ActionGUI semantics

David Basin[1], Manuel Clavel[2,3], Marina Egea[4], Miguel A. García de Dios[2], and Carolina Dania[2,3]

[1] ETH Zürich, Switzerland
basin@inf.ethz.ch
[2] IMDEA Software Institute, Madrid, Spain
[manuel.clavel,miguelangel.garcia,carolina.dania]@imdea.org
[3] Universidad Complutense, Madrid, Spain
[4] Atos Research & Innovation Dept., Madrid, Spain
marina.egea@atosresearch.eu

Abstract. In this technical report we provide the formal account of our ActionGUI methodology, including the semantics of the modeling languages that we use, the definition of our many-models-to-models transformation, and the proof of its correctness.

1 ComponentUML

In this section we first define ComponentUML *data models*. Then, given a ComponentUML data model D, we define *D-object models*. Finally, we define the *semantics* of a ComponentUML data model D as the set $\mathrm{Sem}(D)$ of all the D-object models.

Notation. Let $\mathrm{TP} = \{\mathsf{Integer}, \mathsf{Real}, \mathsf{String}, \mathsf{Boolean}\}$ be the set of ComponentUML primitive data types. In what follows, we denote by $[\![t]\!]^{\mathrm{TP}}$ the standard carrier set of t, for each primitive data type $t \in \mathrm{TP}$, e.g., $[\![\mathsf{Integer}]\!]^{\mathrm{TP}} = \mathbb{Z}$.

Let $\mathcal{A} \subset [\![\mathsf{String}]\!]^{\mathrm{TP}}$ be the set of all finite strings that only contain letters of the English alphabet.

1.1 ComponentUML data models

Definition. A ComponentUML *data model* is a tuple $\langle C, AT, AS, ASO\rangle$ such that:

- $C \subset \mathcal{A}$ is a set of class identifiers.
- AT is a set of triples $\langle at, c, t\rangle$, also represented as $at_{(c,t)}$, where $at \in \mathcal{A}$ is an attribute identifier, $c \in C$, $t \in C \cup \mathrm{TP}$, and c and t are, respectively, the class and the type of the attribute at.
- AS is a set of triples $\langle as, c, c'\rangle$, also denoted as $as_{(c,c')}$, where $as \in \mathcal{A}$ is an association-end identifier, $c, c' \in C$, and c and c' are, respectively, the source and the target classes of as.
- ASO is a symmetric relation, $ASO \subseteq AS \times AS$, where $(as_{(c,c')}, as'_{(c',c)}) \in ASO$ represents that as' is the association-end opposite to as, and vice versa, and $c, c' \in C$.

Invariants.

- There is no class whose identifier also belongs to TP.
- Attributes and associations-ends of the same class always have different identifiers.
- Every association-end is related with exactly another association-end. That is, for every tuple $\langle as, c, c' \rangle$ in AS, there exists exactly one other tuple $\langle as', c', c \rangle$ in AS such that $(\langle as, c, c' \rangle, \langle as', c', c \rangle)$ in ASO.

1.2 ComponentUML object models

Definition. Let D be a ComponentUML data model $\langle C, AT, AS, ASO \rangle$. Then, a *$D$-object model* is a tuple $\langle O, VA, LK \rangle$, such that:

- O is a set of pairs $\langle o, c \rangle$, where $o \in \mathcal{A}$ is an object identifier and $c \in C$. Each pair $\langle o, c \rangle$, also represented as o_c, denotes that the object o is of the class c.
- VA is a set of triples $\langle o_c, at_{(c,t)}, va \rangle$, where $at_{(c,t)} \in AT$, $o_c \in O$, $t \in \mathrm{TP}$, and $va \in [\![t]\!]^{\mathrm{TP}}$ is a value of type t. Each triple $\langle o_c, at_{(c,t)}, va \rangle$ denotes that va is the value of the attribute at of the object o.
- LK is a set of triples $\langle o_c, as_{(c,c')}, o'_{c'} \rangle$, where $as_{(c,c')} \in AS$, and $o_c, o'_{c'} \in O$. Each tuple $\langle o_c, as_{(c,c')}, o'_{c'} \rangle$ denotes that the object o' is among the objects that are linked to the object o through the association-end as.

Invariants.

- There are no two different values for the same attribute of the same object. (However, it is not necessary that every attribute of an object has a value.)
- For every association-end $as_{(c,c')}$ in AS, such that $(as_{(c,c')}, as'_{(c',c)})$ in ASO, if there is a link $\langle o_c, as_{(c,c')}, o'_{c'} \rangle \in LK$ between two objects o_c and $o'_{c'}$ through this association-end, then there is also a link $\langle o'_{c'}, as'_{(c',c)}, o_c \rangle \in LK$ between these two objects through the opposite association-end.

1.3 Semantics of ComponentUML data models

Definition. Let D be a ComponentUML data model. The *semantics* of D, denoted by $\mathrm{Sem}(D)$, is the set of all the ComponentUML D-object models.

2 SecureUML

In this section we first define SecureUML *data actions* and SecureUML *authorization constraints*, both relative to a given ComponentUML data model. Then, we define SecureUML *security models*, also relative to a given ComponentUML model. Next, given a SecureUML security model S, we define *S-authorized actions*. Finally, we define the *semantics* of a SecureUML security model as the set $\mathrm{Sem}(S)$ of all the S-authorized actions and, based on this definition, we define the notion of a *consistent* SecureUML security model.

Notation. Let D be a data model, and let I be a D-object model, $I = \langle O, VA, LK \rangle$. In what follows, we will use the following notation:

- We denote by $\mathrm{Typ}(D)$ the set of all the OCL types, given the classes declared in D. These types are defined in the OCL standard [1].
- We denote by $\mathrm{Expr}(D)$ the set of all the OCL expressions that have D as their contextual model. These expressions are defined in the OCL standard [1]. Note that, by definition, they do not contain free variables.
- Let X be a set of pairs $\langle x, t \rangle$, also written as x_t, where $x \in \mathcal{A}$ is a variable identifier of type $t \in \mathrm{Typ}(D)$. Then, we denote by $\mathrm{Expr}(D, X)$ the set of all the OCL expressions that have D as their contextual model but that may also contain variables in X as free variables. Moreover, for every $expr \in \mathrm{Expr}(D, X)$, we denote by $\mathrm{FVar}(expr) \subseteq X$ the set of all the free variables contained in $expr$.
- We denote by $\mathrm{Expr}(D_I)$ the set of all the OCL expressions that have D as their contextual model and may also contain as constants the objects $o_c \in O$.
- Let $expr$ be an OCL expression in $\mathrm{Expr}(D_I)$. Then, we denote by $[\![expr]\!]^I$ the evaluation of the expression $expr$ in the object model I, as defined in the OCL standard [1]. Note that the evaluation of an OCL expression always return a *literal expression* in $Expr(D_I)$, which can not be further reduced and which does not contain any variables.
- Let X be a set of variables x_t, where $t \in \mathrm{Typ}(D)$. Then, a (X, I)-*substitution* θ is a function, $\theta : X \to \mathrm{Expr}(D_I)$, that assigns to each variable in X an expression in $\mathrm{Expr}(D_I)$ of the appropriate type.
  Now, let $x_t \in X$ be a variable and let $expr$ in $\mathrm{Expr}(D_I)$ be an expression of type t. Then, we denote by $\theta \oplus \{x_t \mapsto expr\}$, $\theta \oplus \{x_t \mapsto expr\} : X \to \mathrm{Expr}(D_I)$, the *overriding* of θ by $\{x_t \mapsto expr\}$. That is, $\theta \oplus \{x_t \mapsto expr\}(x_t) = expr$, but, for every other $x'_{t'} \in X$, $x_t \neq x'_{t'}$, $\theta \oplus \{x_t \mapsto expr\}(x'_{t'}) = \theta(x'_{t'})$.
  Moreover, for every (X, I)-*substitution* θ, we denote as $\hat{\theta}$ the homomorphic extension of θ over the set $\mathrm{Expr}(D, X)$. Finally, for $expr \in \mathrm{Expr}(D, X)$, we write $\hat{\theta}(expr)$ as $(expr)\theta$.

2.1 SecureUML data actions

Definition. Let D be a data model $\langle C, AT, AS, ASO \rangle$. Then, we denote by $\mathrm{Act}(D)$ the set of all the (atomic) *data actions* that can be executed on D-object models. $\mathrm{Act}(D)$ is defined as follows: for every class $c \in C$, every attribute $at_{(c,t)} \in AT$, and every association-end $as_{(c,c')} \in AS$,

$$\mathsf{Create}(c), \mathsf{Delete}(c) \in \mathrm{Act}(D).$$
$$\mathsf{Read}(at_{(c,t)}), \mathsf{Update}(at_{(c,t)}) \in \mathrm{Act}(D).$$
$$\mathsf{Read}(as_{(c,c')}), \mathsf{Create}(as_{(c,c')}), \mathsf{Delete}(as_{(c,c')}) \in \mathrm{Act}(D).$$

The notation $action(resource)$ reflects that data actions are always upon resources.

2.2 SecureUML authorization constraints

Definition. Let D be a data model $\langle C, AT, AS, ASO\rangle$. Let $u \in C$ be a class that represents the *users*. Let $act \in Act(D)$ be a data action. Then, we denote by $\mathrm{AuthExpr}(D, u, act)$ the set of all the *authorization constraints* that can be imposed on users of type u for executing the data action act with respect to the data model D. Informally, an authorization constraint is an (extended) OCL expression that may contain distinguished keywords (logically interpreted as free variables) that refer to the user attempting to execute the action (caller), to the data upon which the action is to be executed (self), or to the data that the action takes as its arguments (value and target).

More formally, $\mathrm{AuthExpr}(D, u, act)$ is defined as follows: for every class $c \in C$, every attribute $at_{(c,t)} \in AT$, and every association-end $as_{(c,c')} \in AS$:

$$\mathrm{AuthExpr}(D, u, \mathsf{Create}(c)) = \mathrm{Expr}(D, \{\mathsf{caller}_u\}).$$
$$\mathrm{AuthExpr}(D, u, \mathsf{Delete}(c)) = \mathrm{Expr}(D, \{\mathsf{self}_c, \mathsf{caller}_u\}).$$
$$\mathrm{AuthExpr}(D, u, \mathsf{Read}(at_{(c,t)})) = \mathrm{Expr}(D, \{\mathsf{self}_c, \mathsf{caller}_u\}).$$
$$\mathrm{AuthExpr}(D, u, \mathsf{Update}(at_{(c,t)})) = \mathrm{Expr}(D, \{\mathsf{self}_c, \mathsf{value}_t, \mathsf{caller}_u\}).$$
$$\mathrm{AuthExpr}(D, u, \mathsf{Read}(as_{(c,c')})) = \mathrm{Expr}(D, \{\mathsf{self}_c, \mathsf{caller}_u\}).$$
$$\mathrm{AuthExpr}(D, u, \mathsf{Create}(as_{(c,c')})) = \mathrm{Expr}(D, \{\mathsf{self}_c, \mathsf{target}_{c'}, \mathsf{caller}_u\}).$$
$$\mathrm{AuthExpr}(D, u, \mathsf{Delete}(as_{(c,c')})) = \mathrm{Expr}(D, \{\mathsf{self}_c, \mathsf{target}_{c'}, \mathsf{caller}_u\}).$$

2.3 SecureUML security models

Definition. Let D be a data model $\langle C, AT, AS, ASO\rangle$. Then, a SecureUML *$D$-security model* S is a tuple $\langle D, R, RH, u, P\rangle$ such that:

- $R \subset \mathcal{A}$ is a set of role identifiers.
- $RH \subset R \times R$ is a partial order representing the role hierarchy.
- $u \in C$ is a class that represents the *users*.
- P is a set of triples $\langle r, act, expr\rangle$ representing *permissions*: namely, that the role $r \in R$ is granted permission for the action $act \in \mathrm{Act}(D)$ provided the constraint $expr \in \mathrm{AuthExpr}(D, u, act)$ is satisfied.

Definition. Let D be a data model $\langle C, AT, AS, ASO\rangle$. Let S be a D-security model $\langle D, R, RH, u, P\rangle$. Then, $\mathrm{AuthPerm}(S, r, act)$ is the disjunction of all the authorization constraints controlling the access for users in the role r to execute the action act, according to S. $\mathrm{AuthPerm}(S, r, act)$ is defined as follows: Let $Q = \{expr \mid \exists r' \in R.\, \langle r', act, expr\rangle \in P \wedge (r, r') \in RH\}$. Then,

$$\mathrm{AuthPerm}(S, r, act) = \begin{cases} expr_1 \text{ or } \ldots \text{ or } expr_n, & \text{if } Q = \{expr_1, \ldots, expr_n\}. \\ \mathsf{false}, & \text{if } Q = \emptyset. \end{cases}$$

Note that, by definition, $\mathrm{AuthPerm}(S, r, act) \in \mathrm{Expr}(D, X)$, where X is the set containing caller_u plus the appropriate instances of self, target, and value, depending on the type of the action act.

2.4 SecureUML authorized actions

Definition. Let D be a data model $\langle C, AT, AS, ASO\rangle$. Let S be a D-security model $\langle D, R, RH, u, P\rangle$. Let I be a D-object model $\langle O, VA, LK\rangle$. Let $o_u \in O$ be a user, $r \in R$ be a role, and $act \in \mathrm{Act}(D)$ be a D-data action. Moreover, let θ be a $(\mathrm{FVar}(\mathrm{AuthPerm}(S, r, act)), I)$-substitution.

Then, $\langle I, o_u, r, act, \theta\rangle$ is an *S-authorized action* if and only if

$$[\![\mathrm{AuthPerm}(S, r, act)(\theta \oplus \{\mathsf{caller}_u \mapsto o_u\})]\!]^I = \mathsf{true}.$$

Note that, given our definition of AuthPerm,

- No permission is granted for executing an action, unless it is explicitly declared.
- All permissions are inherited along the role hierarchy.

2.5 Semantics of SecureUML security models

Definition. Let D be a data model and let S be a D-security model. Then, the *semantics* of S, given by $\mathrm{Sem}(S)$, is the set of all the S-authorized actions.

3 GUIML

In this section we first define GUIML *layout models*, which simply model graphical user interfaces without considering their behaviors. Then, we define GUIML *statements*, which specify sequences of actions that are possibly conditional and iterated. Next, we define GUIML *behavioral models*, which are GUIML layout models but also with associated behavior, i.e., with statements associated to each of the widget events. Finally, we define a set of inference rules that will provide the (operational) *semantics* of GUIML *events* as the set $\mathrm{Sem}(G, ev)$ of all the *transitions* defined by these rules.

Notation. In what follows, let ET be the set of GUIML event types,

$$\mathrm{ET} = \{\mathsf{onClick}, \mathsf{onCreate}\}.$$

Also, let WT be the set of GUIML widget types,

$$\mathrm{WT} = \{\mathsf{Window}, \mathsf{Table}, \mathsf{Combo\text{-}box}, \mathsf{Button}, \mathsf{Text\ field}, \mathsf{Label}, \mathsf{Boolean\ check}\}.$$

3.1 GUIML layout models

Definition. A GUIML *layout model* H is a tuple $\langle W, WC, X, EV\rangle$ such that:

- W is a set of pairs $\langle w, wt\rangle$, also represented as w_{wt}, where $w \in \mathcal{A}$ is a widget identifier, and $wt \in \mathrm{WT}$ is the widget's type.

- $WC \subset W \times W$ is a relation representing the widget containment.
- X is a set of pairs $\langle\langle x, t\rangle, \langle w, wt\rangle\rangle$, called *widget variables*, also represented as $\langle x_t, w_{wt}\rangle$, where x is a variable identifier, $t \in \mathrm{Typ}(D)$ is the variable's type, and $\langle w, wt\rangle \in W$ is the widget that *owns* this variable.
- EV is a set of pairs $\langle ev, \langle w, wt\rangle\rangle$, also represented as $\langle ev, w_{wt}\rangle$, where $ev \in \mathrm{ET}$ is an event type and $\langle w, wt\rangle \in W$ is the widget that supports this event type.

Invariants.

- The containment relation WC defines set of rooted trees. Moreover, at the root of every tree in WC there is a widget of type Window and, conversely, every widget in W of type Window is the root of a tree in WC.
- There are no two variables owned by the same widget with the same identifier.
- If two widgets are directly contained in the same widget, then they have different identifiers.

Notation. Let H be a GUIML layout model $\langle W, WC, X, EV\rangle$. In what follows we will use the following notation:

- We denote by WC^+ the transitive closure of the containment relation defined in WC.
- Let $w_{wt} \in W$ be a widget in W, $wt \neq \mathsf{Window}$. Then, we denote by $\mathrm{Win}(H, w_{wt})$ the window that contains w_{wt} in W, i.e., $(w_{wt}, \mathrm{Win}(H, w_{wt})) \in WC^+$.
- Let $w_{wt} \in W$ be a widget in W. Then, we denote by $\mathrm{Var}(H, w_{wt})$ the set of variables in X that are owned by w_{wt}, i.e., $\mathrm{Var}(H, w_{wt}) = \{\langle x_t, w'_{wt'}\rangle \mid \langle x_t, w'_{wt'}\rangle \in X \wedge w_{wt} = w'_{wt'}\}$.
- Let $w_{wt} \in W$ be a widget in W. Then, we denote by $\mathrm{Var}^\sharp(H, w_{wt})$ the set of the variables in X that are *visible* from w_{wt}. $\mathrm{Var}^\sharp(H, w_{wt})$ is defined as follows:

$$\begin{aligned}\mathrm{Var}^\sharp(H, w_{wt}) = \mathrm{Var}(H, w_{wt}) \cup \\ \{\langle x_t, w'_{wt'}\rangle \mid \langle x_t, w'_{wt'}\rangle \in \mathrm{Var}(H, w'_{wt'}) \wedge \\ (w'_{wt'}, \mathrm{Win}(H, w_{wt})) \in WC^+\}.\end{aligned}$$

  Note that, by definition, if two widgets are contained in the same window, then their sets of visible variables are identical. Also, the set of visible variables of a widget is the same than the set of visible variables of the widget's containing window.

3.2 GUIML statements

Definition. Let D be a data model $\langle C, AT, AS, ASO\rangle$. Let H be a GUIML layout model $\langle W, WC, X, EV\rangle$. Let w_{wt} be a window in H, i.e., $w_{\mathsf{Window}} \in W$. Then, we denote by $\mathrm{Stm}(D, H, w)$ the set of all the *statements* that can be written in the context of the window w. This set is inductively defined as follows:

Base case (data actions): The building block for statements are the data actions along with the GUI actions. The GUIML data actions are the SecureUML data actions introduced before, except that they now take additional arguments that, depending on the action's type, either specify, using OCL (extended with widget variables), the object *self* upon which the action is to be executed, or the *value* and *target* of this action, or indicate the widget *variable* where the action's outcome is to be stored. To reflect this difference between the GUIML data actions and their corresponding SecureUML data actions, we use the notation $action(resource)[\boldsymbol{arguments}]$ for GUIML data actions. Thus, if $action(resource)[\boldsymbol{arguments}]$ is a GUIML data action, then $action(resource)$ is its corresponding SecureML data action.

- For every entity create action $\mathsf{Create}(c) \in \mathrm{Act}(D)$ and every *variable* of type c in $\mathrm{Var}^\sharp(H, w_{wt})$, then
$$\mathsf{Create}(c)[variable] \in \mathrm{Stm}(D, H, w).$$
- For every entity delete action $\mathsf{Delete}(c) \in \mathrm{Act}(D)$ and every expression *self* of type c in $\mathrm{Expr}(D, \mathrm{Var}^\sharp(H, w_{wt}))$, then
$$\mathsf{Delete}(c)[self] \in \mathrm{Stm}(D, H, w).$$
- For every attribute read action $\mathsf{Read}(at_{(c,t)}) \in \mathrm{Act}(D)$, every expression *self* of type c in $\mathrm{Expr}(D, \mathrm{Var}^\sharp(H, w_{wt}))$, and every widget *variable* of type t in $\mathrm{Var}^\sharp(H, w_{wt})$, then
$$\mathsf{Read}(at_{(c,t)})[self, variable] \in \mathrm{Stm}(D, H, w).$$
- For every attribute update action $\mathsf{Update}(at_{(c,t)}) \in \mathrm{Act}(D)$, every expression *self* of type c in $\mathrm{Expr}(D, \mathrm{Var}^\sharp(H, w_{wt}))$, and every expression *value* of type t in $\mathrm{Expr}(D, \mathrm{Var}^\sharp(H, w_{wt}))$, then
$$\mathsf{Update}(at_{(c,t)})[self, value] \in \mathrm{Stm}(D, H, w).$$
- For every association-end read action $\mathsf{Read}(as_{(c,c')}) \in \mathrm{Act}(D)$, every expression *self* of type c in $\mathrm{Expr}(D, \mathrm{Var}^\sharp(H, w_{wt}))$, and every *variable* of type $\mathrm{Set}(c')$ in $\mathrm{Var}^\sharp(H, w_{wt})$, then
$$\mathsf{Read}(as_{(c,c')})[self, variable] \in \mathrm{Stm}(D, H, w).$$
- For every association-end create action $\mathsf{Create}(as_{(c,c')}) \in \mathrm{Act}(D)$, every expression *self* of type c in $\mathrm{Expr}(D, \mathrm{Var}^\sharp(H, w_{wt}))$, and every expression *target* of type c' in $\mathrm{Expr}(D, \mathrm{Var}^\sharp(H, w_{wt}))$, then
$$\mathsf{Create}(as_{(c,c')})[self, target] \in \mathrm{Stm}(D, H, w).$$
- For every association-end delete action $\mathsf{Delete}(as_{(c,c')}) \in \mathrm{Act}(D)$, every expression *self* of type c in $\mathrm{Expr}(D, \mathrm{Var}^\sharp(H, w_{wt}))$, and every expression *target* of type c' in $\mathrm{Expr}(D, \mathrm{Var}^\sharp(H, w_{wt}))$, then
$$\mathsf{Delete}(as_{(c,c')})[self, target] \in \mathrm{Stm}(D, H, w).$$

Base case (GUI actions):

- For every type $t \in \mathrm{Typ}(D)$, every *variable* of type t in $\mathrm{Var}^\sharp(H, w_{wt})$ and every expression *value* of type t in $\mathrm{Expr}(D, \mathrm{Var}^\sharp(H, w_{wt}))$, then

$$\mathsf{Set}[\mathit{variable}, \mathit{value}] \in \mathrm{Stm}(D, H, w).$$

- For every window $\langle w', \mathsf{Window}\rangle \in W$, every list of variables $\mathit{variable}_1, \ldots,$ $\mathit{variable}_n$, such that, for $1 \leq i \leq n$, $\mathit{variable}_i$ is of type t_i in $\mathrm{Var}(H, w'_{\mathsf{Window}})$, and every list of expressions $\mathit{value}_1, \ldots, \mathit{value}_n$, such that, for $1 \leq i \leq n$, value_i is of type t_i in $\mathrm{Expr}(D, \mathrm{Var}^\sharp(H, w_{wt}))$, then

$$\mathsf{Open}[w', (\boldsymbol{\mathit{variable}, \mathit{value}})] \in \mathrm{Stm}(D, H, w).$$

- Finally,

$$\mathsf{Back}, \mathsf{Fail}, \mathsf{Skip} \in \mathrm{Stm}(D, H, w).$$

Inductive case

- For every expression *cond* of type $\mathsf{Boolean}$ in $\mathrm{Expr}(D, \mathrm{Var}^\sharp(H, w_{wt}))$, and every statements $\mathit{stm}_1, \mathit{stm}_2 \in \mathrm{Stm}(D, H, w)$, then

$$\mathsf{if\_then\_else}[\mathit{cond}, \mathit{stm}_1, \mathit{stm}_2] \in \mathrm{Stm}(D, H, w).$$

- For every expression *source* of type $\mathsf{Sequence}(t)$ in $\mathrm{Expr}(D, \mathrm{Var}^\sharp(H, w_{wt}))$, every widget variable *variable* of type t in $\mathrm{Var}^\sharp(H, w_{wt})$, and every statement $\mathit{body} \in \mathrm{Stm}(D, H, w)$, then

$$\mathsf{iterator}[\mathit{source}, \mathit{variable}, \mathit{body}] \in \mathrm{Stm}(D, H, w).$$

- For all statements stm, $\mathit{stm}' \in \mathrm{Stm}(D, H, w)$, then

$$\mathit{stm}\ ;\ \mathit{stm}' \in \mathrm{Stm}(D, H, w).$$

3.3 Behavioral GUIML models

Definition. Let D be a data model $\langle C, AT, AS, ASO\rangle$. Let H be a GUIML layout model $\langle W, WC, X, EV\rangle$. Then, a GUIML *behavioral model* G is a tuple $\langle D, H, EST\rangle$ such that:

- EST is a set of pairs $\langle\langle ev, w_{wt}\rangle, \mathit{stm}\rangle$, where
  - $\langle ev, w_{wt}\rangle \in EV$ is an event.
  - $\mathit{stm} \in \mathrm{Stm}(D, H, \mathrm{Win}(H, w_{wt}))$ is the statement associated to this event.

Invariants.

- Every event is associated with exactly one statement.
- In every sequence of statement associated to an event, the GUI actions Open and Back can only appear (if at all) at the last position.[5]

[5] When this last position is occupied by an if-then-else, then Open and Back can only appear (if at all) at the last position of its then- or else-branches (and recursively in the case of nested if-then-elses). The situation is similar for iterator statements.

3.4 Operational semantics for events

Notation. Let D be a data model $\langle C, AT, AS, ASO\rangle$. Let I be a D-object model $\langle O, VA, LK\rangle$. In what follows we will use the following notation:

- Let $o_c \in O$ be an object. Then, $(VA \setminus o_c)$ denotes the set that results from *deleting* from VA every triple that contains o_c. That is, $(VA \setminus o_c) = \{\langle o'_{c'}, at_{(c',t')}, va\rangle \mid \langle o'_{c'}, at_{(c',t')}, va\rangle \in VA \wedge o'_{c'} \neq o_c\}$.
- Let $o_c \in O$ be an object. Then, $(LK \setminus o_c)$ denotes the set that results from *deleting* from LK every triple that contains o_c. That is, $(LK \setminus o_c) = \{\langle o'_{c'}, as_{(c',c'')}, o''_{c''}\rangle \mid \langle o'_{c'}, as_{(c',c'')}, o''_{c''}\rangle \in LK \wedge o'_{c'} \neq o_c \wedge o''_{c''} \neq o_c\}$.
- Let $at_{(c,t)} \in AT$ be an attribute. Let $o_c \in O$ be an object and let $va \in [\![t]\!]^{\mathrm{TP}}$ be a value of type t. Then $VA \oplus \langle o_c, at_{(c,t)}, va\rangle$ denotes the set that results from *overriding* (i.e., updating) in VA the value of the attribute at of the object o_c with va. That is, $(VA \oplus \langle o_c, at_{(c,t)}, va\rangle) = \{\langle o_c, at_{(c,t)}, va\rangle\} \cup \{\langle o'_{c'}, at'_{(c',t')}, va'\rangle \mid \langle o'_{c'}, at'_{(c',t')}, va'\rangle \in VA \wedge o'_{c'} \neq o_c \wedge at'_{(c',t')} \neq at_{(c,t)}\}$.

Definition. Let D be a data model $\langle C, AT, AS, ASO\rangle$. Let H be a GUIML layout model $\langle W, WC, X, EV\rangle$. Let G be a GUIML behavioral model $\langle D, H, EST\rangle$. Let $ev \in EV$ be an event in G with $\langle ev, stm\rangle \in EST$. Then, $\mathrm{Sem}(G, ev)$ is the set of all the transitions

$$\langle stm, I, \theta\rangle \longrightarrow^* \langle \mathsf{Skip}, I', \theta'\rangle$$

where $\longrightarrow^*$ is the transitive closure of the small-step transition relation $\longrightarrow$ defined by the following inference rules. For every D-object model $I = \langle O, VA, LK\rangle$ and every (X, I)-substitution we have:

Base case (data actions)

$$\frac{o_c \notin O}{\langle \mathsf{Create}(c)[variable], I, \theta\rangle \longrightarrow \langle \mathsf{Skip}, \langle O \cup \{o_c\}, VA, LK\rangle, \theta \oplus \{variable \mapsto o_c\}\rangle}.$$

$$\frac{[\![(self)\theta]\!]^I = o}{\langle \mathsf{Delete}(c)[self], I, \theta\rangle \longrightarrow \langle \mathsf{Skip}, \langle (O \setminus o_c), (VA \setminus o_c), (LK \setminus o_c)\rangle, \theta\rangle}.$$

$$\frac{[\![(self.at)\theta]\!]^I = va}{\langle \mathsf{Read}(at_{(c,t)})[self, variable], I, \theta\rangle \longrightarrow \langle \mathsf{Skip}, I, \theta \oplus \{variable \mapsto va\}\rangle}.$$

$$\frac{[\![(self)\theta]\!]^I = o, [\![(value)\theta]\!]^I = va}{\langle \mathsf{Update}(at_{(c,t)})[self, value], I, \theta\rangle \longrightarrow \langle \mathsf{Skip}, \langle O, (VA \oplus \langle o, at, va\rangle), LK\rangle, \theta\rangle}.$$

$$\frac{[\![(self.as)\theta]\!]^I = \{o_1, \ldots, o_n\}}{\langle \mathsf{Read}(as_{(c,c')})[self, variable], I, \theta\rangle \longrightarrow \langle \mathsf{Skip}, I, \theta \oplus \{variable \mapsto \{o_1, \ldots, o_n\}\}\rangle}.$$

$$\frac{[\![(self)\theta]\!]^I = o, [\![(target)\theta]\!]^I = o', (as_{(c,c')}, as'_{(c',c)}) \in ASO}{\langle \mathsf{Create}(as_{(c,c')})[self, target], I, \theta\rangle \longrightarrow \langle \mathsf{Skip}, \langle O, VA, (LK \cup \{\langle o, as, o'\rangle, \langle o', as', o\rangle\})\rangle, \theta\rangle}.$$

$$\frac{[\![(self)\theta]\!]^I = o, [\![(target)\theta]\!]^I = o', (as_{(c,c')}, as'_{(c',c)}) \in ASO}{\langle \mathsf{Delete}(as_{(c,c')})[self, target], I, \theta\rangle \longrightarrow \langle \mathsf{Skip}, \langle O, VA, (LK \setminus \{\langle o, as, o'\rangle, \langle o', as', o\rangle\})\rangle, \theta\rangle}.$$

Base case (GUI actions)

$$\frac{[\![(value)\theta]\!]^I = va}{\langle \mathsf{Set}[variable, value], I, \theta\rangle \longrightarrow \langle \mathsf{Skip}, I, \theta \oplus \{variable \mapsto va\}\rangle}.$$

$$\frac{}{\langle \mathsf{Open}[\langle w, \mathsf{Window}\rangle, (\boldsymbol{variable, value})], I, \theta\rangle \longrightarrow \langle \mathsf{Skip}, I, \theta\rangle}.$$

$$\frac{}{\langle \mathsf{Back}, I, \theta\rangle \longrightarrow \langle \mathsf{Skip}, I, \theta\rangle}.$$

Inductive case

$$\frac{[\![(cond)\theta]\!]^I = \mathsf{true}}{\langle \mathsf{If\_then\_else}[cond, stm_1, stm_2], I, \theta\rangle \longrightarrow \langle stm_1, I, \theta\rangle}.$$

$$\frac{[\![(cond)\theta]\!]^I = \mathsf{false}}{\langle \mathsf{If\_then\_else}[cond, stm_1, stm_2], I, \theta\rangle \longrightarrow \langle stm_2, I, \theta\rangle}.$$

$$\frac{[\![(source)\theta]\!]^I = [v_1, \ldots, v_n]}{\langle \mathsf{Iterator}[source, variable, body], I, \theta\rangle \longrightarrow \langle (\mathsf{Set}(variable, v_1)\,;\, body\,;\ldots;\mathsf{Set}(variable, v_n)\,;\, body), I, \theta\rangle}.$$

$$\frac{\langle stm_1, I, \theta\rangle \longrightarrow \langle stm'_1, I', \theta'\rangle}{\langle (stm_1\,;\, stm_2), I, \theta\rangle \longrightarrow \langle stm'_1\,;\, stm_2, I', \theta'\rangle}.$$

$$\frac{}{\langle (\mathsf{Skip}\,;\, stm_2), I, \theta\rangle \longrightarrow \langle stm_2, I, \theta\rangle}.$$

4 Security-aware GUIML

In this section we first characterize *security-awareness* of GUIML behavior models in terms of a transition relation defined by a security-aware version of the inference rules that define the (non security-aware) operational semantics of GUIML events. Then, we define a model transformation that, given a GUIML model G and a SecureUML model S, generates a new GUIML model that is *security aware* with respect to S. Finally, we formalize and prove the correctness of our model transformation.

4.1 Operational semantics for security-aware events

Informally, security-aware events are those events where the execution of the associated actions are *conditional* on the satisfaction of the corresponding authorization constraints. However, which constraint these are depends, of course, on the role of the actual user who triggers this event. Thus, in order to be able to refer to the user's role (when specifying the aforementioned conditions within the statement associated to the event), we will explicitly require that:

- The underlying data model D includes a class Role, with an attribute name of type String.
- Every window in the GUIML model owns two distinguished variables, caller (of the same type than the users) and role (of type Role), whose intented values are, respectively, the actual user and its role

Moreover, when discussing security-awareness with respect to a security model S we will be interested only in D-object models whose objects of type Role are *conformant* with the roles declared in S, in the following sense: Let D be a data model $\langle C, AT, AS, ASO\rangle$, such that $\mathsf{Role} \in C$ and $\langle \mathsf{name}, \mathsf{Role}, \mathsf{String}\rangle \in AT$. Let S be a D-security model $\langle D, R, RH, u, P\rangle$. Let $I = \langle O, VA, LK\rangle$ be a D-object model. Then, we say that I is *R-conformant* if and only if

$$R = \{va \mid \langle o_{\mathsf{Role}}, \mathsf{name}_{(\mathsf{Role},\mathsf{String})}, va\rangle \in VA\}.$$

Definition. Let D be a data model $\langle C, AT, AS, ASO\rangle$, such that $\mathsf{Role} \in C$ and $\langle \mathsf{name}, \mathsf{Role}, \mathsf{String}\rangle \in AT$. Let S be a D-security model $\langle D, R, RH, u, P\rangle$. Let H be a GUIML layout model $\langle W, WC, X, EV\rangle$ such that, for every $w_{\mathsf{Window}} \in W$, $\langle \mathsf{role}_{\mathsf{Role}}, w_{\mathsf{Window}}\rangle \in X$ and $\langle \mathsf{caller}_u, w_{\mathsf{Window}}\rangle \in X$. Let G be a GUIML behavioral model $\langle D, H, EST\rangle$. Let $ev \in EV$ be an event in G whose associated statement is stm, i.e., $\langle ev, stm\rangle \in EST$. Then, the *security-aware* operational semantics for the event ev is given by the set of all the transitions

$$\langle stm, I, \theta\rangle \longrightarrow^*_{\mathrm{sec}} \langle \mathsf{Skip}, I', \theta'\rangle$$

such that I is R-conformant and $\longrightarrow^*_{\mathrm{sec}}$ is the transitive closure of the small-step transition relation $\longrightarrow_{\mathrm{sec}}$ defined by the *security-aware versions* of the inference rules that define the operational semantics of GUIML events. Formally, for every

GUIML data action $act[\boldsymbol{arg}]$, the security-aware version of the corresponding inference rule includes the following additional condition:

$$[\![(\mathrm{AuthPerm}(S, [\![(\mathsf{role}_{\mathrm{Win}(H,ev)}.\mathsf{name})\theta]\!]^I, act)(\mathrm{Subst}(act[\boldsymbol{arg}])))\theta]\!]^I = \mathsf{true},$$

where $\mathrm{Subst}(act[\boldsymbol{arg}])$ is the substitution defined below, which depends on the type of the action act.

$$\begin{aligned}
&\mathrm{Subst}(\mathsf{Create}(c)[variable]) = \\
&\qquad \{\mathsf{caller}_u \mapsto \langle \mathrm{caller}_u, w_{\mathsf{Window}}\rangle\}. \\
&\mathrm{Subst}(\mathsf{Delete}(c)[self]) = \\
&\qquad \{\mathsf{caller}_u \mapsto \langle \mathrm{caller}_u, w_{\mathsf{Window}}\rangle, \mathsf{self}_c \mapsto self\}. \\
&\mathrm{Subst}(\mathsf{Read}(at_{(c,t)})[self, variable]) = \\
&\qquad \{\mathsf{caller}_u \mapsto \langle \mathrm{caller}_u, w_{\mathsf{Window}}\rangle, \mathsf{self}_c \mapsto self\}. \\
&\mathrm{Subst}(\mathsf{Update}(at_{(c,t)})[self, value]) = \\
&\qquad \{\mathsf{caller}_u \mapsto \langle \mathrm{caller}_u, w_{\mathsf{Window}}\rangle, \mathsf{self}_c \mapsto self, \mathsf{value}_t \mapsto value\}. \\
&\mathrm{Subst}(\mathsf{Read}(as_{(c,c')})[self, variable]) = \\
&\qquad \{\mathsf{caller}_u \mapsto \langle \mathrm{caller}_u, w_{\mathsf{Window}}\rangle, \mathsf{self}_c \mapsto object\}. \\
&\mathrm{Subst}(\mathsf{Create}(as_{(c,c')})[self, target]) = \\
&\qquad \{\mathsf{caller}_u \mapsto \langle \mathrm{caller}_u, w_{\mathsf{Window}}\rangle, \mathsf{self}_c \mapsto self, \mathsf{target}_{c'} \mapsto target\}. \\
&\mathrm{Subst}(\mathsf{Delete}(as_{(c,c')})[self, target]) = \\
&\qquad \{\mathsf{caller}_u \mapsto \langle \mathrm{caller}_u, w_{\mathsf{Window}}\rangle, \mathsf{self}_c \mapsto self, \mathsf{target}_{c'} \mapsto target\}.
\end{aligned}$$

The inference rules for GUI actions are not modified in their security-aware versions. The inference rules for if-then-else statements, iterator statements, or sequences of statements also remain unmodified.

4.2 Security-aware model transformation

Definition. Let D be a data model $\langle C, AT, AS, ASO\rangle$, such that $\mathsf{Role} \in C$ and $\langle \mathsf{name}, \mathsf{Role}, \mathsf{String}\rangle \in AT$. Let S be a D-security model $\langle D, R, RH, u, P\rangle$. Let H be a GUIML layout model $\langle W, WC, X, EV\rangle$ such that, for every $w_{\mathsf{Window}} \in W$, $\langle \mathsf{role}_{\mathsf{Role}}, w_{\mathsf{Window}}\rangle \in X$ and $\langle \mathsf{caller}_u, w_{\mathsf{Window}}\rangle \in X$. Let G be a GUIML behavioral model $\langle D, H, EST\rangle$. Then, $\mathrm{Sec}(G, S)$ is the *S-security-aware version* of G, defined as follows:

$$\mathrm{Sec}(G, S) = \langle D, H, \{\langle ev, \mathrm{Sec}(stm, S)\rangle \mid \langle ev, stm\rangle \in EST\}\rangle.$$

Here $\mathrm{Sec}(stm, S)$ is the *S-security-aware version* of the statement stm associated to the event ev, defined recursively as follows:

Base case (data actions): Let $R = \{r_1, \ldots, r_n\}$. Then,

$$
\begin{array}{l}
\mathrm{Sec}(act[\boldsymbol{arg}], S) = \\
\quad \mathsf{If\_then\_else}[r_1 = \mathsf{role}_{\mathrm{Win}(H,ev)}.\mathsf{name}, \\
\qquad \mathsf{If\_then\_else}[\mathrm{AuthPerm}(S, r_1, act)(\mathrm{Subst}(act[\boldsymbol{arg}])), \\
\qquad\qquad act[\boldsymbol{arg}], \\
\qquad\qquad \mathsf{Fail}], \\
\qquad \ldots \\
\qquad\quad \mathsf{If\_then\_else}[r_n = \mathsf{role}_{\mathrm{Win}(H,ev)}.\mathsf{name}, \\
\qquad\qquad \mathsf{If\_then\_else}[\mathrm{AuthPerm}(S, r_n, act)(\mathrm{Subst}(act[\boldsymbol{arg}])), \\
\qquad\qquad\qquad act[\boldsymbol{arg}]), \\
\qquad\qquad\qquad \mathsf{Fail}], \\
\qquad\qquad \mathsf{Fail}] \ \ldots\ ].
\end{array}
$$

Here $\mathrm{Subst}(act[\boldsymbol{arg}])$ is the substitution defined above, where w_{Window} is in this case the window that contains the widget that supports the event ev.

Base case (GUI actions):

$\mathrm{Sec}(\mathsf{Set}[variable, value], S) = \mathsf{Set}[variable, value]$.
$\mathrm{Sec}(\mathsf{Back}, S) = \mathsf{Back}$.
$\mathrm{Sec}(\mathsf{Skip}, S) = \mathsf{Skip}$.
$\mathrm{Sec}(\mathsf{Open}[w_{\mathsf{Window}}, (\boldsymbol{variable}, \boldsymbol{value})], S) = \mathsf{Open}(w_{\mathsf{Window}}, (\boldsymbol{variable}, \boldsymbol{value}))$.

Inductive cases.

$$
\begin{array}{l}
\mathrm{Sec}(\mathsf{if\_then\_else}[cond, stm_1, stm_2], S) = \\
\quad \mathsf{if\_then\_else}[cond, \mathrm{Sec}(stm_1, S), \mathrm{Sec}(stm_2, S)]. \\
\mathrm{Sec}(\mathsf{iterator}[source, variable, body], S) = \mathsf{iterator}[source, variable, \mathrm{Sec}(body, S)]. \\
\mathrm{Sec}((stm_1\,;\, stm_2), S) = (\mathrm{Sec}(stm_1, S)\,;\, \mathrm{Sec}(stm_2, S)).
\end{array}
$$

4.3 Correctness

The following theorem basically states that the evaluation of a transformed statement following the non-security-aware operational semantics for events returns the same result than its evaluation using the security-aware version of this semantics and, therefore, that the transformed statement respects the authorization constraints formalized in the underlying security model.

Theorem. Let D be a data model $\langle C, AT, AS, ASO\rangle$, such that $\mathsf{Role} \in C$ and $\langle \mathsf{name}, \mathsf{Role}, \mathsf{String}\rangle \in AT$. Let S be a D-security model $\langle D, R, RH, u, P\rangle$. Let H be a GUIML layout model $\langle W, WC, X, EV\rangle$ such that for every $w_{\mathsf{Window}} \in W$,

$\langle \mathsf{role}_{\mathsf{Role}}, w_{\mathsf{Window}} \rangle \in X$ and $\langle \mathsf{caller}_u, w_{\mathsf{Window}} \rangle \in X$. Let G be a GUIML behavioral model $\langle D, H, EST \rangle$. Let $w_{\mathsf{Window}} \in W$ be a window and let $stm \in \mathrm{Stm}(D, H, w)$. Then, for every R-conformant D-object data model I, and every (X, I)-substitution θ,

$$\langle \mathrm{Sec}(stm, S), I, \theta \rangle \longrightarrow^* \langle \mathsf{Skip}, I', \theta' \rangle. \iff \langle stm, I, \theta \rangle \longrightarrow^*_{\mathrm{sec}} \langle \mathsf{Skip}, I', \theta' \rangle.$$

Proof. By induction on stm.

Acknowledgements

This work is partially supported by the EU FP7-ICT Project "NESSoS: Network of Excellence on Engineering Secure Future Internet Software Services and Systems" (256980) by the Spanish Ministry of Science and Innovation Project "DESAFIOS-10" (TIN2009-14599-C03-01), and by Comunidad de Madrid Program "PROMETIDOS-CM" (S2009TIC-1465).

References

1. Object Management Group. *Object Constraint Language specification Version 2.3.1*, January 2012. `http://www.omg.org/spec/OCL/2.3.1`.