Trabajo Fin de Máster en Programación y Tecnología Software

Máster en Investigación en Informática

Facultad de Informática

Universidad Complutense de Madrid

Developing Secure Business Applications from Secure BPMN Models

Javier Valdazo Parnisari

Director: Manuel García Clavel

Curso Académico: 2011-2012

Calificación obtenida: 5

Autorización de difusión

El abajo firmante, matriculado en el Máster en Investigación en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor, el presente Trabajo Fin de Máster: *Developing Secure Business Applications from Secure BPMN Models*, realizado durante el curso académico Curso Académico: 2011-2012 bajo la dirección de Manuel García Clavel en el Departamento de Sistemas Informáticos y Computación, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

En Madrid, a los 27 días del mes de septiembre de 2012,

Javier Valdazo Parnisari
02746255D

Manuel García Clavel
Director

Resumen

En este trabajo se propone una metodología para el desarrollo de aplicaciones seguras de procesos de negocios a partir de modelos BPMN Seguros. Es un proceso de tres pasos. En el primer paso, modelos de ActionGUI son automáticamente generados a partir de un modelo BPMN Seguro. Estos modelos de ActionGUI especifican tanto el comportamiento a alto nivel como la seguridad de la aplicación de negocios deseada. En el segundo paso, el modelador completará los modelos ActionGUI generados con la información relevante de la aplicación de negocios deseada que no puede ser expresada en el modelo BPMN dado. Finalmente, en el tercer paso, usando el generador de código disponible en ActionGUI, la aplicación de negocios deseada es generada automáticamente a partir de los modelos previamente completados.

Palabras clave: Seguridad basada en modelos, BPMN, ActionGUI, BPMN Seguro, SoD, BoD, MDE.

Abstract

In this work we propose a methodology for developing secure business applications from Secure BPMN models. In a nutshell, it is a three-step process. In the first step, ActionGUI models will be automatically generated from the given Secure BPMN model. These ActionGUI models specify the high-level behaviour, and the security of the desired business application. In the second step, the modeler will complete the generated ActionGUI models with all the relevant information about the desired business application that can not be expressed in the given BPMN model. Finally, in the third step, the desired business application will be automatically generated from the finalized models, using the available ActionGUI code generator.

Keywords: Model-driven security, BPMN, ActionGUI, Secure BPMN, SoD, BoD, MDE.

Acknowledgments

First of all I would like to express my gratitude to Manuel Clavel for his support, his effort, his advices, and for guiding me during my very first steps in research.

I would like to thank Samuel Burri for his very insightful comments and fruitful suggestions when I was given my first steps in this research line.

I would also like to thank to Luis Llana, Miguel Palomino and Jaime Sánchez for the evaluation of this work and the feedback provided.

I would like to specially thank to IMDEA for supporting my research.

I would like to thank the EU FP7-ICT-2009.1.4 Project, "NESSoS: Network of Excellence on Engineering Secure Future Internet Software Services and Systems" (256980), to the Spanish Ministry of Science and Innovation Project "DESAFIOS-10" (TIN2009-14599-C03-01), and to Comunidad de Madrid Program "PROMETIDOS-CM" (S2009TIC-1465) for partially supporting this work.

Thanks to Narciso for caring about every student in the Master, always seeking for the student interest.

I would also like to thank Manuel, Marina, Caro, Miguel and Gonzalo, for their constant help, and for making a great research group to work with.

Thanks to Ale, Juli and Caro, for their support, their advices, but mainly for their friendship.

Also I would like to thank to my friends, either in Spain or in Argentina, for always being there for me.

I would like to thanks my spanish relatives for making me feel at home.

Last, thanks to my family, that has always been at my side in all the important decisions of my life, specially decisions regarding my career. Thanks mom, that given this shared path of life, in different timeline, you understand everything I go through.

Contents

1

Introduction

Broadly speaking, business processes are used to describe which activities need to be completed to accomplish a specific business goal, in which order they are to be executed, and by whom. They are very useful for assigning responsibilities among agents (humans or computers) and for defining interactions among them. BPMN [19] is a well-known language for modeling business processes. It graphically depicts business processes using *workflows*, where *nodes* represent activities, *edges* represent activity flow, and *gateways* represent activity coordination. Business applications are software solutions ensuring that all the activities which encompass a business process are completed in the required order and by the required agents. The standard components of a business application are: (i) a *task management window*, where agents can choose the next task (activity instance) to be executed; and (ii) for each activity, a *task execution window*, where agents can execute to completion a previously chosen task. The abuse of special privileges is a serious security threat in many business processes. Secure BPMN [2, 8] has been proposed as a formal way of integrating *separation of duties* (SoD) and *binding of duties* (BoD) requirements into business processes. SoD requirements prevent a user from executing two activities with conflicting interest. BoD requirements prevent widespread dissemination of sensitive information by forcing that two related activities are to be executed by the same agent.

The main contribution of the work presented here is a novel methodology (unpublished yet) for developing secure business applications from Secure BPMN models. In a nutshell, it is a three-step process. In the first step, ActionGUI models will be automatically generated from the given secure BPMN model. ActionGUI [9] is a language for modeling security-aware, data-centric business applications, to whose design and implementation we have contributed as part of our work. The generated ActionGUI models specify the high-level behaviour of the desired business application, based on the information contained in the given BPMN model. In the second step, the modeler will complete the generated ActionGUI models with all the relevant information about the desired business application that can not be expressed in the given BPMN model. In the third step, the desired business application will be automatically generated from the finalized models, using the available ActionGUI code generator.

Finally, our motivation is twofold. On the one hand, to the best of our knowledge, current BPMN tools (including Activiti [1] or JBPM [16]) do not support Secure BPMN models. On the other hand, although these tools generate software from models, they also require manually

encoding all the details about the desired business applications that can not be modeled in BPMN and that can not be generated from the templates provided by these tools, including security concerns, like SoD or BoD requirements.

2

Context

2.1 Model-driven software engineering

As time pass by, software systems grows in complexity and software developers must work at increasingly higher levels of abstraction to cope with this complexity. Nowadays, modeling software is a key way for developers to work at those higher levels. The role played by models in software projects is comparable to the role played by blueprints in engineering projects. In the long run, a detailed model saves time and money since it allows developers to consider alternatives, select the best option, work out details, and achieve agreement before anyone starts building the application .

Proponents of model-driven engineering have in the past been guilty of making overambitious claims: positioning it as the Holy Grail of software engineering where modeling completely replaces programming. This vision is, of course, unrealizable in its entirety for simple complexity-theoretic reasons. If the modeling languages are sufficiently expressive then basic problems such as the consistency of the different models or views of a system become undecidable. However, there are specialized domains where MDE can truly deliver its full potential, as it has been shown [9].

2.2 Model-driven security

In model-driven development, system designs are specified using graphical modeling languages like UML [20] and system artifacts such as code and configuration data are automatically generated from the models. Model-driven security [17, 7, 3, 4, 11, 6, 13, 5] is a specialization of this paradigm, where system designs are modeled together with their security requirements and security infrastructures are directly generated from the models. More specifically, models can be used for the following four activities in the development of secure system [5]:

A1. Precisely documenting security requirements together with design requirements.

A2. Analyzing security requirements.

A3. Model-based transformation, such as migrating security policies on application data to policies for other system layers or artifacts.

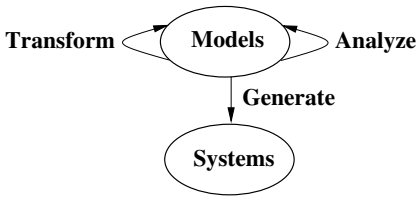


Figure 2.1: Use of Models in Model-Driven Security

A4. Generating code, including complete, configured security infrastructures.

Figure 2.1 depicts these activities and their interrelationships. Designers specify security-design models that combine security and design requirements (A1). As our modeling languages have a well-defined semantics, we can formally analyze these designs (A2). When designing secure systems, security may be relevant at different system layers or views. Using model transformations, we can migrate a security policy from one model to other models (A3). Finally, we can use tools to automatically generate code and other system artifacts directly from the models (A4).

2.3 Data-centric applications

Data-centric applications are applications that manage information, typically stored in a database. In many cases, users access this information through graphical user interfaces (GUIs). Informally, a GUI consists of widgets (e.g., windows, text-fields, lists, and combo-boxes), which are visual elements that display and store information and support events (like "clicking-on" or "typing-in"). A GUI defines the layout for the widgets, as well as the actions that the widgets' events trigger either on the application's database (e.g., to create, delete, or update information) or upon other widgets (e.g., to open or close a window).

There is an important, but little explored, link between visualization and security: When the application data is protected by an access control policy, the application GUI should be aware of and respect this policy. For example, the GUI should not display options to users for actions (e.g., to read or update information) that they are not authorized to execute on application data. This, of course, prevents the users from getting (often cryptic) security warnings or error messages directly from the database management system. It also prevents user frustration, for example from filling out a long electronic form only to have the server reject it because the user lacks a permission to execute some associated action on the application data. However, manual encoding the application's security policy within the GUI code is cumbersome and error prone. Moreover, the resulting code is difficult to maintain, since any changes in the security policy will require manual changes to the GUI code.

2.4 Business applications

Broadly speaking, business processes are used to describe which activities need to be completed to accomplish a specific business goal, in which order they are to be executed, and by whom. They are very useful for assigning responsibilities among agents (humans or computers) and for defining interactions among them [12]. BPMN [19] is a well-known language for modeling

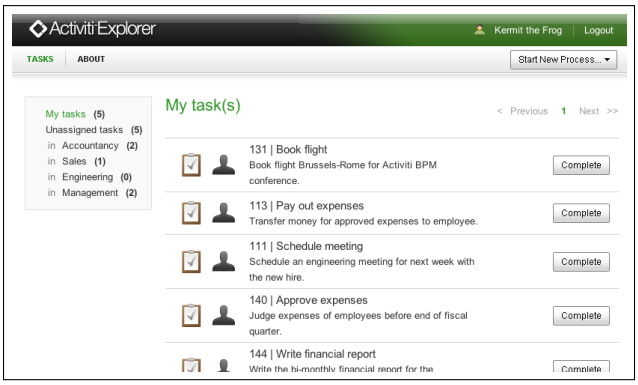


Figure 2.2: A typical task management window (generated using the Activiti framework [1]).

business processes. It graphically depicts business processes using *workflows*, where *nodes* represent activities, *edges* represent activity flow, and *gateways* represent activity coordination.

The abuse of special privileges is a serious security threat in many business processes.[10, 14]. Secure BPMN [2, 8] has been proposed as a formal way of integrating *separation of duties* (SoD) and *binding of duties* (BoD) requirements into business processes. SoD requirements prevent a user from executing two activities with conflicting interest. BoD requirements prevent widespread dissemination of sensitive information by forcing that two related activities are to be executed by the same agent.

In general, business applications are software solutions ensuring that all the activities which encompass a business process are completed in the required order and by the required agents. The standard components of a business application are: (i) a *task management window* (like the one shown in Figure 2.2), where agents can choose the next task (activity instance) to be executed; and (ii) for each activity, a *task execution window*, where agents can execute to completion a previously chosen task.[1]

[1]We are well-aware that this is a simplification. Typically, agents must navigate through several windows in order to complete a complex task. However, for our present purposes, it is enough to consider this simple case.

3

ActionGUI

ActionGUI [9] is a domain-specific modeling language for developing data-centric applications with fine grained access control policies. It supports *model-driven engineering*. More specifically, it fosters a development methodology that focuses on creating models of different aspects or views of data-centric applications from which artifacts such as code and configuration data is automatically generated.

Crucially, ActionGUI supports the principle of *separation of concerns*. In particular, ActionGUI models consists of three models:

- A *data* model that defines the application domain.
- A *security* model that defines the application fine grained access control policy.
- A *GUI* model that defines the application graphical interface.

As expected, the security model and the GUI model *depends* on the data model, since the former specifies which users can access which data and the latter defines through which graphical widgets the user will access this data. (Thus, a change in the data model will generally imply a change in the other two models.) However, the security model and the GUI model are truly independent from each other. We believe this is in correspondence with what happens in reality: the security engineer should not be concerned about the application graphical user interface, while the GUI designer may not know in full detail the access control policy that the application must respect.

ActionGUI offers the following advantages for developers of data-centric applications with fine grained access control policies. By working with models, GUI designers can focus on the GUI's layout and behavior, instead of wrestling with the different, often complex, technologies that are used to implement them. Then, by supporting the separating concerns, GUI designers can make changes in the GUI model without having to agree upon these changes with the security engineers, and viceversa.[1] Last but not least, ActionGUI models are formal objects and

[1] Of course, radical changes on the access control policy may turn useless a graphical user interface. When *usability* is taking into account, the proclaimed *independence* of the security model and the GUI model may demand some further discussion.

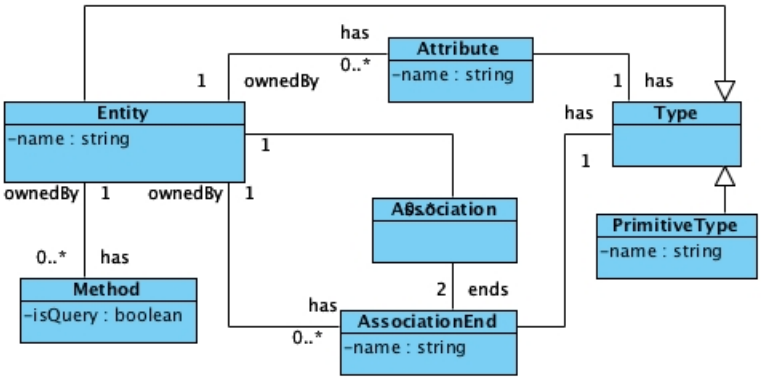


Figure 3.1: The ComponentUML metamodel.

therefore developers can reason about their properties. In fact, one can reason about the properties of each of the models conforming an ActionGUI models. E.g., Is there a valid instance of the data model? Will a user in a given role ever be allowed to access a given resource? Will a user ever reach a window from which it could attempt to access a given resource? Furthermore, one can reason about the properties of the ActionGUI models themselves. E.g., Will a user in a given role ever be allowed to triggered all the actions associated to a given widget event?

In the remaining of this section, we introduce the modeling languages that we use within ActionGUI. These languages are: ComponentUML, for modeling data; SecureUML, for modeling the access control policy; and GUIML, for modeling the application's GUI. To illustrate the main modeling concepts and relationships provided by these languages, we work through a running example: a simple chat application named ChitChat, which supports multiple chat rooms where users can converse with each other in different chat rooms.

3.1 Data models (ComponentUML)

We begin then with ComponentUML, which is the language that we use for modeling the application data. ComponentUML also gives us a context to introduce the constraint language OCL [18], which we intensively use in ActionGUI when modeling both access control policies and the application's GUIs.

Data models provide a data-oriented view of a system. Typically they are used to specify how data is structured, the format of data items, and their logical organization, i.e., how data items are grouped and related. ComponentUML essentially provides a subset of UML class models where *entities* (classes) can be related by *associations* and may have *attributes* and *methods*. Attributes and *association-ends* have *types* (either primitive types or entity types). As expected, the type associated to an attribute is the type of the attribute's values, and the type associated to an association-end is the type of the objects which may be linked at this end of the association. The ComponentUML metamodel is shown in Figure 3.1.

Example 1 (The ChitChat data model) *In our ChitChat application, each user has a nickname, a password, an e-mail address, a mood message, and a status. The user may participate and be invited to participate in any number of chat rooms. Each chat room is created by a user, has a*

name, a starting and ending date, and it manages the messages sent by its participants.

The model shown in Figure 3.2 specifies the data model of the ChitChat application, using the textual notation of ComponentUML.

OCL: constraints and queries

Modeling, specially software modeling, has traditionally been a synonym for producing diagrams. Most models consist of a number of "bubbles and arrows" pictures and some accompanying text. The information conveyed by such a model has a tendency to be incomplete, informal, imprecise, and sometimes even inconsistent. Many of the flaws in the model are caused by the limitations of the diagrams being used. A diagram simply cannot express the statements that should be part of a thorough specification. To specify software systems, expressions written in a formal language offer a number of benefits over the use of diagrams: they are unambiguous and cannot be interpreted differently by different people, for example, an analyst and a programmer; they make the model more precise and more detailed; and they can be checked by automated tools to ensure that they are correct and consistent with other elements of the model. On the other hand, a model written in a formal language that uses expressions alone is often not easily understood. The good thing about "bubbles and arrows" pictures is that their intended meaning is easy to grasp.

The UML notation is largely based on diagrams. However, to provide the level of conciseness and expressiveness that is required for certain aspects of a design, the UML standard defines the Object Constraint Language (OCL) [18]. Over the last years, the domain of application of modeling languages and of modeling in general has evolved to include issues like domain-specific metamodels definition, model transformation, design model testing, and model validation and simulation. Most of these new applications make extensive use of OCL.

As part of the UML standard, OCL was originally intended for modeling properties that could not be easily or naturally captured using graphical notation (e.g., class invariants in a UML class diagram). In fact, OCL expressions are always written in the context of a model, and they are evaluated on an instance of this model. This evaluation returns a value but does not change anything; OCL is a side-effect free language.

OCL is a strongly type language. Expressions either have a primitive type (namely, Boolean, Integer, Real, and String), a class type, or a collection type, whose base type is either a primitive type or a class type. OCL provides the standard operators on primitive types and on collections. For example, the operator includes checks whether an object is part of a collection, and the operator isEmpty checks whether a collection is empty. More interestingly, OCL provides a dot-operator to access the values of the objects' attributes and association-ends. For example, let u be an object of the class ChatUser. Then, the expression u.nickname refers to the value of the attribute nickname for the ChatUser u, and the expression u.participates refers to the objects linked to the ChatUser u through the association-end participates. Furthermore, OCL provides the operator allInstances to access to all the objects of a class. For example, the expression ChatRoom.allInstances() refers to all the objects of the class ChatRoom. Finally, OCL provides operators to iterate on collections. These are forAll, exists, select, reject, and collect. For example, ChatUser.allInstances()−>select(u|u.status='on−line') refers to the collection of objects of the class ChatUsers whose attribute status has the value "on-line".

Example 2 (ChitChat's entity invariants) *To illustrate the syntax (and the semantics) of the OCL language, we formalize here some entity (class) invariants for ChitChat's data model. For*

```
entity ChatUser isUser{
   String nickname
   String password
   String email
   String moodMsg
   String status

   Set(ChatMessage) msgSent oppositeTo from
   Set(ChatRoom) participates oppositeTo participants
   Set(ChatRoom) owns oppositeTo ownedBy
   Set(ChatRoom) invitedTo oppositeTo invitees
}

entity ChatMessage{
   String body
   ChatUser from oppositeTo msgSent
   ChatRoom chat oppositeTo messages
}

entity ChatRoom{
   String name
   Date start
   Date end
   Set(ChatMessage) messages oppositeTo chat
   Set(ChatUser) participants oppositeTo participates
   ChatUser onwedBy oppositeTo owns
   Set(ChatUser) invitees oppositeTo invitedTo
}
```

Figure 3.2: The ChitChat data model.

example, the following OCL expression formalizes that users' nicknames must be unique:

```
ChatUser.allInstances()−>forall(u1,u2|
    u1 <> u2 implies u1.nickname <> u2.nickname).
```

Similarly, we can formalize that the status of a ChitChat user is either "off-line" or "on-line" using the following OCL expression:

```
ChatUser.allInstances()−>forall(u|
    u.status='on−line' or u.status='off−line').
```

Finally, we can formalize that each message has exactly one sender:

```
ChatMessage.allInstances()−>forAll(m|m.from−>size()= 1).
```

3.2 Security models (SecureUML)

SecureUML [7] is a modeling language for formalizing access control requirements that is based on RBAC [15]. In RBAC, permissions specify which roles are allowed to perform given operations. These roles typically represent job functions within an organization. Users are granted permissions by being assigned to the appropriate roles based on their competencies and responsibilities in the organization. RBAC additionally allows one to organize the roles in a hierarchy where roles can inherit permissions along the hierarchy. In this way, the security policy can be described in terms of the hierarchical structure of an organization. However, it is not possible in RBAC to specify policies that depend on dynamic properties of the system state, for example, to allow an operation only during weekdays. SecureUML extends RBAC with *authorization constraints* to overcome this limitation.

SecureUML provides a language for specifying access control policies for actions on protected resources. However, it leaves open what the protected resources are and which actions they offer to actors. These are specified in a so-called "dialect". Figure 3.3 shows the SecureUML metamodel. Essentially, it provides a language for modeling *roles* (with their hierarchies), *permissions*, *actions*, *resources*, and *authorization constraints*, along with their assignments, i.e., which permissions are assigned to a role, which actions are allowed by a permission, which resource is affected by the actions allowed by a permission, which constraints need to be satisfied for granting a permission, and, finally, which resource is affected by an action.

In ActionGUI, we use a specific dialect of SecureUML for modeling the access control policy on data models, named SecureUML+ComponentUML. Its metamodel provides the connection between SecureUML and ComponentUML. Essentially, in this dialect of SecureUML, the protected resources are the entities, along with their attributes, methods, and association-ends (but not the associations as such), and the actions that they offer to the actors are those shown in Figure 3.4. There are two classes of actions: atomic and composite. The *atomic* actions are intended to map directly onto actual operations on the database. These actions are: create and delete for entities; read and update for attributes; create and delete for association-ends; and execute for methods. The underlined actions are the *composite* actions, which hierarchically group lower-level actions. Composite actions allow modelers to conveniently specify permissions for sets of actions. For example, the full access action for an attribute groups together the read and update actions for this attribute.

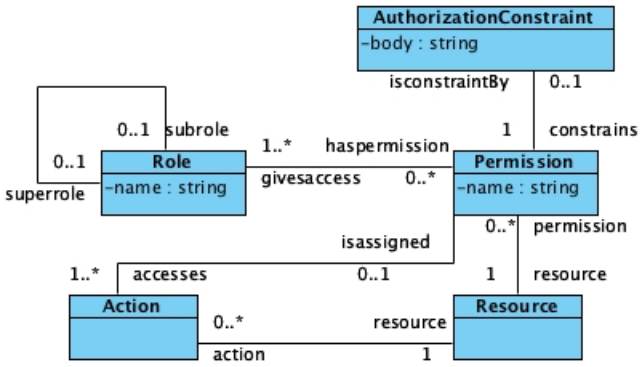


Figure 3.3: The SecureUML metamodel.

Resource	Actions
Entity	create, read, update, delete, full access
Attribute	read, update, full access
Method	execute
AssociationEnd	read, create, delete, full access

Figure 3.4: The SecureUML+ComponentUML actions on protected resources.

Finally, in SecureUML+ComponentUML, authorization constraints are specified using OCL, extended by four keywords, self, caller, value, and target. These keywords have the following meanings:

- self refers to the root resource upon which the action will be performed, if the permission is granted. The root resource of an attribute, an association-end, or a method is the entity to which it belongs.
- caller refers to the actor that will perform the action, if the permission is granted.
- value refers to the value that will be used to update an attribute, if the permission is granted.
- target refers to the object that will be linked at the end of an association, if the permission is granted.

Example 3 (The ChitChat access control policy) *For the sake of our running example, consider the following (partial) access control policy for the ChitChat application:*

- *Only administrators can create or delete users;*
- *Administrators can read any user's nickname, email, mood message, and status.*
- *Any user can read and update its own nickname, password, email, mood message, and status.*
- *Any user can read other users' nicknames, mood messages, and status.*

- *Users can join a chat room by invitation only, but they can leave at any time.*

The model shown in Figure 3.5 specifies this access control policy, using the textual notation of SecureUML+ComponentUML. For example, the permission DisjointChat authorizes any user to leave a chat room at any anytime. More precisely, it allows any user caller *to delete a* participates*-link between a user* self *and a chat room* target *(meaning that the user* self *leaves the chat room* target*), but only if the user* caller *is indeed the user* self *(that is, the user* caller *is the one leaving the chat room* target*), and also the chat room* target *indeed belongs to the collection of chat rooms linked to the user* caller *through the association-end* participates *(that is, the* caller *is actually participating in the chat room* target*).*

3.2.1 GUI models (GUIML)

GUIML is a modeling language for formalizing GUIs of data-centric applications. The GUIML metamodel is shown in Figure 3.6. In a nutshell, GUIML provides a language to model *widgets* (e.g., windows, text-fields, buttons, lists, and tables), *events* (e.g., clicking-on, typing-in), and *actions*, which can be on data (e.g., to update a property of an element in the database) or on other widgets, (e.g., to open a window), as well as the associations that link the widgets with the events that they support and the events with the actions that they trigger. In addition, GUIML provides support to formally model the following features:

- Widgets can be displayed in *containers*, which are also widgets (e.g., a window can contain other widgets).
- Widgets may own *variables*, which are in charge of storing information for later use.
- Events may be only supported upon the satisfaction of specific *conditions*, whose truth value can depend on the information stored in the widgets' variables or with the in the database.
- Actions may be only triggered upon the satisfaction of specific *conditions*, whose truth value can depend on the information stored in the widgets' variables or in the database.
- Actions may take their *arguments* (values that instantiate their parameters) from the information stored in the widgets' variables or in the database.

The GUIML metamodel's invariants specify: (i) for each type of widget, the "default" variables that widgets of this type always own; (ii) for each type of widget, the type of events that widgets of this type may support; and (iii) for each type of action, the arguments that actions of this type require, as well as the arguments (if any) that these actions may additionally take. In particular, the invariants of GUIML's metamodel formalize, among others, the following constraints about the different types of widgets:

- *Windows.* They can contain any type of widget, except windows. Windows are not contained in any widget.
- *Text-field.* They can be typed-in. By default, each text-field owns a variable text of type string, which stores the last string typed-in by the user. The value of the variable text is permanently displayed in the text-field.

```
role Admin {
  permission AnyUser context ChatUser{
    create
    delete
    read nickname
    read email
    read moodMsg
    read status
  }
}

role User{
  permission SelfUser context ChatUser{
    fullAccess nickname constrainedBy [self=caller]
    fullAcesss password constrainedBy [self=caller]
    fullAcesss email constrainedBy [self=caller]
    fullAcesss moodMsg constrainedBy [self=caller]
    fullAccess status constrainedBy [self=caller]
  }
  permission OtherUser context ChatUser{
    read nickname
    read moodMessage
    read status
  }
  permission JointChat context ChatUser{
    add participates constrainedBy [self=caller and caller.invitedTo->includes(target)]
  }
  permission DisjointChat context ChatUser{
    remove participates constrainedBy [self=caller and caller.participates->includes(target)]
  }
}
```

Figure 3.5: The ChitChat security model.

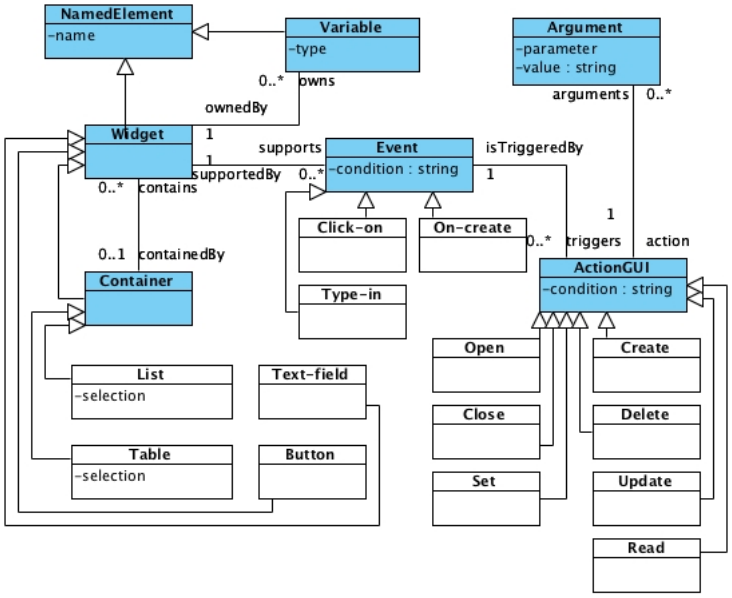


Figure 3.6: The GUIML metamodel.

- *Button.* They can be clicked-on.
- *List.* They contain exactly one text-field. By default, each list owns a variable rows of type collection. A list displays as many rows as elements are in the collection stored by the variable rows, each row containing exactly one instance of the text-field contained by the list. By default, each instance of this text-field owns a variable row whose value is the element associated to this row from those stored by the variable rows. Finally, by default, each list owns a variable selected that holds the element associated to the last row selected by the user.
- *Combo-box.* They are similar to lists, except that rows are displayed in a drop-down box.
- *Table.* They are similar to lists, except that they can contain any number of text-fields, buttons, lists, or even tables.

Also, the invariants of GUIML's metamodel formalize, among others, the following invariants about the different types of actions:

- *Create.* It creates a data item in the database. It takes two arguments: the type of the new data item (*type*) and the variable that will store this element for later reference (*variable*).
- *Delete (entities).* It deletes a data item in the database. It takes as argument the element to be deleted (*object*).
- *Read.* It reads the value of a data item's attribute in the database. It takes three arguments: the data item whose property is to be read (*object*); the property to be read (*attribute*); and the variable that will store, for later reference, the value read (*variable*).

- *Update.* It modifies the value of a data item's attribute in the database. It takes three arguments: the data item whose attribute is to be modified (*object*); the attribute to be modified (*attribute*); and the new value (*value*).
- *Create (association-ends).* It creates a new link in the database between two data items. It takes three arguments: the source data item (*sourceObject*); the target data item (*targetObject*); and the association-end (*associationEnd*) through which the target data item will be linked to the source data item.
- *Delete (association-ends).* It deletes a link in the database between two data items. It takes three arguments: the source data item (*sourceObject*); the target data item (*targetObject*); and the association-end (*associationEnd*) from where the target data item will be removed.
- *Open.* It opens a window. It takes as argument the window to be opened (*target*); additionally, for any of this window's variables, it can take as argument a value to be assigned to this variable when opening the window.
- *Back.* It goes back to the window from which a window was open.
- *Set.* It assigns a new value to a widget's variable. It takes two arguments: the variable (*target*) and the value to be assigned to this variable (*value*).

Finally, actions' conditions and arguments are specified in GUIML models using OCL, extended with the widget's variables. As expected, when evaluating an OCL expression that contains a widget's variable, the value of the corresponding subexpression is the value currently stored in the variable. In case of ambiguity, a widget's variable is denoted by its name, prefixed by the name of its widget (followed by a dot). Also, in case of ambiguity, the name of a widget is prefixed by the name of its container (followed by a dot). Notice that, within the same containers, widgets have unique names. Moreover, a widget's variable can only be used within the window that contains its widget, either directly or indirectly.

Example 4 (The ChitChat login window) *To continue with our running example, consider the following interface for allowing a registered user to login into the ChitChat application: a window* loginWi *containing:*

- *a writable text-field* nicknameEn*, for the user to type its nickname in;*
- *a writable text-field* passwordEn*, for the user to type its password in; and*
- *a clickable button* loginBu*, for the user to login, using as its nickname and password the strings that it typed in the text-fields* nicknameEn *and* passwordEn*, respectively. Upon successful authentication, the user will be directed to the application's main menu window* menuWi *as the logged-in user.*

The model shown in Figure 3.7 specifies this login, using the textual notation of GUIML. Notice that

- *The* authenticated user *should be the registered user in the database whose nickname and password coincide with the values of the* text*-variables owned, respectively, by text-fields* nicknameEn *and* passwordEn*. Using our extended OCL, we define the* authenticated user *as follows:*

```
ChatUser.allInstances()->any(u|
```

```
Window loginWi{
   TextEntry nicknameEn
   TextEntry passwordEn
   Button loginBu {
      ChatUser loggedUser
      Event OnClick{
         if(ChatUser.allInstances()->exists(u|u.nickname=nicknameEn.text
               and u.password=passwordEn.text)){
           loggedUser := ChatUser.allInstances()->any (u|
               u.nickname=nicknameEn.text and u.password=passwordEn.text)
           loggedUser.status := 'on-line'
           open menuWi(caller := loggedUser)
         }
      }
   }
}
```

Figure 3.7: The ChitChat login window.

```
u.nickname= nicknameEn.text
    and u.password=passwordEn.text)
```

Recall that, as an invariant of the ChitChat data model, we specified that nicknames shall be unique. Thus, although the any*-iterator will return* any *registered user satisfying the body of the* any*-iterator, there will be at most one such registered users.*

- *The condition for opening the window* menuWi *should be the existence in the database of a registered user whose nickname and password coincide with the values of the* text*-variables owned, respectively, by text-fields* nicknameEn *and* passwordEn*. We define this condition as follows:*

```
ChatUser.allInstances()->exists(u|
  u.nickname= [nicknameEn.text]
    and u.password=[passwordEn.text])
```

Example 5 (The ChitChat menu window) *Consider now the following interface for allowing a logged-in user to choose an option from ChitChat's main menu: a window* menuWi*, owning a variable* caller *which stores the logged-in user, and containing:*

- *a selectable list* usersLi *with as many rows as registered users are online, each of these rows containing an unwritable text-field* nicknameLa *showing the nickname of the registered user associated to this row;*
- *a clickable button* editProfileBu *for the* caller *to access the interface for editing the profile (i.e., name, password, email, mood message, and status) of the user selected in the list* usersLi*;*

```
Window menuWi{
   ChatUser caller
   Role role
   Table usersLi {
      Sequence(ChatUser) rows := ChatUser.allInstances()−>select(u|u.status='on−line')
      Label nicknameLa {
         String text := row.nickname
      }
   }
   Button editProfileBu {
      Event OnClick{
         open editProfileWi(selectedUser := usersLi.selected)
      }
   }
   Button createChatBu {
      Event OnClick{
         open createChatWi()
      }
   }
   Button closeChatBu {
      Event OnClick{
         back
      }
   }
}
```

Figure 3.8: The ChitChat menu window.

- *a clickable button* createChatBu*) for the* caller *to access the interface for creating a new chat room; and*
- *a clickable button* closeChatBu *for the* caller *to close the window.*

The model shown in in Figure 3.8 specifies this menu window, using the textual notation of GUIML. Notice that the collection of data items to be displayed in the list usersLi*, namely, the* online users*, is defined, using the ActionGUI's extension of OCL, as follows:*

```
ChatUser.allInstances()−>select(u|u.status= 'on−line')
```

3.3 Security-aware GUI models

In [9] we spell out our model-driven engineering approach for developing security-aware GUIs for data-centric applications. The backbone of this approach, illustrated in Figure 3.9, is a model transformation that automatically lifts the access control policy modeled at the level of the data to the level of the GUI [6]. More precisely, given a security model (specifying the access control policy on the application data) and a GUI model (specifying the actions

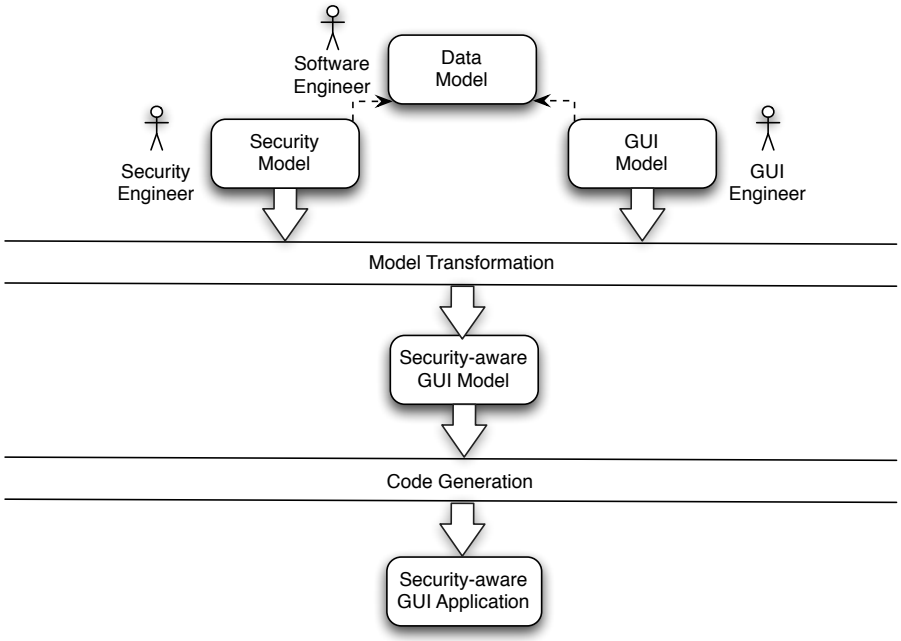


Figure 3.9: Model-driven development of security-aware GUIs.

triggered by the events supported by the GUI's widgets), our model transformation generates a GUI model that is security-aware. The key idea underlying this transformation is that the link between visualization and security is ultimately defined in terms of data actions, since data actions are both controlled by the security policy and triggered by the events supported by the GUI. Thus, under our approach, the process of modeling and generating security-aware GUIs has the following parts:

1. Software engineers specify the application-data model (ComponentUML).

2. Security engineers specify the security-design model (SecureUML).

3. GUI designers specify the application GUI model (GUIML).

4. A model transformation automatically generates a security-aware GUI model from the security model and the GUI model.

5. A code generator automatically produces a security-aware GUI from the security-aware model.

To support a full model-driven engineering development process, we have built a toolkit, named the ActionGUI Toolkit [9]. This features specialized model editors for data, security, and GUI models, and implements the aforementioned model transformation to automatically generate security-aware GUI models. Moreover, our toolkit includes a code generator that, given a security-aware GUI model, automatically produces a complete web application, ready to be deployed in web containers such as Tomcat or GlassFish.

Example 6 (The ChitChat edit-profile window) *To motivate the problem faced by a GUI designer when modeling a security-aware GUI, let us continue with our running example. Consider the interface for allowing a logged-in user* caller *to edit the profile of a previously chosen user* selectedUser*, which may of course be the* caller *itself. More specifically, this interface shall consist of a window such that:*

1. *The current values of the* selectedUser*'s profile are displayed when opening the window.*
2. *The* caller *can type in the new values (if any) for the* selectedUser*'s profile.*
3. *The* selectedUser*'s profile is updated with the new values typed in by the* caller *when he or she clicks on a designated button.*

Recall that a registered user's profile is composed of the following attributes: nickname, password, mood message, email, and status. Recall also that the access control policy for reading and updating users' profiles, as specified in Figure 3.5, is the following:

4. *A user is always allowed to read and update its own nickname, password, mood message, email, and status.*
5. *A user is allowed to read another user's nickname, mood message, and status, but not the user's password or email.*
6. *An administrator is always allowed to read a user's nickname, mood message, status, and email, but not the user's password.*

Now, if the GUI designer only takes into consideration the functional requirements (1–3), the ChitChat edit-profile window can be modeled as shown in Figure 3.10. Namely, a window editProfileWi*, which owns the variable* selectedUser *and* caller*, and contains:*

- *A writable text-field* nicknameEn*, for the* caller *to type in the new value (if any) with which to update the* selectedUser*'s nickname. Notice that when the text-field* nicknameEn *is created, its "default" variable* text *will be assigned the current value of the* selectedUser*'s nickname, and therefore this value will be the string initially displayed in the text-field* nicknameEn*, as requested.*
- *Analogous writable text-fields for each of the other elements in a registered user's profile: password, mood message, email, and status.*
- *A clickable button* updateBu *for the* caller *to trigger the sequence of actions that will update, as requested, the* selectedUser*'s nickname, password, mood message, and status, with the new values (if any) typed by the* caller *in the corresponding text-fields.*

Clearly, the edit-profile window modeled in Figure 3.10 does not satisfy 'per se' the security requirements (4–6): any caller *can read and update any value contained in the profile of any* selectedUser*! Then, shall we modify the edit-profile window to specify, for each action triggered by an event, and for each role considered by the ChitChat access control policy, the conditions under which the given action can be securely executed by a user with the given role? The answer is 'No', since the ActionGUI model transformation will automatically lift the ChitChat access control policy to the edit-profile window model.*

```
Window editProfileWi{
    ChatUser caller
    ChatUser selectedUser
    TextEntry nicknameEn{
        String text := selectedUser.nickname
    }
    TextEntry passwordEn{
        String text := selectedUser.password
    }
    TextEntry moodMsgEn{
        String text := selectedUser.moodMsg
    }
    TextEntry emailEn{
        String text := selectedUser.email
    }
    TextEntry statusEn{
        String text := selectedUser.status
    }
    Button updateBu {
        Event OnClick{
            selectedUser.nickname := nicknameEn.text
            selectedUser.password := passwordEn.text
            selectedUser.moodMsg := moodMsgEn.text
            selectedUser.email := emailEn.text
            selectedUser.status := statusEn.text
        }
    }
}
```

Figure 3.10: The ChitChat edit-profile window (although security-unaware).

4

BPMN

The standard Business Process Model and Notation (BPMN) [19] was developed by the Object Management Group. “The primary goal of BPMN is to provide a notation that is readily understandable by all business users, from the business analysts that create the initial drafts of the processes, to the technical developers responsible for implementing the technology that will perform those processes, and finally, to the business people who will manage and monitor those processes.” The intent of BPMN is to standardize a business process model and notation in the face of many different modeling notations and viewpoints.

There are different types of Diagrams within BPMN. Namely, Process Diagrams, Choreography Diagrams, and Collaborations Diagrams. For the purpose of this work, we are only interestd in Process Diagrams. In particular, our work is centered around the following modeling elements of BPMN Process Diagrams.

Activities

An Activity is work that is performed within a Business Process. An Activity can be Atomic or Non-Atomic (Compound).

There are different types of Activities within a BPMN Process. Namely, Task Activities, Sub-Process Activities, and Call Activities. Here we are only interested in Atomic Task Activities.

Task

A Task is an Atomic Activity within a Process flow. A Task is used when the work in the Process cannot be broken down to a finer level of detail.

There are different types of Tasks within BPMN. Namely, User Tasks, Service Tasks, Send Tasks, Receive Tasks, Manual Tasks, Business Rule Tasks, and Script Tasks. Here we are only interested in User Tasks. A User Task is known as the typical workflow task where a human performs the task with the assistance of a software application and is scheduled through a task list manager of some sort. We represent User Tasks as shown in Figure 4.1.

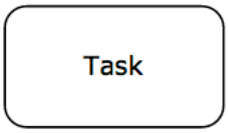


Figure 4.1: Task

Figure 4.2: Flow

Flows

A Sequence Flow is used to show the order of Flow Elements in a Process or a Choreography. Each Sequence Flow has only one source and only one target. The source and target must be from the set of the following Flow Elements: Events, Activities, and Gateways.

There are different types of Sequence Flows within BPMN. Namely, Default Flow, Uncontroled Flows, Conditional Flows, Exception Flows, Message Flows or Compensation Associations. Here we are only interested in Uncontroled Flows and Conditional Flows (but only after Exclusive Gateways). Uncontrolled Flow refers to a flow that is not affected by any conditions or does not pass through any Gateway. The simplest example of this is a single Sequence Flow connecting two Activities. Conditional Flow refers to a flow with a condition Expression that is evaluated at runtime to determine whether or not the flow is continued. We support conditional flows only after exclusive gateways. We represent Sequence Flows as shown in Figure 4.2.

Events

An Event is something that *happens* during the course of a Process. Events affect the flow of the Process and usually have a cause or an impact and in general require or allow for a reaction.

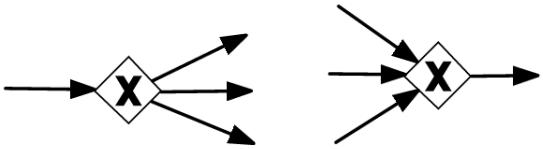
There are three main types of events: Start Events that indicates where a process will start; End Events which indicate where a path of a process will end; and Intermediate Events which indicate where something happens somewhere between the start and end of a process. Here we are only interested in Basic Start and End Events We represent them as shown in Figure 4.3. In particular, we do not consider the following types of Start Events: Message, Timer, Conditional, Signal, Multiple, Parallel Multiple. Also, we do not consider the following types of End Events: Message, Escalation, Error, Cancel, Compensation, Signal, Terminate, and Multiple.

(a) Start Event (b) End Event

Figure 4.3: Events

(a) Exclusive Gateway (b) Parallel Gateway

Figure 4.4: Gateways



(a) Exclusive Gateway Multiple Outputs (b) Exclusive Gateway Multiple Inputs

Figure 4.5: Exclusive Gateways

Gateways

Gateways are used to control how flows interact as they converge and diverge within a Process. If the flow does not need to be controlled, then a Gateway is not needed. The term *Gateway* implies that there is a gating mechanism that either allows or disallows passage through the Gateway–that is, as tokens arrive at a Gateway, they can be merged together on input and/or split apart on output as the Gateway mechanisms are invoked. Gateways can define all the types of Sequence Flow behavior: Decisions/branching (exclusive, inclusive, and complex), merging, forking, and joining. Here we are only interested in Exclusive and Parallel Gateways. We represent them as shown in Figure 4.4

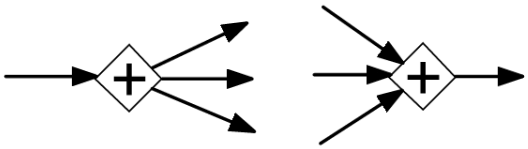
Exclusive Gateway

A diverging Exclusive Gateway is used to create alternative paths within a Process flow. This is basically the *diversion point in the road* for a Process. For a given instance of the Process, only one of the paths can be taken. A Decision can be thought of as a question that is asked at a particular point in the Process. The question has a defined set of alternative answers. Each answer is associated with a condition Expression that is associated with a Gateway's outgoing Sequence Flows. We also refer to this kind of exclusive gateway as *exclusive gateway multiple outputs*. We represent them as shown in Figure 4.5(a).

A converging Exclusive Gateway is used to merge alternative paths. We also refer to this gateway as *exclusive gateway multiple input*. We represent them as shown in Figure 4.5(b). Each incoming Sequence Flow token is routed to the outgoing Sequence Flow without synchronization.

Parallel Gateway

A Parallel Gateway is used to synchronize parallel flows and to create parallel flows. A Parallel Gateway creates parallel paths without checking any conditions. Each outgoing Sequence Flow receives a token upon execution of this Gateway. We represent them as shown in Figure 4.6(a). We also refers refer to this Gateway as *parallel gateway forking*.



(a) Parallel Gateway Forking

(b) Parallel Gateway Joining

Figure 4.6: Parallel gateways

For incoming flows, the Parallel Gateway will wait for all incoming flows before triggering the flow through its outgoing Sequence Flows. We represent them as shown in Figure 4.6(b). We also refer to this Gateway as *parallel gateway joining*.

5

From BPMN to ActionGUI

In this and the next chapter we present our methodology for developing secure business applications from Secure BPMN models. In a nutshell, our proposal is a three-step process. In the first step, ActionGUI models are automatically generated from the given secure BPMN model. These ActionGUI models specify the high-level behaviour of the desired business application, based on the information contained in the given BPMN model. This first step is indeed the cornerstone of our proposal. It is grounded on a novel mapping from Secure BPMN models to ActionGUI models. We define here this mapping, in 5.1 and 5.2, for BPMN models without SoD and BoD requirements, and we extend it, in the next section, to cover as well the aforementioned requirements. Then, in the second step, the modeler will complete the generated ActionGUI models with all the relevant information about the desired business application that can not be expressed in the given BPMN model. This second step is illustrated in Sections 5.3 and 5.4 with an example. Finally, in the third step, the desired business application will be automatically generated from the finalized models, using the available ActionGUI code generator. For the purpose of this work, this last step of our proposal is not particularly relevant. See [5] for further information.

In what follows, given a BPMN model $\mathcal{W}$, we denote by *bp2ag*($\mathcal{W}$) the ActionGUI model that corresponds to $\mathcal{W}$, according to our mapping. The model *bp2ag*($\mathcal{W}$) is the outcome of the first-step in our methodology. Also, given an ActionGUI model *bp2ag*($\mathcal{W}$), we denote its data model and its gui model, respectively, by *bp2ag_dm*($\mathcal{W}$) and *bp2ag_gm*($\mathcal{W}$). The security model of an ActionGUI model *bp2ag*($\mathcal{W}$) will play no role in our mapping.

5.1 Generating the business process data model

For any BPMN model $\mathcal{W}$, the (generic) business process data model *bp2ag_dm*($\mathcal{W}$) is shown in Figure 5.1, where $A_1,\ldots, A_n$ should be replaced by the names of the activities in $\mathcal{W}$. Basically, the enumerated entity *Activity* models the activities in $\mathcal{W}$. The entity *Task* models the instances of the activities in $\mathcal{W}$. Its attributes *status* and *type* model, respectively, the current status of the task and its activity type, while its association-ends *process* and *agent* model, respectively, the process to which the task belongs and the agent that executes the task. The entity *Process* models the instances of $\mathcal{W}$. Its attribute *id* models the process number. Finally, the entity *Agent* models the business process agents.

enumEntity Activity{
  $A_1, \ldots, A_n$
}
entity Task{
  String status
  Activity type
  Process process **oppositeTo** tasks
  Agent agent **oppositeTo** tasks
}
entity Process{
  Integer id
  Set(Task) tasks **oppositeTo** process
}
entity Agent {
  Set(Task) tasks **oppositeTo** agent
}

Figure 5.1: The (generic) model *bp2ag_dm*($\mathcal{W}$).

5.2 Generating the business process GUI model

For any BPMN model $\mathcal{W}$, the (generic) business process GUI model *bp2ag_gm*($\mathcal{W}$) consists of:

- A *task management window*, named TaskMng, where agents can create a new process and/or choose the next task to be executed.
- For each activity A in $\mathcal{W}$, a *task execution window*, named TaskOK_A, where agents can execute to completion a previously chosen task of type A.

5.2.1 The tasks management window

It contains two widgets: a button NewProcess, for creating a new process, and a table TaskList, for choosing among the pending tasks the next one to be executed.

The definition of the button NewProcess is given in Figure 5.2, where *first*($\mathcal{W}$) should be replaced by the name of the first activity in $\mathcal{W}$. Basically,

- The button is labeled 'New'.
- When the button is clicked-on, a new instance newProc of the entity Process is created. Also, a new instance newTask of the entity Task is created, with its attributes status and type set, respectively, to 'Pending' and *first*($\mathcal{W}$). Finally, the newly created task is linked to the newProc.

The definition of the table TaskList is given in Figure 5.3, where $A_1, \ldots, A_n$ should be replaced by the names of the activities in $\mathcal{W}$. Basically,

- The rows in the table TaskList are the pending tasks.

```
Button NewProcess{
   String text := 'New'
   Event onClick{
      newProc := new Process
      newTask := new Task
      newTask.status := 'Pending'
      newTask.process += newProc
      newTask.type := first(W)
   }
}
```

Figure 5.2: The (generic) model *bp2ag_gm*($\mathcal{W}$). The button NewProcess.

- There are four columns in the table TaskList. For each row,
  - the first column shows the number of the process to which the row, i.e., the task associated to this row, belongs;
  - the second column shows the type of the row;
  - the third column shows the status of the row; and,
  - the fourth column contains a button, named Run.
- For each row, when the button Run is clicked-on,
  - It changes the status of the row to 'Running';
  - It links the row to the instance of the entity Agent that models the caller, i.e., the current user.
  - If the row is an instance of an activity A_i, for $i = 1, \ldots, n$, it opens the window TaskOK_A_i, and set its variable selTask (whose role is to hold the task to be completed in this window) to row.

5.2.2 The task execution windows

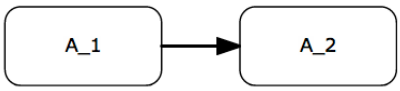
Each window TaskOK_A simply contains a button, named OK. When this button is clicked-on, the state of the current process is changed depending on what follows the activity A in $\mathcal{W}$. Logically, for each of the possible follow-ups, there is a different definition of the button OK. Also, each window TaskOK_A owns a variable selTask, which holds the task of type A to be completed. In the remaining of this section, we specify, for a selection of possible follow-ups, the actions triggered by the on-click event of the button OK

Simple flow Its definition is given in Figure 5.4, where A_1 and A_2 should be replaced by the corresponding activities in $\mathcal{W}$. Basically, when the button OK is clicked-on:

- It changes the status of selTask to 'Completed'.
- It creates a new instance newTask of the entity Task, with its attribute status set to 'Pending' and its attribute activity set to A_2. Also, it links newTask to the same process to which selTask belongs.

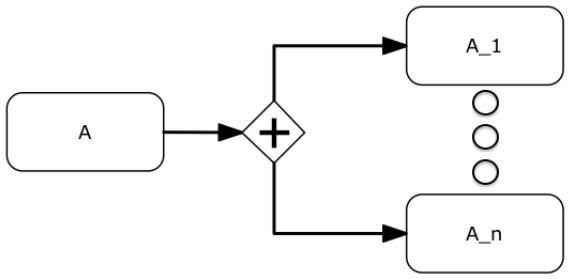
```
Table TaskList{
   Sequence(Task) rows :=
      Task.allInstances()−>select(t|t.status = 'Pending')
   Label Process{
      String text := row.process.id
   }
   Label Activity{
      String text := row.type
   }
   Label Status{
      String text := row.status
   }
   Button Run{
      Event onClick{
         row.status := 'Running'
         row.agent += caller
         if (row.type = A_1) {
            open TaskOK_A_1  (selTask := row)
         }
         ⋮
         if (row.type = A_n) {
            open TaskOK_A_n  (selTask := row)
         } } }
```

Figure 5.3: The (generic) model *bp2ag_gm*($\mathcal{W}$). The table TaskList.




```
Button Ok{
   String text := 'OK'
   Event onClick {
      selTask.status := 'Completed'
      newTask := new Task
      newTask.status := 'Pending'
      newTask.type := A_2
      newTask.process += selTask.process
   }
}
```

Figure 5.4: The (generic) model *bp2ag_gm*($\mathcal{W}$). The button OK (simple flow).



Button Ok{
  String text := 'OK'
  Event onClick {
    selTask.status := 'Completed'
    newTask_1 := new Task
    newTask_1.status := 'Pending'
    newTask_1.type := A_1
    newTask_1.process += selTask.process
    ⋮
    newTask_N := new Task
    newTask_N.status := 'Pending'
    newTask_N.type := A_n
    newTask_N.process += selTask.process
  }
}

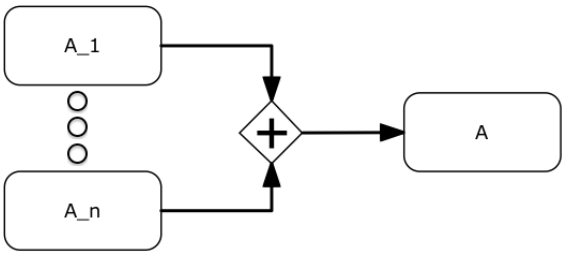
Figure 5.5: The (generic) model *bp2ag_gm*($\mathcal{W}$). The button OK (forking).

Forking Its definition is given in Figure 5.5, where $A_1, \ldots, A_n$ should be replaced by the corresponding activities in $\mathcal{W}$. Basically, when the button OK is clicked-on:

- It changes the status of the selTask to 'Completed'.
- For each activity $A_1, \ldots, A_n$, it creates a new instance newTask_i of the entity Task, with its attributes status and type set, respectively, to 'Pending' and A_i. Also, the newly created task is linked to the process to which selTask belongs.

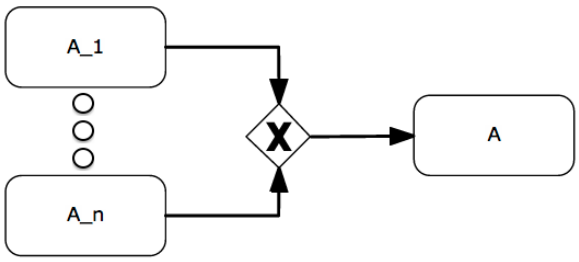
Joining Its definition is given in Figure 5.6, where $A_1, \ldots, A_n$ and A should be replaced by the corresponding activities in $\mathcal{W}$. Basically, when the button OK is clicked-on:

- It changes the status of selTask to 'Completed'.
- If $A_1, \ldots, A_n$ are completed, then it creates a new instance newTask of the entity Task, with its attribute status set to 'Pending' and its attribute activity set to A. Also, it links newTask to the same process to which selTask belongs.




```
Button Ok{
   String text := 'OK'
   Event onClick {
      selTask.status := 'Completed'
      if (Set{A_1, ..., A_n}->forAll(a|
            selTask.process.tasks->exists(t|
              t.type=a and t.status='Completed')))
          {
            newTask := new Task
            newTask.status := 'Pending'
            newTask.type := A
            newTask.process += selTask.process
          }
      }
   }
}
```

Figure 5.6: The (generic) model *bp2ag_gm*($\mathcal{W}$). The button OK (joining).



Button Ok{
  String text := 'OK'
  Event onClick {
    selTask.status := 'Completed'
    newTask := new Task
    newTask.status := 'Pending'
    newTask.type := A
    newTask.process += selTask.process
  }
}

Figure 5.7: The (generic) model *bp2ag_gm*($\mathcal{W}$). The button OK (exclusive gateway multiple input).

Exclusive gateway multiple input Its definition is given Figure 5.7, where A should be replaced by the corresponding activity in $\mathcal{W}$. Basically, when the button OK is clicked-on:

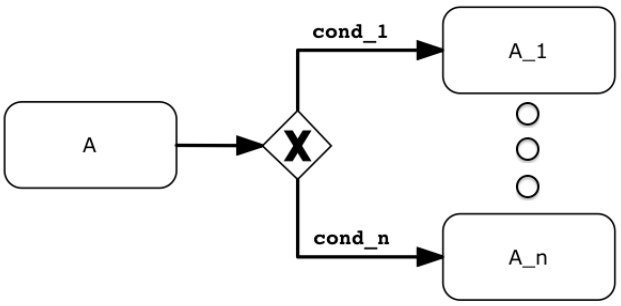
- It changes the status of selTask to 'Completed'.
- It creates a new instance newTask of the entity Task, with its attribute status set to 'Pending' and its attribute activity set to A. Also, it links newTask to the same process to which selTask belongs.

Exclusive gateway multiple output Its definition is given in Figure 5.8, where $A_1, \ldots, A_n$ and A should be replaced by the corresponding activities in $\mathcal{W}$. Basically, when the button OK is clicked-on:

- It changes the status of selTask to 'Completed'.
- It evaluates the conditions from top to bottom, and the body of the first condition that evaluates to true is performed, and the rest are skipped.
- Thus, if the $cond_i$ evaluates to true, the model creates a new instance newTask of the entity Task, with its attribute status set to 'Pending' and its attribute activity set to A_i. Also, it links newTask to the same process to which selTask belongs.

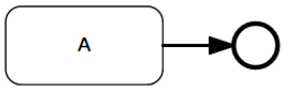
Ending Its definition is given in Figure 5.9. Basically, when the button OK is clicked-on:

- It changes the status of selTask to 'Completed'.



Button Ok{
  String text := 'OK'
  Event onClick {
    selTask.status := 'Completed'
    if($cond_1$){
      newTask := new Task
      newTask.status := 'Pending'
      newTask.type := A_1
      newTask.process += selTask.process
    }
    ⋮
    else if ($cond_n$){
      newTask := new Task
      newTask.status := 'Pending'
      newTask.type := A_n
      newTask.process += selTask.process
    }
  }
}

Figure 5.8: The (generic) model *bp2ag_gm*($\mathcal{W}$). The button OK (exclusive gateway multiple output).




```
Button Ok{
    String text := 'OK'
    Event onClick {
        selTask.status := 'Completed'
    }
}
```

Figure 5.9: The (generic) model *bp2ag_gm*($\mathcal{W}$). The button OK (ending).

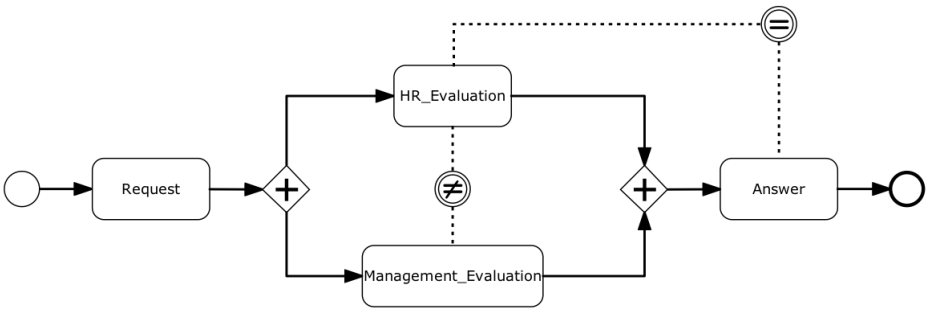


Figure 5.10: Holidays: a Secure BPMN model for processing holiday requests.

5.3 Completing the business process data model

In Figure 5.10 we show a simple Secure BPMN model, name Holidays. It specifies how to process holiday requests. The process starts with the activity Request (a request is submitted by an employee) and it ends with the activity Answer (the request is answered). Interestingly, before the activity Answer is executed, two other activities need to be completed, possibly in parallel: namely, HR_Evaluation (a staff member of the HR department evaluates the request) and Management_Evaluation (a manager evaluates the request). It also specifies two security requirements: (i) an SoD requirement, namely, that the HR staff member and the manager that evaluate a request must not be the same person; and (ii) a BoD requirement, namely, that the person that answers a request must be the same HR staff member than evaluates the request.

Recall that the generic business process data model does not fully specify the domain of the business process Holidays. Within our methodology, this should be done by the modeler by extending the generic business process data model. To illustrate this step, we show in Figure 5.11 an extension of *bp2ag_dm*(Holidays) that (partially) models the domain of the business process Holidays. In particular, it includes an entity *Request* that models the holiday requests. Its attribute *days* models the total number of days that are requested, while its association-ends *process* models the process to which the requests belongs.

```
entity Request{
  Integer days
  Boolean approvedHR
  Boolean approvedManager
  Process process oppositeTo request
}
entity Process{
  Request request oppositeTo process
}
entity Employee{
  String name
  Agent asAgent oppositeTo asEmployee
}
entity Agent{
  Employee asEmployee oppositeTo asAgent
}
```

Figure 5.11: Completing the data model *bp2ag_dm*(Holidays).

5.4 Completing the business process GUI model

Recall that the generic business process GUI model does not specify how a task is to be completed: e.g., which inputs (if any) the agent has to provide, how the agent is to provide these inputs, which properties (if any) these inputs need to satisfy, and so on. Within our methodology, all these details (which are, however, very relevant for the correctness and usability of a business application) are specified by the modeler by extending the definitions of the task execution windows in the generic business process GUI model.

To illustrate this step we show in Figure 5.12 an extension of the definition of the window TaskOK_Request that take into consideration that (i) we have modeled holiday requests using the entity Request, and that (ii) all requests have an attribute days that indicates the number of days which are requested. In particular, the window TaskOK_Request includes a text field Days, for writing the number of days which are request. Now, when the button OK is clicked-on (in addition to the expected actions, given that what follows is a parallel gateway forking):

- It creates a new instance newRequest of the entity Request and set its attribute days to the number written in the text field Days. Also, it links the newly created request to the process to which selTask belongs.

```
Window TaskOK_Request{
    Task selTask
    Label Number{
        String text := 'Total: '
    }
    TextEntry Days
    Button OK{
        String text := 'OK'
        Event onClick{
            newRequest := new Request
            newRequest.process += selTask.process
            newRequest.days := Days.text.toInteger()

            newTask_1 := new Task
            newTask_1.status := 'Pending'
            newTask_1.type := HR_Evaluation
            newTask_1.process += selTask.process
            newTask_2 := new Task
            newTask_2.status := 'Pending'
            newTask_2.type := Management_Evaluation
            newTask_2.process += selTask.process
        }
    }
}
```

Figure 5.12: Completing the GUI model *bp2ag_dm*(Holidays). The window TaskOK_Request (2 step).

6

From Secure BPMN to ActionGUI

The only difference between a BPMN model and a Secure BPMN model is that the latter specifies also two symmetric relations, representing separation of duties (SoD) and binding of duties (BoD) policies, on the business process $\mathcal{W}$

$$\mathrm{SoD}_{\mathcal{W}}, \mathrm{BoD}_{\mathcal{W}} \subset \mathrm{Activity}_{\mathcal{W}} \times \mathrm{Activity}_{\mathcal{W}}$$

where $\mathrm{Activity}_{\mathcal{W}}$ are the activities in $\mathcal{W}$. These relations impose the following constraints on $\mathcal{W}$:

- Let p be a $\mathcal{W}$-process and let A, A' be a pair of activities in $\mathcal{W}$ such that $(A, A') \in \mathrm{SoD}_{\mathcal{W}}$. Then, if an agent has performed a task of type A in p, then he/she must not perform the corresponding task of type A' in p.
- Let p be a $\mathcal{W}$-process and let A, A' be a pair of activities in $\mathcal{W}$ such that $(A, A') \in \mathrm{BoD}_{\mathcal{W}}$. Then, if an agent has performed a task of type A in p, then he/she must perform the corresponding task of type A' in p.

To capture SoD and BoD requirements in our mapping from BPMN models to ActionGUI model we only need to modify the definition of the button Run in the table TaskList in the *task management window* TaskMng. In Figure 6.1 we show our definition of the SoD and BoD security-aware table TaskList, where, $A_1, \ldots, A_n$ should be replaced by the names of the activities in $\mathcal{W}$, and, for $1 \leq i \leq n$, $\mathrm{SoD}_{\mathcal{W}}(A_i)$ and $\mathrm{BoD}_{\mathcal{W}}(A_i)$ should be replaced by the following sets:

$$\begin{aligned} \mathrm{SoD}_{\mathcal{W}}(A_i) &= \{A' \mid \mathrm{SoD}_{\mathcal{W}}(A_i, A')\} \\ \mathrm{BoD}_{\mathcal{W}}(A_i) &= \{A' \mid \mathrm{BoD}_{\mathcal{W}}(A_i, A')\} \end{aligned}$$

According to this new definition, when the button Run is clicked-on, the status of the row will be changed to 'Running' (and, subsequently, the corresponding window TaskOK will be opened), only if the applicable SoD and BoD constraints are satisfied, taking into consideration:

```
Button Run{
  Event onClick{
    if(row.type = A1) {
      if (caller.tasks
            ->select(t|t.process = row.process)
            ->forAll(t| SoD_W(A1)->excludes(t.type))
          and
            row.process.tasks
              ->select(t| BoD_W(A1)->includes(t.type))
              ->forAll(t|t.agent = caller}))
          {
            row.status := 'Running'
            row.agent += caller
            open TaskOK_A1( selTask := row)
          }
      }
    ⋮
    if(row.type = An) {
      if (caller.tasks
            ->select(t|t.process = row.process)
            ->forAll(t| SoD_W(An)->excludes(t.type))
          and
            row.process.tasks
              ->select(t| BoD_W(An)->includes(t.type))
              ->forAll(t|t.agent = caller))
          {
            row.status := 'Running'
            row.agent += caller
            open TaskOK_An( selTask := row)
          }
    }
  }
}
```

Figure 6.1: The (generic) model *bp2ag_gm*($\mathcal{W}$). The SoD and BoD security-aware table `TaskList`.

(i) the type of the row (i.e., of the task whose status is to be changed) and (ii) the tasks that the caller owns within the process to which rows belongs.

To illustrate this extension of our mapping, we apply it to our running example: namely, the business process Holidays. According to the Secure BPMN model shown in Figure 5.10, these are the SoD and BoD relations restricting the business process Holidays:

$$\begin{aligned} \mathrm{SoD}_{\mathrm{Holidays}} &= \{(\mathrm{HR\_Evaluation}, \mathrm{Management\_Evaluation})\} \\ \mathrm{BoD}_{\mathrm{Holidays}} &= \{(\mathrm{HR\_Evaluation}, \mathrm{Answer})\} \end{aligned}$$

In the Figure 6.2 the table TaskList of the model *bp2ag_gm*(*Holiday*) is showed. Some simplifications were made for the sake of simplicity, based on the following facts:

- forAll(t|Set{}−>excludes(t.type))= true.
- select(t|Set{}−>includes(t.type))−>forAll(t|t.agent = caller)= true

```
Button Run{
  Event onClick{
    if(row.type = HR_Evaluation) {
      if (caller.tasks
            ->select(t|t.process = row.process)
            ->forAll(t|Set{Management_Evaluation})->excludes(t.type)
          and
            row.process.tasks
            ->select(t|Set{Answer})->includes(t.type)
            ->forAll(t|t.agent = caller))
      {
        row.status := 'Running'
        row.agent += caller
        open TaskOK_HR_Evaluation( selTask := row)
      }
    }
    if(row.type = Management_Evaluation) {
      if (caller.tasks
            ->select(t|t.process = row.process)
            ->forAll(t|Set{HR_Evaluation})->excludes(t.type))
      {
        row.status := 'Running'
        row.agent += caller
        open TaskOK_Management_Evaluation( selTask := row)
      }
    }
    if(row.type = Answer) {
      if (row.process.tasks
            ->select(t|Set{HR_Evaluation})->includes(t.type)
            ->forAll(t|t.agent = caller))
      {
        row.status := 'Running'
        row.agent += caller
        open TaskOK_Answer( selTask := row)
      }
    }
    if(row.type = Request)
    {
      row.status := 'Running'
      row.agent += caller
      open TaskOK_Request( selTask := row)
    }
  }
}
```

Figure 6.2: The model *bp2ag_gm*(Holiday). The SoD and BoD security-aware table `TaskList`.

7

Tooling support

In this section we show the steps involved in the transformation of a Secure BPMN model into its corresponding Secure Business Application. For each step in the transformation we describe the procedure used, the input required, the output produced, and expose the existent drawbacks along with their possible solutions.

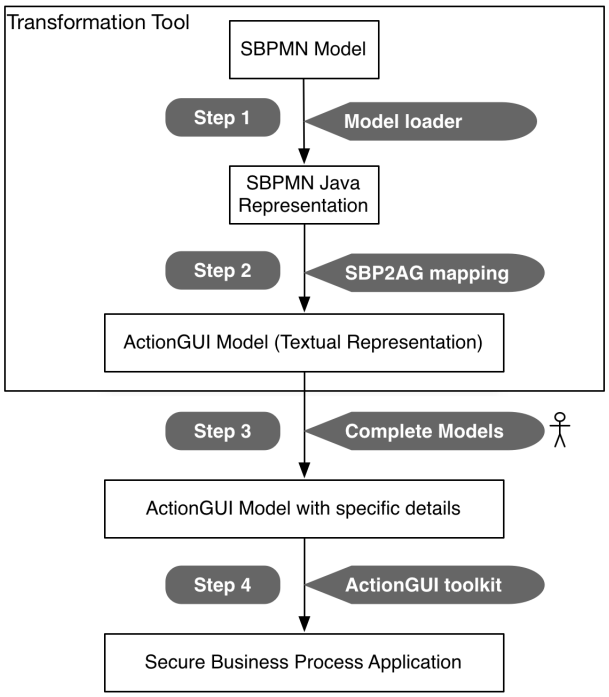


Figure 7.1: Steps performed to obtain the Secure Business Process Application

In Figure 7.1 we show a diagram that has a chain of rectangular shaped boxes. From box to box, the Secure BPMN model is transformed and completed. Between each box there are two labels, the one on the left is just to enumerate the steps, while the one on the right is the

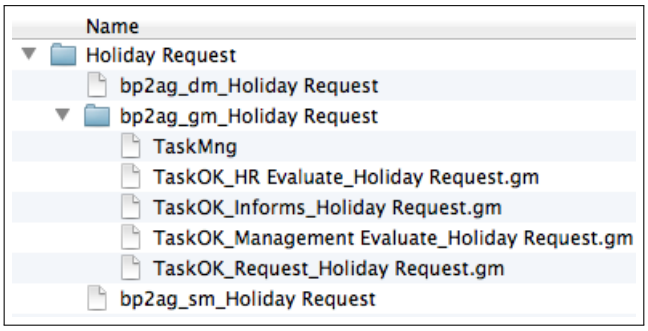


Figure 7.2: Files generated by SBP2AG Mapping (Holidays)

process used in each transformation step. Some of the steps also have a person on its right, that means the step requires manual intervention. The step that does not have a person on the right are performed automatically. The diagram also has a square box named *Transformation Tool*, that is a tool that we have developed as part of our work. It is composed of the *Model Loader* and *SBP2AG Mapping*.

Step 1: Model Loader. The Secure BPMN model is loaded into its java representation. The BPMN standard, provides the standard XML representation for BPMN graphical models. We use Signavio to obtain the XML representation of a BPMN graphical model. Signavio automatically transforms the graphical business process into the XML representation and vice versa. We have developed a parser to load the XML representation of a BPMN model into its Java representation (limited to BPMN models that we are interested in)

Step 2: SBP2AG Mapping. It automatically performs the map from the Secure BPMN model's java representation to the corresponding ActionGUI model. Given a Secure BPM model $\mathcal{W}$, named $name_{\mathcal{W}}$, which contains the activities $a_1, \ldots, a_n$. The tool generates a folder named $name_{\mathcal{W}}$, as the one shown in Figure 7.2. This folder contains:

- A file named $bp2ag\_dm_{\mathcal{W}}$ that has the data model.
- A folder named $bp2ag\_ag_{\mathcal{W}}$ containing:
  - A file named $TaskMng$ which has the *TaskMng* window model. This model includes all the SoD and BoD constraints defined on $\mathcal{W}$
  - For each a_i in $a_1, \ldots, a_n$, a file named $TaskOK\_a_{i\mathcal{W}}$ contains the execution window that models the completion window for each activity in $\mathcal{W}$.

Step 3: Complete the GUI-models. The ActionGUI model generated in the previous step is completed by the modeler.

Step 4: ActionGUI toolkit It generates a web application ready to be deployed. This is automatically done by the ActionGUI toolkit.

8

Conclusions and future work

In this work we have proposed a methodology for developing secure business applications from Secure BPMN models. To the best of our knowledge, the current BPMN business application development tools do not provide support for Secure BPMN models. In addition, in our methodology, the many details about a business application that can not be expressed in a BPMN model are specified by the modeler using ActionGUI, saving him/her the hassle of having to manually encode them, as it is typically the case in the available BPMN business application development tools.

We plan to extend our mapping to cover all the standard BPMN language. Dealing with cycles, in particular, may introduce some additional complexity to our mapping when combined with SoD or BoD requirements. Finally, we plan to extend Secure BPMN by allowing to further precise the SoD and BoD requirements using OCL expressions (which will play a role analogous to the so-called authorization constraints in SecureUML). By doing so, we expect to be able to security requirements like the following: "Since John and Jack are brothers, the should not be able to approve each other's orders."

Bibliography

[1] Activiti. BPM modeling tool and BPM execution tool: Activiti, version 5.9,http://www.activiti.org/, March 2012.

[2] D. Basin, S. J. Burri, and G. Karjoth. Separation of duties as a service. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 423–429, New York, NY, USA, 2011. ACM.

[3] D. Basin, M. Clavel, J. Doser, and M. Egea. A metamodel-based approach for analyzing security-design models. In G. Engels, B. Opdyke, D. Schmidt, and F. Weil, editors, *Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems (MODELS '07)*, volume 4735 of *LNCS*, pages 420–435. Springer-Verlag, 2007.

[4] D. Basin, M. Clavel, J. Doser, and M. Egea. Automated analysis of security-design models. *Information and Software Technology*, 51(5):815–831, 2009.

[5] D. Basin, M. Clavel, and M. Egea. A decade of model-driven security. In *Proceedings of the 16th ACM symposium on Access control models and technologies*, SACMAT '11, pages 1–10, New York, NY, USA, 2011. ACM.

[6] D. Basin, M. Clavel, M. Egea, and M. Schläpfer. Automatic generation of smart, security-aware GUI models. In F. Massacci, D. S. Wallach, and N. Zannone, editors, *Proceedings of the 2nd International Symposium on Engineering Secure Software and Systems (ESSoS'10)*, volume 5965 of *LNCS*, pages 201–217, Pisa, Italy, 2010. Springer.

[7] D. Basin, J. Doser, and T. Lodderstedt. Model driven security: From UML models to access control infrastructures. *ACM Transactions on Software Engineering and Methodology*, 15(1):39–91, 2006.

[8] D. A. Basin, S. J. Burri, and G. Karjoth. Obstruction-free authorization enforcement: Aligning security with business objectives. In *CSF*, pages 99–113. IEEE Computer Society, 2011.

[9] D. A. Basin, M. Clavel, M. Egea, M. A. G. de Dios, C. Dania, G. Ortiz, and J. Valdazo. Model-driven development of security-aware GUIs for data-centric applications. In A. Aldini and R. Gorrieri, editors, *FOSAD*, volume 6858 of *Lecture Notes in Computer Science*, pages 101–124. Springer, 2011.

[10] D. D. Clark and D. R. Wilson. A comparison of commercial and military computer security policies. *Security and Privacy, IEEE Symposium on*, 0:184, 1987.

[11] M. Clavel, V. Silva, C. Braga, and M. Egea. Model-driven security in practice: An industrial experience. In I. Schieferdecker and A. Hartman, editors, *Proceedings of 4th European Conference on Model Driven Architecture-Foundations and Applications (ECMDA-FA '08) - Industrial Track*, volume 5095 of *LNCS*, pages 327–338, Berlin-Germany, 2008. Springer-Verlag.

[12] B. Curtis, M. I. Kellner, and J. Over. Process modeling. *Commun. ACM*, 35(9):75–90, 1992.

[13] M. A. G. de Dios, C. Dania, M. Schläpfer, D. Basin, M. Clavel, and M. Egea. SSG: A model-based development environment for smart, security-aware GUIs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, volume 2, pages 311–312, Cape Town-South Africa, 2010. ACM.

[14] Ernst and Young. European fraud survey 2011, 2011.

[15] D. F. Ferraiolo, R. S. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security*, 4(3):224–274, 2001.

[16] jBPM. BPM modeling tool and BPM execution tool: jBPM, version 5.2,http://www.jboss.org/jbpm, December 2011.

[17] T. Lodderstedt, D. Basin, and J. Doser. SecureUML: A UML-based modeling language for model-driven security. In J.-M. Jézéquel, H. Hussmann, and S. Cook, editors, *Proceedings of the 5th international Conference on the Unified Modeling Language: Model Engineering, Concepts, and Tools (UML'02)*, volume 2460 of *LNCS*, pages 426–441. Springer-Verlag, 2002.

[18] Object Management Group. *Object Constraint Language specification Version 2.2*, February 2010. OMG document available at `http://www.omg.org/spec/OCL/2.2`.

[19] Object Management Group. Business process model and notation (BPMN), version 2.0,. OMG Standard, http://www.omg.org/spec/BPMN/2.0/PDF, January 2011.

[20] Object Management Group. Unified modeling language (UML). OMG Standard, August 2011.