A Global Semantics for Views (Long Version)

last revision: February 25, 2000

Christine Choppy[1], Pascal Poizat[2], and Jean-Claude Royer[2]

[1] LIPN, Institut Galilée - Université Paris XIII,
Avenue Jean-Baptiste Clément, F-93430 Villetaneuse, France
Christine.Choppy@lipn.univ-paris13.fr
[2] IRIN, Université de Nantes
2 rue de la Houssinière, B.P. 92208, F-44322 Nantes cedex 3, France
{Pascal.Poizat,Jean-Claude.Royer}@irin.univ-nantes.fr
http://www.sciences.univ-nantes.fr/info/perso/permanents/poizat/

Abstract. We focus on the specification of mixed systems that contain static and dynamic aspects. In previous existing approaches, the formal links and consistency between the static and the dynamic aspects are either not defined, or encompassed within a single framework. Moreover, in mixed systems, the definition of structuring and uniform mechanisms are also required. Our approach aims at keeping advantage of the languages dedicated to both aspects (algebraic specifications for data types, and state transition diagrams for dynamic behaviour) while providing an underlying unifying framework accompanied by an appropriate semantic model. This underlying framework is based on our notion of *views*. In [3] we addressed the composition of any number of views in a composition. Here we address the strong links that exist between all aspects of a single component, and that altogether build its global semantics. After presenting the (state and transition) formulas and their semantics, we show how to glue the different aspects together, and we present the retrieval of the global view for components. We provide here a new set of rules for the STS part of our views, taking into account the composition of all aspects. We then give the formal (global) semantics for such global views. We illustrate our view model on a password management example.

Keywords: mixed specifications, view formalism, global semantics.

1 Introduction

We focus on the specification of mixed systems that contain static and dynamic aspects and feature a certain level of complexity that requires the definition of structuring mechanisms.

We use two standard ways of achieving structuring/modularity that are, on the one hand, the extension (a simple form of inheritance), and on the other hand, the union (or composition). These two means come from the algebraic specification formalisms [19] and have been adapted to the object oriented analysis and design context.

In the last few years, the need for a separation of concerns with reference to static and dynamic aspects appeared. This issue was addressed in (non exhaustive list) LOTOS [6] (Abstract Data Types and CSP/CCS), SDL [4] (Abstract Data Types and State/Transition Diagrams), and also more recently in approaches combining Z and process algebras (CSP-OZ [17] for example).This is also reflected in object oriented analysis and design approaches such as OMT or UML [18] where static and dynamic aspects are dealt with by different diagrams (class diagrams, interaction diagrams, statecharts). However, in both approaches, the (formal) links and consistency between the two aspects are not defined, which limits the possibilities of reasoning on the whole component.

Some approaches encompass both aspects within a single framework, like for example LTS [15], rewriting logic [10], TLA [7].

Our approach aims at keeping advantage of the languages dedicated to both aspects (*e.g.* algebraic specifications for data types, and state transition diagrams for dynamic behaviour) while providing an underlying unifying framework accompanied by an appropriate semantic model.

Our specification formalism (called KORRIGAN) is integrated into a framework for system specification (Fig. 1). In [13, 12], we define a methodology for our specifications that makes use of conditions in order to build abstractions of systems and that is applied to generate LOTOS and SDL specifications and verify them. In [14], we explain how concurrent object oriented code may be generated from our specifications.

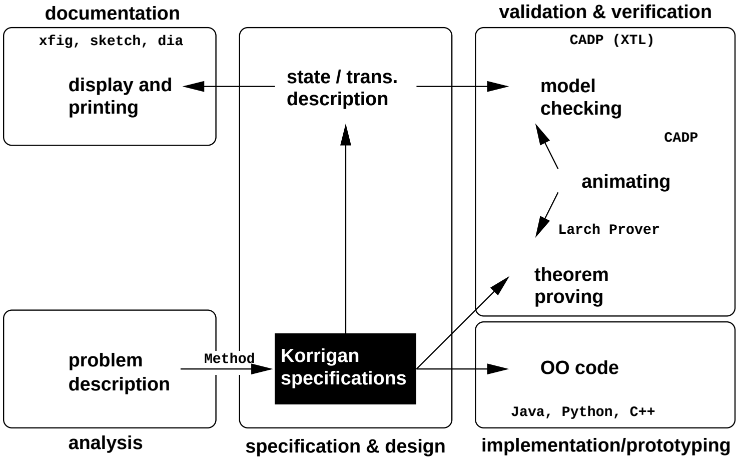


Fig. 1. The specification framework

After presenting our choices for the kind of communication between identified communicating components, we present each of the view classes of our model (and the corresponding specification language).Whereas in [3] we addressed the composition of any number of views, we address here the strong links that exist between all aspects of a single component, and that altogether build its global semantics. Therefore, after presenting the formulas (and their semantics) we show how to glue the different aspects together, and we present the retrieval of the global view for components. We provide here a new set of rules for the STS part of our views (see below Definition 4), taking into account the composition of all aspects. We then give the formal (global) semantics for such global views. We illustrate our view model on a password management example.

2 The Password Management Example

The system is composed of several basic users, a privileged user (root) and the password manager. The basic users may change their password. The privileged user may act as a basic user (changing its password) but may also create passwords for new users and change passwords for actual users.

The creation of a password is done as follows: (i) the user id is prompted, then either (iia) it already exists and an error is signaled, or (iib) the password is prompted for. Thereafter (if no error occurred), the password is prompted for once again and either (iiia) it is different (and an error is signaled), or (iiib) it is the same and the password file is modified.

The change of a password is the same but for the difference that (iv) the user has to exist and that the (old) password is asked for (and verified) before the new password part (iib, iiia or iiib) takes place.

More complex systems, with password restrictions, may exist but we do not take them into account here.

3 The View Model

A Specification Framework. We will here present the different views that compose our view model. As we deal with identified communicating components, we first present our communication choices and our notion of component identifier.

Communication Choices. Dealing with communicating components, one has first to make choices with respect to the communication arity and the various ways to express communication (Table 1).

Since we are concerned both with object-orientation and readability, we will restrict ourselves to the $1 \rightarrow \mathrm{n}$ communication type. LOTOS-like multiway communication may be more expressive but less easily understandable. Moreover, we will allow both explicit (with our process identifiers being some kind of URL) and implicit communication (through the use of the `sender` keyword, denoting the sender of the last message being received by the component).

Identifiers. A step to object orientation are (object) identifiers. Moreover, our communication mechanisms need component identifiers. Identifiers are simple

Table 1. Communication choices

	* $1 \rightarrow 1$ (*e.g.* Object-Oriented languages, SDL)
arity	* $1 \rightarrow$ n (in parallel or via atomic sequence)
	* n $\leftrightarrow$ n (*e.g.* LOTOS multiway synchronization)
implicit	
	* using special keywords (*e.g.* `sender` in SDL)
	* by structuring and possibly hiding (*e.g.* SDL, LOTOS)
explicit	
	* `from` + process identifier(s)
	* `to` + process identifier(s) (*e.g.* SDL)

terms of `PId` sorts. Each different component class has its associated `PId` sort. `PId` subsorts correspond to related (through inheritance) component classes.

We have two important relations on `PId`: *object equality* and *object structure*. Object equality expresses that two objects with the same identifier denote actually the same object: $\forall o, o' \in Obj, Id(o) = Id(o') \Leftrightarrow o = o'$.

Object structure is expressed by the fact that, whenever an object O is composed of several sub-objects, then the object identifier is a prefix for all sub-objects identifiers: $\forall o \in Subobjects(O), \exists i : PId \, . \, Id(o) = Id(O).i$.

This way, identifiers build a tree-ordered id-indexed structure over object composition. *Id-indexing* is widely used when dealing with components built over other (sub-)components.

3.1 The Different View Classes

We define an *Internal Structuring View* abstraction (Fig. 2) that expresses the fact that, in order to design an object, it is useful to be able to express it under its different aspects (here the static and dynamic aspects, with no exclusion of further aspects that may be identified later on).

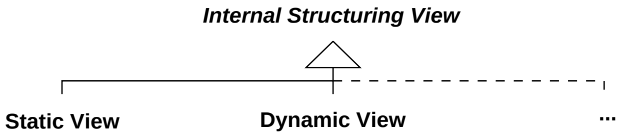


Fig. 2. Internal structuring class diagram for the view model (using the UML notation)

We therefore have here a first structuring level (*internal structuring*) for objects. Another structuring level is the *external structuring* (Fig. 3), expressed by the fact that an object may be composed of several other objects. An object may then be either a simple one (integrating different internal structuring views in an *integration view*), or a composite object (*composition view*).

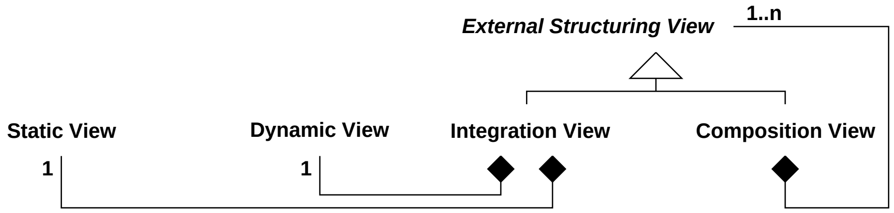


Fig. 3. External structuring class diagram for the view model (using the UML notation)

Integration views follow an *encapsulation principle*: the static part (data type) may only be accessed through the dynamic part (behaviour) and its identifier (`Id`).

The whole class diagram for the view model is given in Figure 4.

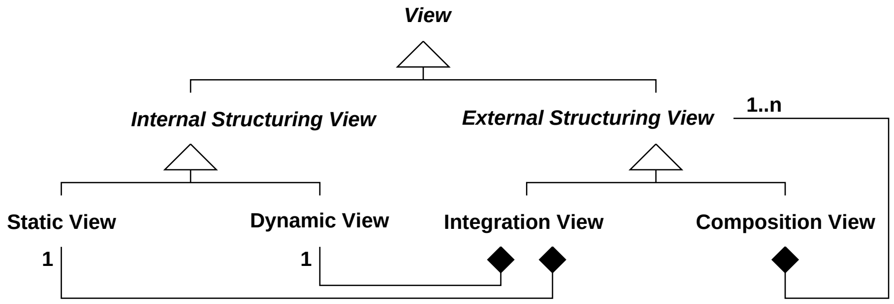


Fig. 4. The view model class diagram (using the UML notation)

Views. In a unifying approach, views are used to describe the different aspects of a component (internal and external structuring).

Definition 1 (View). *A* view *V_T of a component T is a triple (A_T, α, D_T) where A_T is the data part, α is an abstraction of T and D_T a dynamic layer over A_T.*

The data part addresses the functional issues for the component (definition of the operation values, in an algebraic way). The abstraction defines a point of view for the component, and the dynamic layer addresses the possible sequences of operations the component may perform.

Definition 2 (View data part). *A* data part *A_T (in a view structure (A_T, α, D_T)) is an algebraic specification with the following components $(A', G, V, \Sigma_T, Ax_T, \overline{A})$ where:*

- *A' is a set of imported basic algebraic specifications (datatypes without views).*
- *G is a set of formal parameter algebraic specifications.*

- *V are parameter variables.*
- *Σ_T, Ax_T are the signature and axioms (first order formulas) for T.*
- *$\overline{A}$ are hidden (or non-exported) operations.*

Dynamic signatures are signatures given in a LOTOS-like form [2], *i.e.* enriched with arguments concerning the explicit communication scheme.

There is a simple correspondence between dynamic and usual (static) signatures:

- `comm ?m:Msg from u:PId`
  (reception of a message `m` from `u` on the gate `comm`)
  constructor `comm : T, Msg, PId` → `T`
- `comm !m:Msg to u:Pid`
  (emission of a message `m` to `u` on the gate `comm`)
  constructor `comm_c : T` → `T`
  observer `comm_o_m : T` → `Msg`
  observer `comm_o_id : T` → `PId`

The view dynamic layer describes the possible sequences of actions the component may perform. The main idea is to define operations in terms of their effect on conditions (C) that build the component abstraction ([13] explains how these conditions are identified). Hence, with each operation, its preconditions and postconditions will be associated.

Definition 3 (View dynamic layer). *A* dynamic layer *D_T (in a view structure (A_T, α, D_T)) is a tuple $(Id, O_T, \overline{O_T})$ where:*

- *Id is an identifier of sort PId_T.*
- *O_T is a set of operations requirements (built over dynamic signatures, pre and postconditions),* i.e. *$O_T \subseteq Term_{\Sigma_T} \times P \times Q$ where P are preconditions and Q are postconditions.*
- *$\overline{O_T}$ (built on $\overline{A}$) are hidden members of O_T.*

The preconditions are first order formulas expressing the operation requirements in terms of the values of the C conditions (P_C) and of the $Term_{\Sigma_T}$ arguments (denoted by P_{io}).

For example, given a specific buffer operation $put(self, x)$ that is only possible with even numbers, and when the buffer is not full, we have the following precondition $\neg full \wedge even(x)$ where $full$ belongs to C (hence to P_C), and $even(x)$ belongs to P_{io}.

The postconditions are a set of first order formulas $\{Q_{C'}\}$ expressing for each operation the values C' of the C conditions *after* the operation is applied in terms of the values of the C conditions (Q_C), of some new Cl "limit" conditions[1] (Q_{Cl}) and the $Term_{\Sigma_T}$ arguments (denoted by Q_{io}) *before* the operation is applied.

[1] Experience shows that these are required to express the C conditions values in postconditions.

Taking the same *put* operation, and in the case where the buffer abstraction is built over $C = \{empty, full\}$, we have the postcondition $\{Q_{empty'}, Q_{full'}\}$ and $Q_{empty'} = false, Q_{full'} = nextFull$ (*i.e.* the buffer is *full* if, before the *put* operation took place, it would be *full* the next time). As one can see, we had to use a limit condition ($nextFull$) to express the postcondition over *full*.

α uses the condition predicates(C) to define equivalence classes for T. C being finite, there is a finite number of equivalence classes.

Definition 4 (View abstraction). *An* abstraction α *(in a view structure* (A_T, α, D_T)*) is a tuple* (C, Cl, Φ, Φ_0) *where:*

- $C \subseteq \Sigma_T$ *is a finite set of predicates.*
- $Cl \subseteq \Sigma_T$ *is a finite set of "limit" conditions used in postconditions of operations.*
- $\Phi \subseteq Ax_T$ *is a set of formulas over members of* C *(valid combinations).*
- $\Phi_0 \subseteq Ax_T$ *is a formula over members of* C *(initial values).*

The syntax for views in KORRIGAN is given in Figure 5. The `SPECIFICATION` part corresponds to A_T, the `ABSTRACTION` part to α, and the `OPERATIONS` part to D_T. There is no need to state explicitly $\overline{O}$ as it may be retrieved using $\overline{A}$. There is no identifier either since identifiers shall only appear when the component is used in a composition (external structuring views).

VIEW T	
SPECIFICATION	**ABSTRACTION**
imports A' **generic on** G **variables** V **hides** $\overline{A}$ **ops** Σ **axioms** Ax	**conditions** C **limit conditions** Cl **with** Φ **initially** Φ_0 **OPERATIONS** O_i **pre:** P **post:** Q

Fig. 5. The KORRIGAN syntax (views)

As explained in [3], views may be defined using either the abstraction and operation parts or Symbolic Transition Systems (STS) [5, 16]. A trivial abstraction and operation parts may then be derived from the STS. Conversely, an STS may be obtained in a unique way from the abstraction and operation parts. Note that STS parts of views are given using dynamic signatures (see Definition above). The corresponding KORRIGAN syntax is given in Figure 6. Examples of view structures in their alternative form are given in Figure 7 (see also [3]).

One interesting property is that, as said before, views use an abstraction (α) to describe equivalence classes for the components. The STS has exactly one state per equivalence class. The number of equivalence classes being finite, the

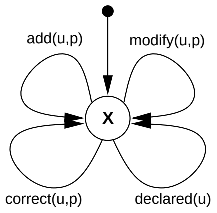
VIEW T	
SPECIFICATION	**STS**
imports A'	.→initialState
generic on G	sourceState − transition → targetState
variables V	...
hides $\overline{A}$	
ops Σ	
axioms Ax	

Fig. 6. The Korrigan alternative syntax (views)

STS is also finite.

In the following, we describe the static and the dynamic views. In our unifying approach, dynamic views are the same as static views but for some differences: (i) no full algebraic specification (only signatures matter) and, (ii) the `from/to` elements (using PId types). This stems from the fact that dynamic views focus only on the communication abstraction.

Static Views. Static views (SV) describe the static aspect of a component. A *static view* SV_T of a component T is a view. The password manager static view is given in Figure 7.



STATIC VIEW StaticPasswordManager (SPM)

SPECIFICATION

imports Boolean, UserId, Password
ops
  empty : → SPM
  add : SPM, UserId, Password → SPM
  modify : SPM, UserId, Password → SPM
  declared : SPM, UserId → Boolean
  correct : SPM, UserId, Password → Boolean
axioms ...

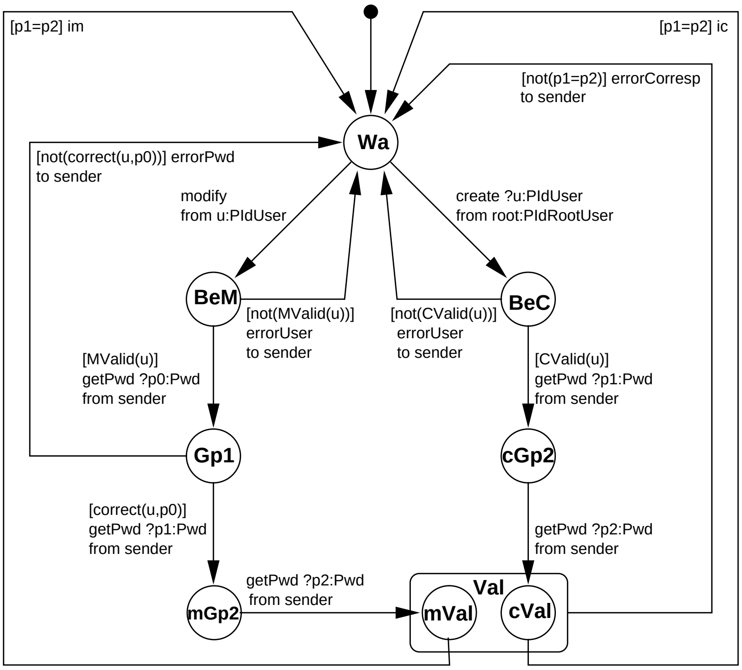
STS

.→X
X − add ?u : UserId ?p : Password → X
X − modify ?u : UserId ?p : Password → X
X − declared ?u : UserId !b : Boolean → X
X − correct ?u : UserId ?p : Password !b : Boolean → X

Fig. 7. Password Manager static view (SPM)

Dynamic Views. Dynamic views (DV) describe the behavioural aspect of a component, that is a process (or an active object). For dynamic views, we consider only communication protocols (emissions and receptions, in terms of senders and receivers) and not the definition of outputs in terms of inputs. In this sense, it is really a partial view (communication abstraction) of a component, without any global semantics.

A *dynamic view* DV_T of a component T is a view where the signatures (Σ_T) are dynamic and where the axiom part (Ax_T) is reduced to Φ and Φ_0. The password manager dynamic view (graphical representation of the STS part) is given in Figure 8.



The state names stand for `waiting` (**Wa**), `begin modification` (**BeM**), `begin creation` (**BeC**), `get first password` (**Gp1**), `get second password on modification` (**mGp2**), `get second password on creation` (**cGp2**), `validate` (**Val**, **mVal** on modification and **cVal** on creation).

Fig. 8. Password Manager dynamic view (DPM)

External Structuring Views. An external structuring view (ESV) may contain one or more global semantics components (*i.e.* integration views or composition views). External structuring views have a global semantics.

Definition 5 (External Structuring View). *An* external structuring view *ESV_T of a component T is a triple (A_T, Θ, K_T) where A_T is an optional data part, Θ is the ESV "glue" part, and K_T the ESV composition layer.*

Some new parameters and/or basic algebraic data types may be needed in the ESV.

Definition 6 (ESV data part). *An ESV* data part A_T *(in an ESV structure* (A_T, Θ, K_T)*) is made up of following components* $(A', G, V, \overline{A})$ *where:*

- *A' is a set of imported basic algebraic specifications (datatypes without views).*
- *G is a set of formal parameter algebraic specifications.*
- *V are parameter variables.*
- *$\overline{A}$ are hidden (or non-exported) operations.*

Θ describes how the components of T are glued altogether.

Definition 7 (ESV glue part). *An ESV* glue part Θ *(in an ESV structure* (A_T, Θ, K_T)*) is a tuple* $(Ax_\Theta, \Phi, \Psi, \Phi_0, \delta)$ *where:*

- *Ax_Θ are axioms relating guards and operations across different views (see the Password Manager Integration View in Figure 11).*
- *Φ is a (state) formula built over id-indexed C members of the components views*[2]*.*
- *Ψ is a (transition) id-indexed formula relating (gluing) transitions from several components STS. This is a generalized form of synchronized product vector [1].*
- *Φ_0 is a (state) id-indexed formula expressing the initial states of the components (it should be coherent with the Φ_0s of the composition components, that is $\vdash \Phi_0 \Rightarrow \bigwedge_{id} id.\Phi_0(Obj_{id})$).*
- *δ defines what to do with non-glued transitions (see Section 4).*

Φ and Ψ are implicitly universally quantified over time (*i.e.* we have[3] $AG\Phi$ and $AG\Psi$). Ψ is given as couples $\Psi = \{\psi_i = (\psi_{i_s}, \psi_{i_d})\}$, and means $AG \bigwedge_i \psi_{i_s} \Leftrightarrow \psi_{i_d}$. Θ uses environments, state and transition formulas.

Environments. An environment Γ binds variables to states or to transitions.

$$\Gamma : Var \rightarrow (STATE \cup TRANS)$$

State and transition formulas are simple ones: there are no fixpoint or special modalities like in more expressive temporal logics such as CTL, μCRL or XTL [9]. The main point here is that this is expressive enough to express the glue between components. More complex logics may be defined afterwards to verify the models properties (following for example the [8] approach).

State Formulas. The state formulas Φ interpretation domain is $STATE$, the STS set of states.

$$\Phi ::= c \mid x \mid \exists x.\Phi_1 \mid true \mid false \mid \neg\Phi_1 \mid \Phi_1 \wedge \Phi_2 \mid \ -\Psi_1 \mid i.\Phi_1$$

A state verifies a condition c (in the C set of conditions given in the view structure) if c is true in s (denoted by $c(s)$).

[2] This is used for example to express (mutual) exclusion between two conditions/states of two different components.

[3] Where AG denotes “always globally”.

$$\frac{c(s)}{\Gamma, s \vdash c}\ PROP_s$$

A state verifies x if x is bound to a state s' in the environment Γ, and if s and s' are similar ($s \sim s'$). Two states are *similar* if $\forall c \in C\ .\ c(s) \Leftrightarrow c(s')$.

$$\frac{\{x \mapsto s'\} \in \Gamma, s \sim s'}{\Gamma, s \vdash x}\ IDENT_s$$

This formula is used to bind variables to states. A state s verifies $\exists x.\Phi_1$ if it verifies Φ_1 in an environment where x has been bound to s.

$$\frac{\Gamma \cup \{x \mapsto s\}, s \vdash \Phi_1}{\Gamma, s \vdash \exists x.\Phi_1}\ BIND_s$$

Every state verifies *true*.

$$\frac{}{\Gamma, s \vdash true}\ TRUE_s$$

No state verifies *false*.

$$\frac{}{\Gamma, s \not\vdash false}\ FALSE_s$$

A state verifies $\neg\Phi_1$ if it does not verify Φ_1.

$$\frac{\Gamma, s \not\vdash \Phi_1}{\Gamma, s \vdash \neg\Phi_1}\ NOT_s$$

A state verifies $\Phi_1 \wedge \Phi_2$ if it verifies both Φ_1 and Φ_2.

$$\frac{\Gamma, s \vdash \Phi_1 \wedge \Gamma, s \vdash \Phi_2}{\Gamma, s \vdash \Phi_1 \wedge \Phi_2}\ AND_s$$

This state formula links state and transition formulas. A state s verifies $-\Psi_1$ if there is a transition t outgoing from s that verifies Ψ_1.

$$\frac{\exists t = s \xrightarrow{l} s' \in T, \Gamma, t \vdash \Psi_1}{\Gamma, s \vdash -\Psi_1}\ TRANS$$

A state verifies $i.\Phi_1$ if it has a subcomponent i that verifies Φ_1. This rule enables us to deal with the system structuring (indexing).

$$\frac{\exists i.s' \in s\ ,\ \Gamma, s' \vdash \phi_1}{\Gamma, s \vdash i.\phi_1}\ INDEX_s$$

Transition Formulas. The transition formulas Ψ interpretation domain is $TRANS$ the STS set of transitions.

$$\Psi ::= [g]l[g'] \mid x \mid \exists x.\Psi_1 \mid true \mid false \mid \neg\Psi_1 \mid \Psi_1 \wedge \Psi_2 \mid\ > \Phi_1 \mid i.\Psi_1$$

A t transition verifies $s \xrightarrow{[g_t]\ l_t\ [g'_t]} s'$ if it has similar offers, and if t guards imply the formula ones. Two offers $l = po_i$ and $l_t = p_t o_{t_j}$ are similar (denoted by $l \sim l_t$) if the communication gates have the same name ($p = p_t$), if the numbers of offer parameters are the same ($i = j$), and if these offer parameters correspond ($\forall i.o_i = o_{t_i}$).

$$\frac{t = s \xrightarrow{[g_t]\ l_t\ [g'_t]} s', l \sim l_t, g_t \Rightarrow g, g'_t \Rightarrow g'}{\Gamma, t \vdash [g]l[g']}\ PROP_t$$

A transition verifies x if x is bound to a transition t' in the Γ environment, and if t and t' are similar in the Γ environment ($t \sim_\Gamma t'$), that is if $\Gamma, t \vdash t'$ and $\Gamma, t' \vdash t$.

$$\frac{\{x \mapsto t'\} \in \Gamma, t \sim t'}{\Gamma, t \vdash x}\ IDENT_t$$

EXTERNAL STRUCTURING VIEW T	
SPECIFICATION	**COMPOSITION δ**
imports A'	**is**
generic on G	$id_i : Obj_i[I_i]$
variables V	**axioms** Ax_Θ
hides $\overline{A}$	**with** Φ, Ψ **initially** Φ_0

Fig. 9. The KORRIGAN syntax (external structuring views)

This formula is used to bind variables to transitions. A transition t verifies $\exists x.\Psi_1$ if it verifies Ψ_1 in an environment where x has been bound to t.

$$\frac{\Gamma \cup \{x \mapsto t\}, t \vdash \Psi_1}{\Gamma, t \vdash \exists x.\Psi_1} \; BIND_t$$

Every transition verifies $true$.

$$\frac{}{\Gamma, t \vdash true} \; TRUE_t$$

No transition verifies $false$.

$$\frac{}{\Gamma, t \nvdash false} \; FALSE_t$$

A transition verifies $\neg\Psi_1$ if it does not verify Ψ_1.

$$\frac{\Gamma, t \nvdash \Psi_1}{\Gamma, t \vdash \neg\Psi_1} \; NOT_t$$

A transition verifies $\Psi_1 \wedge \Psi_2$ if it verifies both Ψ_1 and Ψ_2.

$$\frac{\Gamma, t \vdash \Psi_1 \wedge \Gamma, t \vdash \Psi_2}{\Gamma, t \vdash \Psi_1 \wedge \Psi_2} \; AND_t$$

This transition formula links transition and state formulas. A transition t verifies $> \Phi_1$ if it outgoes from a state s to a state s' such that s' verifies Φ_1.

$$\frac{t = s \xrightarrow{l} s', \Gamma, s' \vdash \Phi_1}{\Gamma, t \vdash > \Phi_1} \; TARGET$$

A transition verifies $i.\Psi_1$ if it has a subcomponent i that verifies Ψ_1. This rule enables us to deal with the system structuring (indexing).

$$\frac{\exists i.t' \in t \;,\; \Gamma, t' \vdash \psi_1}{\Gamma, t \vdash i.\psi_1} \; INDEX_t$$

As usual, $\models$ may be defined for states as $s \models \Phi$ iff $\{\}, s \vdash \Phi$, and for transitions as $t \models \Psi$ iff $\{\}, t \vdash \Psi$.

Definition 8 (ESV composition layer). *An ESV* composition layer K_T *(in a composition view structure* (A_T, Θ, K_T)*) is a tuple* (Id, Obj_{id}, I_{id}) *where:*

- *Id is an identifier of sort* PId_T.
- Obj_{id} *is a id-indexed family of objects (integration views or composition views). Identifiers follow the equality and structuring principles.*
- I_{id} *are id-indexed sets of terms instantiating the components parameters.*

The syntax for external structuring views in KORRIGAN is given in Figure 9.

There is a shorthand that may be used in ESVs: the *range* operator. The *range* operator is a universal quantifier over a set of identifiers.

For example, to stress that a `car` is made up of four `wheels`, we shall write in the composition section $i : [1..4] * wheel.i : WHEEL$. The same operator may be used in the `with` part of this section to express that all `wheels` have to go in the same direction (using for example two "moving" and "rearMode" conditions for the `wheel` components, and stating that, up to these two conditions, any two `wheels` are similar.

INTEGRATION VIEW T

SPECIFICATION	**COMPOSITION** δ
imports A'	**is**
generic on G	STATIC s: $Obj_s[I_s]$
variables V	DYNAMIC d: $Obj_d[I_d]$
hides $\overline{A}$	**axioms** Ax_Θ
	with Φ, Ψ **initially** Φ_0

Fig. 10. The KORRIGAN syntax (integration views)

```
INTEGRATION VIEW Password Manager
COMPOSITION ALONE
is
    STATIC s: SPM
    DYNAMIC d: DPM
axioms
    d.MValid(s,u) == s.declared(s,u)
    d.CValid(s,u) == not(s.declared(s,u))
    d.correct(s,u,p) == s.correct(s,u,p)
with true, {
    (d.([true] im), s.(modify(d.u,d.p1))),
    (d.([true] ic), s.(add(d.u,d.p1)))
 }
initially true
```

Fig. 11. Password Manager integration view

Integration Views. Integration views (IV) are used to describe objects that have a global semantics (*i.e.* the integration of all the object aspects). Integration views are a special kind of external structuring views (same structure) with the constraint that it is composed of exactly one view per aspect of the component (for the moment, one static and one dynamic view, identified respectively by s and d).

The syntax for integration views in KORRIGAN is given in Figure 10, and its application to the case study in Figure 11.

Composition Views. A composition view (CV) is an external structuring view that may contain one or more global semantics components (*i.e.* integration views or composition views). A composition corresponds to the union of subcomponents. Its structure and the corresponding KORRIGAN syntax is the same as for external structuring views.

4 A Global Semantics for Views

External Structuring Views

These views have a global semantics in the sense that they specify objects with all aspects (internal structuring) well defined. First of all, we have to explain how we may obtain a view structure (such a view is global for the same reasons) for ESV. Then, we shall give its semantics. Another way could have been to give directly a semantics from the ESV structure.

ESV View Structures Retrieval. ESV view structures are a special kind of view where states (resp. transitions) are built as an indexed compound of states (transitions). This indexing enables us to keep the system internal structure while "gluing" components together. This gluing is used in order to specify how the states and transitions of the static and dynamic aspects are related.

We may work through the obtaining of a view structure for an ESV either in its (A_T, α, D_T) form, or its (A_T, STS) form (with $STS = (S, S_0, T)$). We shall work on the retrieval of a (A_T, STS) structure from the ESV structure $(A_T, \varTheta, K_T)$, as working on STS is easier.

$\varTheta$ (in ESV structures, Definition 7) gives glue rules for both states and transitions. We should also define what to do with any transition that has not to be explicitly glued (*stand-alone transition*). This is the role of the δ component of $\varTheta$. We propose three different possibilities:

- `KEEP` : any stand-alone transition may happen alone (glued with some ϵ transition) or glued with any other stand-alone transition.
- `ALONE` : as in LOTOS, any stand-alone transition has to happen alone.
- `LOOSE` : stand-alone transitions are discarded (default).

Then, we have the following retrieval rules (implemented in [11]).

Initial states are built from couples of indexed initial states of the dynamic and the static views (denoted by $\langle\ s.s_{0_s}/d.s_{0_d}\ \rangle$) that verify the $\varPhi$ and $\varPhi_0$ components of the ESV structure.

$$\frac{s_{0_s} \in S_0(Obj_s), s_{0_d} \in S_0(Obj_d), \langle\ s.s_{0_s}/d.s_{0_d}\ \rangle \models \varPhi \wedge \varPhi_0}{\langle\ s.s_{0_s}/d.s_{0_d}\ \rangle \in S_0}\ INIT$$

The set of states is built from the initial states ($STATE_0$ rule) and the target states of the transitions ($STATE_1$ rule).

$$\frac{s \in S_0}{s \in S}\ STATE_0 \qquad\qquad \frac{s \in S^4,\ s \xrightarrow{l} s' \in T}{s' \in S}\ STATE_1$$

A couple of transitions (one from the static view, the other from the dynamic view) that satisfies some couple of transition formulas from the $\varPsi$ component of the ESV structure are glued into a composed transition (denoted by $\langle\ s.s_s/d.s_d\ \rangle \xrightarrow{[\langle\ s.g_s/d.g_d\ \rangle]\ \langle\ s.l_s/d.l_d\ \rangle\ [\langle\ s.gio_s/d.gio_d\ \rangle]} \langle\ s.s'_s/d.s'_d\ \rangle$) if the corresponding composed (glued from the two transition source states) source state ($\langle\ s.s_s/d.s_d\ \rangle$) is a valid state and if the corresponding composed target state ($\langle\ s.s'_s/d.s'_d\ \rangle$) satisfies the $\varPhi$ component of the ESV structure. The composition of states and transitions uses indexing in order to keep the system structure.

[4] This is redundant with the $TRANSACTION$ rules.

$$\frac{\begin{array}{c} t_s = s_s \xrightarrow{[g_s]\ l_s\ [gio_s]} s'_s \in T(Obj_s), \\ t_d = s_d \xrightarrow{[g_d]\ l_d\ [gio_d]} s'_d \in T(Obj_d), \\ \exists(\psi_{i_s}, \psi_{i_d}) \in \psi \mid ts \models \psi_{i_s}\ ,\ td \models \psi_{i_d}, \\ s = \langle\ s.s_s/d.s_d\ \rangle, s' = \langle\ s.s'_s/d.s'_d\ \rangle, \\ g = \langle\ s.g_s/d.g_d\ \rangle, gio = \langle\ s.gio_s/d.gio_d\ \rangle, l = \langle\ s.l_s/d.l_d\ \rangle, \\ s \in S\ ,\ s' \models \phi \end{array}}{s \xrightarrow{[g]\ l\ [gio]} s' \in T} \quad \begin{array}{r} TRANSACTION_L \\ \texttt{(LOOSE)} \end{array}$$

Two other rules take into account the specific `KEEP` and `ALONE` semantics.

The first one is used in conjunction with the `LOOSE` rule to take into account the `KEEP` semantics.

$$\frac{\begin{array}{c} t_s = s_s \xrightarrow{l_s} s'_s \in T(Obj_s) \cup \{\epsilon\}, \\ t_d = s_d \xrightarrow{l_d} s'_d \in T(Obj_d) \cup \{\epsilon\}, \\ \nexists(\psi_{i_s}, \psi_{i_d}) \in \psi \mid t_s \models \psi_{i_s}, \\ \nexists(\psi_{i_s}, \psi_{i_d}) \in \psi \mid t_d \models \psi_{i_d}, \\ s = \langle\ s.s_s/d.s_d\ \rangle, s' = \langle\ s.s'_s/d.s'_d\ \rangle, l = \langle\ s.l_s/d.l_d\ \rangle, \\ s \in S\ ,\ s' \models \phi \end{array}}{s \xrightarrow{l} s' \in T} \quad \begin{array}{r} TRANSACTION_K \\ \texttt{(KEEP)} \end{array}$$

The second one is used in conjunction with the `LOOSE` rule to take into account the `ALONE` semantics.

$$\frac{\begin{array}{c} t_s = s_s \xrightarrow{l_s} s'_s \in T(Obj_s) \cup \{\epsilon\}, \\ t_d = s_d \xrightarrow{l_d} s'_d \in T(Obj_d) \cup \{\epsilon\}, \\ \nexists(\psi_{i_s}, \psi_{i_d}) \in \psi \mid t_s \models \psi_{i_s}, \\ \nexists(\psi_{i_s}, \psi_{i_d}) \in \psi \mid t_d \models \psi_{i_d}, \\ s = \langle\ s.s_s/d.s_d\ \rangle, s' = \langle\ s.s'_s/d.s'_d\ \rangle, l = \langle\ s.l_s/d.l_d\ \rangle, \\ s \in S\ ,\ s' \models \phi \\ t_s = \epsilon \vee t_d = \epsilon \end{array}}{s \xrightarrow{l} s' \in T} \quad \begin{array}{r} TRANSACTION_A \\ \texttt{(ALONE)} \end{array}$$

The rules for the $A_T = (A', G, V, \Sigma_T, Ax_T, \overline{A})$ part (from [3]) are:

- $A' = A'(ESV) \cup I_{G_i} \bigcup_i A'(Obj_i(ESV))$, I_{G_i} being instantiations I_i of $G(Obj_i(ESV))$
- $G = \bigcup_i i.G(Obj_i(ESV))$, removing instantiated I_i of $G(Obj_i(ESV))$
- $V = \bigcup_i i.V(Obj_i(ESV))$, removing instantiated I_i of $V(Obj_i(ESV))$
- $\Sigma_T = \bigcup_i i.\Sigma(Obj_i(ESV))$
- $Ax_T = Ax_\Theta \bigcup_i i.Ax(Obj_i(ESV))$. Some renaming and indexing has to be done on the axioms (see [3]).
- $\overline{A} = \overline{A} \bigcup_i \overline{A}(Obj_i(ESV))$

These rules use indexing to build a product (static) type for A_T.

ESV Semantics. The semantics is given in an operational and rewriting way. The semantic function profile is: $Term \times \mathit{\Gamma} \times STATE \times EVENT \rightarrow Term \times \mathit{\Gamma} \times STATE$ where $EVENT$ is a communication event (LOTOS-like offer). It is written: $\boxed{t, \mathit{\Gamma}, s} \xrightarrow{e} \boxed{t', \mathit{\Gamma}', s'}$. Here environments ($\mathit{\Gamma}$) bind terms to variables.

Components are initially instantiated. This is denoted by the application of some $*$ generator to $\vec{t}$ terms. Here $\vec{v}$ denotes a vector v of terms and $\overset{Ax,\mathit{\Gamma}}{\downarrow}(t)$ the normal form of term t, using the Ax axioms in the $\mathit{\Gamma}$ environment.

$$\textbf{if } \langle\, s_s/s_d \,\rangle \in S_0 \textbf{ then } \xrightarrow{*(\vec{t}\,)} \boxed{\overset{Ax_T}{\downarrow}(\vec{t}\,), \{\}, \langle\, s_s/s_d \,\rangle}$$

Then, transitions may happen. $\triangleleft$ denotes the overriding union of environments and $unify$ is a function that unifies two terms and returns an environment in which both terms are equal (substitution). Finally, $\overline{c_d}$ denotes the event that corresponds (through communication) to c_d. We use suffixes to denote inputs ($\imath$) and outputs (o) and indices to denote static ($._s$) and dynamic ($._d$) parts.

$$\begin{gathered}
\textbf{if} \\
\langle\, s_s/s_d \,\rangle \xrightarrow{[\langle\, g_s/g_d \,\rangle]\ \langle\, c_s\ \vec{\imath_s}\ \vec{o_s}/c_d\ \vec{\imath_d}\ \vec{o_d} \,\rangle\ [\langle\, gio_s/gio_d \,\rangle]} \langle\, s'_s/s'_d \,\rangle \in T \\
\sigma_{event} = unify(c_d\ \vec{\imath_d}\ \vec{o_d}\ ,\ \overline{c_d}\ \vec{a\imath}\ \vec{ao})\ \wedge\ \sigma_{event} \neq \emptyset \\
\sigma_{s/d} = unify(c_s\ \vec{\imath_s}\ \vec{o_s}\ ,\ c_d\ \vec{\imath_d}\ \vec{o_d})\ \wedge\ \sigma_{s/d} \neq \emptyset \\
\mathit{\Gamma}' = \mathit{\Gamma} \triangleleft \sigma_{event} \triangleleft \sigma_{s/d} \\
\overset{Ax_T,\mathit{\Gamma}'}{\downarrow}(g_s) \wedge \overset{Ax_T,\mathit{\Gamma}'}{\downarrow}(g_d) \wedge \overset{A',\mathit{\Gamma}'}{\downarrow}(gio_s) \wedge \overset{A',\mathit{\Gamma}'}{\downarrow}(gio_d) \\
\textbf{then} \\
\boxed{t\ ,\ \mathit{\Gamma}\ ,\ \langle\, s_s/s_d \,\rangle} \xrightarrow{\overline{c_d}\ \vec{a\imath}\ \vec{ao}} \boxed{\overset{Ax_T,\mathit{\Gamma}'}{\downarrow}(c_s(t, \vec{\imath_s}, \vec{o_s}))\ ,\ \mathit{\Gamma}'\ ,\ \langle\, s'_s/s'_d \,\rangle}
\end{gathered}$$

This rule expresses that, whenever a transition that has the current state as its source state may be applied then it is applied, obtaining new (current) term and environment. The definition of applicability is:

- there is an event that unifies with the transition;
- after unifying static and dynamic transitions, all guards are true.

5 Conclusion

We presented here the different views that compose our model. In our unifying approach, views are used to describe the different aspects of a component. Hence, there are static and dynamic views. This corresponds to a first structuring level that we call internal structuring view. A more complex structuring level is the external structuring view, dealing with collections of communicating components. We formally define all these concepts.

The goal of this paper is to define a global semantics for external structuring views. An external structuring view may contain one or more global semantics components (*i.e.* integration views or composition views). An ESV has a data part, a glue part and a composition layer. The glue part is a triple of formulas: a state formula, a transition formula and a state initial formula. Therefore, we define logical rules for a state or a transition to verify a formula. The ESV composition layer is an indexed family of objects views.

The definition of a global semantics for such an ESV is done in two steps. First, we explain how we may obtain a view structure from an ESV. ESV view structures are a special kind of view where states (resp. transitions) are built as an indexed compound of states (transitions). This indexing enables us to keep the system internal structure while "gluing" components together. This gluing is used in order to specify how the states and transitions of the static and dynamic aspects are related. Then, we give the semantics in an operational style, based on rewriting over the state and the dynamic parts. The main rule states how, from a given system state (a state, a term and an environment), an event rewrites the system state. Another way could have been to give directly a semantics from the ESV structure.

Future work will deal with a proof that the semantics diagram commutes and the definition of an adequate model-checking procedure for our view structures.

References

1. André Arnold. *Systèmes de transitions finis et sémantique des processus communicants.* Etudes et recherches en informatique. Masson, 1992.
2. Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–29, January 1988.
3. Christine Choppy, Pascal Poizat, and Jean-Claude Royer. Control and Datatypes using the View Formalism. Research Report 188, IRIN, 1999. /papers/rr188.ps.gz in Poizat's web page.
4. Jan Ellsberger, Dieter Hogrefe, and Amardeo Sarma. *SDL : Formal Object-oriented Language for Communicating Systems.* Prentice-Hall, 1997.
5. M. Hennessy and H. Lin. Symbolic Bisimulations. *Theoretical Computer Science*, 138(2):353–389, 1995.
6. ISO/IEC. LOTOS: A Formal Description Technique based on the Temporal Ordering of Observational Behaviour. ISO/IEC 8807, International Organization for Standardization, 1989.
7. Leslie Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.
8. Radu Mateescu. *Vérification des propriétés temporelles des programmes parallèles.* PhD thesis, Institut National Polytechnique de Grenoble, 1998.
9. Radu Mateescu and Hubert Garavel. XTL: A Meta-Language and Tool for Temporal Logic Model-Checking. In *International Workshop on Software Tools for Technology Transfer STTT'98*, 1998.
10. José Meseguer. Rewriting logic as a semantic framework for concurrency: a progress report. In *CONCUR'96 : Concurrency Theory*, volume 1119 of *Lecture Notes in Computer Science*, pages 331–372, Pisa, Italy, 1996.
11. Gaël Nedelec, Marielle Papillon, Christelle Piedsnoirs, and Gwen Salaün. CLAP: a Class Library for Automata in Python. Master's thesis, 1999. CLAP.html in Poizat's web page.

12. Pascal Poizat, Christine Choppy, and Jean-Claude Royer. Un support méthodologique pour la spécification de systèmes "mixtes". Research Report 180, IRIN, Novembre 1998. /papers/rr180.ps.gz in Poizat's web page.
13. Pascal Poizat, Christine Choppy, and Jean-Claude Royer. Concurrency and Data Types: A Specification Method. An Example with LOTOS. In J. Fiadeiro, editor, *Recent Trends in Algebraic Development Techniques, Selected Papers of the 13th International Workshop on Algebraic Development Techniques WADT'98*, volume 1589 of *Lecture Notes in Computer Science*, pages 276–291, Lisbon, Portugal, 1999. Springer-Verlag.
14. Pascal Poizat, Christine Choppy, and Jean-Claude Royer. From Informal Requirements to COOP: a Concurrent Automata Approach. In J.M. Wing, J. Woodcock, and J. Davies, editors, *FM'99 - Formal Methods, World Congress on Formal Methods in the Development of Computing Systems*, volume 1709 of *Lecture Notes in Computer Science*, pages 939–962, Toulouse, France, 1999. Springer-Verlag.
15. Gianna Reggio and Mauro Larosa. A graphic notation for formal specifications of dynamic systems. In John Fitzgerald, Cliff B. Jones, and Peter Lucas, editors, *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods (Proc. 4th Intl. Symposium of Formal Methods Europe, Graz, Austria, September 1997)*, volume 1313 of *Lecture Notes in Computer Science*, pages 40–61. Springer-Verlag, September 1997. ISBN 3-540-63533-5.
16. Carron Shankland, Muffy Thomas, and Ed Brinksma. Symbolic Bisimulation for Full LOTOS. In *Algebraic Methodology and Software Technology AMAST'97*, volume 1349 of *Lecture Notes in Computer Science*, pages 479–493. Springer-Verlag.
17. Graeme Smith. A Semantic Integration of Object-Z and CSP for the Specification of Concurrent Systems. In J. Fitzgerald, C.B. Jones, and P. Lucas, editors, *Formal Methods Europe (FME'97)*, volume 1313 of *Lecture Notes in Computer Science*, pages 62–81. Springer-Verlag, 1997.
18. Rational Software. Unified Modeling Language, Version 1.1. Technical report, Rational Software Corp, http://www.rational.com/uml, September 1997.
19. Martin Wirsing. *Algebraic Specification*, volume B of *Handbook of Theoretical Computer Science*, chapter 13, pages 675–788. Elsevier, 1990. Jan Van Leeuwen, Editor.

A The Password Manager Specification

We give in this appendix the full KORRIGAN specification of the password manager case study.

A.1 The Static View of the Password Manager

The static view of the password manager in KORRIGAN is given in Figure 12. We have made here the choice to define it "from scratch". Another approach could have been to define it as an instantiation of static view of sets (see [3] appendix for an example of this).

```
                 STATIC VIEW StaticPasswordManager (SPM)
                            SPECIFICATION
imports Boolean, UserId, Password
ops
    empty : → SPM
    add : SPM, UserId, Password → SPM
    modify : SPM, UserId, Password → SPM
    declared : SPM, UserId → Boolean
    correct : SPM, UserId, Password → Boolean
axioms
 modify(empty,u,p) == add(empty,u,p);
 (u = u2) ⇒ modify(add(spm,u2,p2),u,p) == add(spm,u,p);
 not(u = u2) ⇒ modify(add(spm,u2,p2),u,p) == add(modify(spm,u,p),u2,p2);
 declared(empty,u) == false;
 (u = u2) ⇒ declared(add(spm,u2,p2),u) == true;
 not(u = u2) ⇒ declared(add(spm,u2,p2),u) == declared(spm,u);
 correct(empty,u,p) == false;
 (u=u2) ⇒ correct(add(spm,u2,p2),u,p) == (p = p2);
 not(u=u2) ⇒ correct(add(spm,u2,p2),u,p) == correct(spm,u,p);
                                  STS
  .→X
  X − add ?u : UserId ?p : Password → X
  X − modify ?u : UserId ?p : Password → X
  X − declared ?u : UserId !b : Boolean → X
  X − correct ?u : UserId ?p : Password !b : Boolean → X
```

Fig. 12. The static view of the Password Manager

Figure 13 gives a graphical representation for STS part of the static view of the password manager.

A.2 The Dynamic View of the Password Manager

The dynamic view of the password manager is made up of two distinct activities: creation and modification of passwords. Figure 14 gives a graphical represen-

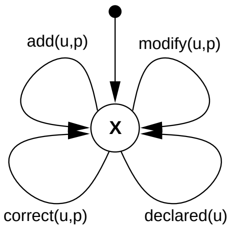


Fig. 13. A graphical representation of the Password Manager static view STS

tation of it with boxes denoting activities and dashed lines denoting abstract transitions.

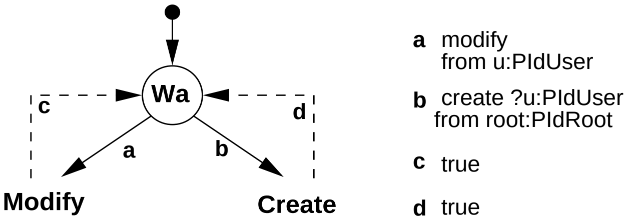


Fig. 14. The dynamic view of the Password Manager

The Modify activity. Figure 15 gives a graphical representation of the password modification activity. It is important to note that, to the contrary of the STS of components, the STS of activities contain end states.

The Create activity. Figure 16 gives a graphical representation of the password creation activity.

Integration of the activities. Figure 17 gives a graphical representation for the STS part of the dynamic view of the password manager resulting from the activity integration.

A.3 The Integration View of the Password Manager

The password manager integration view (in terms of static and dynamic parts) in Korrigan is given in Figure 18. The glue is made up of two parts: one gluing dynamic guards to static operations, and one gluing static and dynamic transitions.

The first glue part is defined in the `axioms` clause. The first axiom is related to the condition of validity of a given user when modifying a password. The

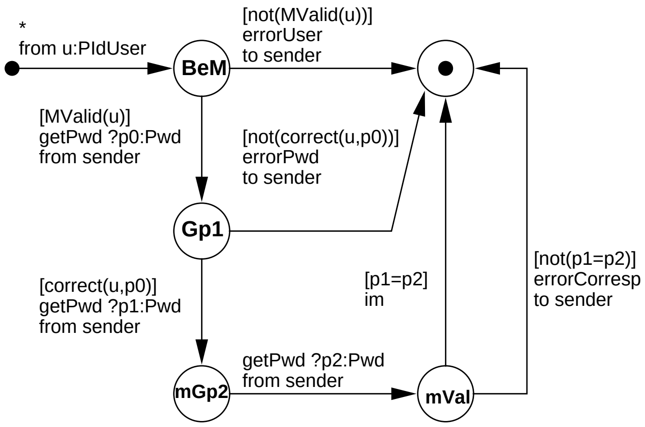


Fig. 15. A graphical representation of the Modify activity

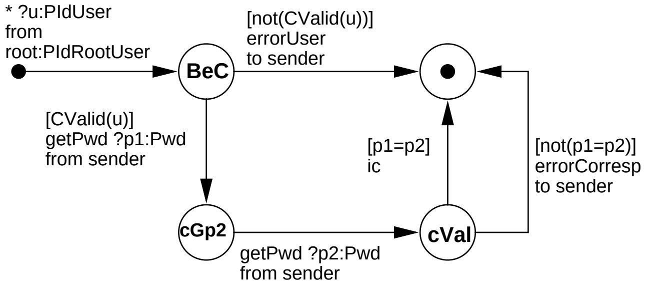


Fig. 16. A graphical representation of the Create activity

user obtained through the dynamic part has to be declared in the static part. The second axiom is the counterpart of the first axiom as far as user creation is considered. The new user must not be declared in the static part. The last axiom states that the condition of correctness (between user identification and user password) used in the dynamic part corresponds to the one defined in the static part.

The second glue part is defined in the `with` clause. It role is to glue the (hidden) dynamic operations `im` and `ic` with (respectively) the static modification and creation operations. The correspondence between parameters is achieved through the glue.

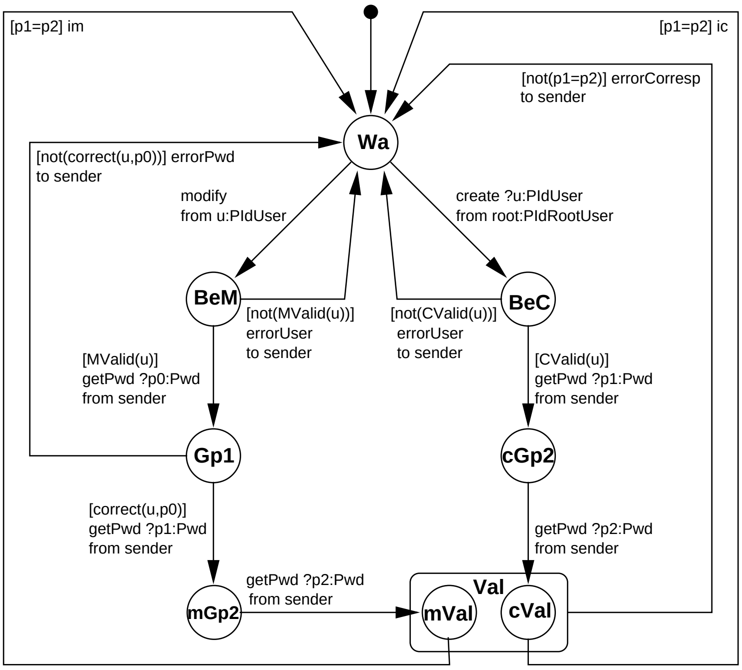


Fig. 17. A graphical representation of the Password Manager dynamic view STS

INTEGRATION VIEW Password Manager
COMPOSITION ALONE

is
    STATIC *s*: SPM
    DYNAMIC *d*: RDPM
axioms
    d.MValid(s,u) == s.declared(s,u)
    d.CValid(s,u) == not(s.declared(s,u))
    d.correct(s,u,p) == s.correct(s,u,p)
with true, {
    (d.([true] im), s.(modify(d.u,d.p1))),
    (d.([true] ic), s.(add(d.u,d.p1)))
}
initially true

Fig. 18. The integration view of the Password Manager