2nd ACM SIGHIT International Health Informatics Symposium. (2012) 305–314
4. Sandhu, R., Samarati, P.: Authentication, Access Control, and Audit. ACM Computing Surveys **28**(1) (March 1996) 241–243
5. Schneier, B., Kelsey, J.: Secure Audit Logs to Support Computer Forensics. ACM Transaction on Information and System Security **2**(2) (May 1999) 159–176
6. Committee on National Security Systems: National Information Assurance (IA) – Glossary. Available at: http://www.cnss.gov/Assets/pdf/cnssi_4009.pdf (2010)
7. Basel Committee on Banking Supervision: Basel II: International Convergence of Capital Measurement and Capital Standards. Available at: http://www.bis.org/publ/bcbs107.pdf (2004)
8. United States Congress: Sarbanes-Oxley Act of 2002. Available at: http://www.sec.gov/about/laws/soa2002.pdf (2002)
9. Object Management Group: Business Process Model and Notation (BPMN) – Version 2.0. Available at: http://www.omg.org/spec/BPMN/2.0/PDF (2011)
10. Object Management Group: OMG Unified Modeling Language (OMG UML), Superstructure – Version 2.4.1. Available at: http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF/ (2011)

11. Selic, B.: The Pragmatics of Model-driven Development. IEEE Software **20**(5) (September 2003) 19–25
12. Stahl, T., Völter, M.: Model-Driven Software Development. John Wiley & Sons (2006)
13. Basin, D., Doser, J., Lodderstedt, T.: Model Driven Security: From UML Models to Access Control Infrastructures. ACM Transactions on Software Engineering and Methodology (TOSEM) **15**(1) (January 2006)
14. Hoisl, B., Sobernig, S.: Integrity and Confidentiality Annotations for Service Interfaces in SoaML Models. In: Proceedings of the International Workshop on Security Aspects of Process-aware Information Systems (SAPAIS), Vienna, IEEE (2011)
15. Strembeck, M., Mendling, J.: Modeling Process-related RBAC Models with Extended UML Activity Models. Information and Software Technology (IST) **53**(5) (2010) 456–483
16. Wolter, C., Menzel, M., Schaad, A., Miseldine, P., Meinel, C.: Model-driven business process security requirement specification. Journal of Systems Architecture **55**(4) (April 2009)
17. Deursen, A.V., Klint, P.: Little Languages: little Maintenance? Journal of Software Maintenance: Research and Practice **10**(2) (March 1998) 75–92
18. Mernik, M., Heering, J., Sloane, A.: When and How to Develop Domain-specific Languages. ACM Computing Surveys (CSUR) **37**(4) (December 2005) 316–344
19. Strembeck, M., Zdun, U.: An Approach for the Systematic Development of Domain-Specific Languages. Software: Practice and Experience (SP&E) **39**(15) (October 2009)
20. Cannon, J.C., Byers, M.: Compliance Deconstructed. ACM Queue **4**(7) (September 2006)
21. Damianides, M.: How does SOX change IT? Journal of Corporate Accounting & Finance **15**(6) (2004)
22. Mishra, S., Weistroffer, H.R.: A Framework for Integrating Sarbanes-Oxley Compliance into the Systems Development Process. Communications of the Association for Information Systems (CAIS) **20**(1) (2007)
23. Hohpe, G., Woolf, B.: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley, Boston, Massachusetts, USA (2004)
24. Mühl, G., Fiege, L., Pietzuch, P.: Distributed Event-Based Systems. Springer, Berlin Heidelberg (2006)
25. Mens, T., Gorp, P.V.: A Taxonomy of Model Transformation. Electronic Notes in Theoretical Computer Science **152** (2006) 125–142
26. Sendall, S., Kozaczynski, W.: Model Transformation: The Heart and Soul of Model-Driven Software Development. IEEE Software **20**(5) (2003)
27. Zdun, U., Strembeck, M.: Modeling Composition in Dynamic Programming Environments with Model Transformations. In: Proc. of the 5th International Symposium on Software Composition, Lecture Notes in Computer Science (LNCS), Vol. 4089, Springer-Verlag (March 2006)
28. Axenath, B., Kindler, E., Rubin, V.: AMFIBIA: A Meta-Model for the Integration of Business Process Modelling Aspects. In Leymann, F., Reisig, W., Thatte, S.R., van der Aalst, W., eds.: The Role of Business Processes in Service Oriented Architectures. Number 06291 in Dagstuhl Seminar Proceedings (2006)
29. Zdun, U.: Patterns of Component and Language Integration. In: D. Manolescu, M. Voelter, J. Noble (editors): Pattern Languages of Program Design 5. (2006)
30. Object Management Group: OMG Meta Object Facility (MOF) Core Specification – Version 2.4.1. Available at: http://www.omg.org/spec/MOF/2.4.1/PDF/ (2011)

31. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, Massachusetts, USA (1995)
32. Object Management Group: OMG Object Constraint Language (OCL) – Version 2.3.1. Available at: http://www.omg.org/spec/OCL/2.3.1/PDF (2012)
33. Object Management Group: OMG Unified Modeling Language (OMG UML), Infrastructure – Version 2.4.1. Available at: http://www.omg.org/spec/UML/2.4.1/Infrastructure/PDF/ (2011)
34. Jürjens, J.: Modelling Audit Security for Smart-Card Payment Schemes with UML-SEC. In: Proceedings of the 16th International Conference on Information Security, Paris, France (2001)
35. Rodríguez, A., Fernández-Medina, E., Trujillo, J., Piattini, M.: Secure Business Process Model Specification through a UML 2.0 Activity Diagram Profile. Decision Support Systems **51**(3) (June 2011) 446–465
36. Hoisl, B., Strembeck, M.: Modeling Support for Confidentiality and Integrity of Object Flows in Activity Models. In: Proc. of the 14th International Conference on Business Information Systems (BIS), Berlin, Springer – Lecture Notes in Business Information Processing (LNBIP), Vol. 87 (June 2011) 278–289
37. Schefer, S., Strembeck, M.: Modeling Process-Related Duties with Extended UML Activity and Interaction Diagrams. In: Proc. of the International Workshop on Flexible Workflows in Distributed Systems, Electronic Communications of the EASST. (March 2011)
38. Schefer, S., Strembeck, M.: Modeling Support for Delegating Roles, Tasks, and Duties in a Process-Related RBAC Context. In: Proc. of the International Workshop on Information Systems Security Engineering (WISSE), Springer – Lecture Notes in Business Information Processing (LNBIP), Vol. 83 (June 2011)
39. Schefer-Wenzl, S., Strembeck, M.: Modeling Context-Aware RBAC Models for Business Processes in Ubiquitous Computing Environments. In: Proc. of the 3rd International Conference on Mobile, Ubiquitous and Intelligent Computing (MUSIC). (June 2012)
40. Fernández-Medina, E., Trujillo, J., Villarroel, R., Piattini, M.: Access Control and Audit Model for the Multidimensional Modeling of Data Warehouses. Decision Support Systems **42**(3) (December 2006) 1270–1289
41. Memon, M., Hafner, M., Breu, R.: SECTISSIMO: A Platform-independent Framework for Security Services. In: Proceedings of the Modeling Security Workshop in Association with MODELS 2008, Toulouse, France (2008)

A Textual Syntax for the `SecurityAudit` Package

```
<SecurityAudit>    ::= (<AuditEventSource> | <AuditRule>)*
<AuditEventSource> ::= '<<AuditEventSource>>' UML::Element.name ':' <Trigger> '->' <Publication> <AuditRule>
<Trigger>          ::= UML::Trigger.name
<Publication>      ::= UML::Signal.name | <Signal>
<Signal>           ::= UML::Signal.name '{' UML::Signal.attribute.name [',' UML::Signal.attribute.name]* '}'
<AuditRule>        ::= '<AR>' UML::AuditRule.name ( '->' <Subscription> ': {' <Action> [',' <Action>]* '}' <Condition>+ )?
<Subscription>     ::= <Publication>
<Action>           ::= UML::BehavioralFeature.name
<Condition>        ::= '<C>' ( UML::Condition.name | '[' <Operand> ',' <Operator> ',' <Operand> ']' )
<Operand>          ::= <Property> | <Constant>
<Property>         ::= UML::Property.name
<Constant>         ::= UML::LiteralSpecification
<Operator>         ::= UML::OperatorKind::EnumerationLiteral
```