Control and Datatypes using the View Formalism (Long Version)

last revision: February 9, 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 herein deal with mixed specification formalisms, *i.e.* formalisms with both a static (data types) and a dynamic (behaviour) part. Our formalism is based on *symbolic transition systems* (STS) [9], that allow one to specify systems at an abstract level and to avoid state explosion. STS are a kind of guarded finite state/transition diagrams where states and transitions are labelled with open terms.
Both dynamic and static parts of objects are specified, in a unifying approach, as formal structures that we call *views*. These components interpretation structures use STS, and we show how these may be derived from their view structures.
The system is structured by means of collections of objects (with identities). A temporal logic is used to glue the components altogether and expresses a generalized form of synchronous product [1]. We then show how a view structure and its interpretation structure may be obtained.
The formalism is explained using a simplified phone service example.
Keywords: mixed specifications, method, formalism, symbolic transition systems, object-orientation.

1 Introduction: on Formalism Requirements

We herein deal with mixed specification formalisms, *i.e.* formalisms with a part for both the static (data types) and the dynamic (behaviour) parts of the systems such as LOTOS [11], SDL [7], Estelle [10], UML [22], Raise [23] or extension of non mixed formalisms as ObjectZ/CSP [21]. One of the aims of this work is to provide an integration of both static and dynamic parts of the specification which is achieved through a formal structure that we call *views*. Another aim is to allow for the structuring of specifications. One of the interesting points in this work being to provide a framework for the combination of both static-dynamic integration and structuring. Moreover, we agree with [19] who think object-oriented concepts (reuse, encapsulation, inheritance for example) are as

important for specification as they are for programming. Therefore, our work takes into account three main concerns: Concurrency (control, communication, behaviour), Data types and Object-orientation.

Defining or choosing a new specification formalism [3], one should care to keep a good balance between three axis : Readability - or user-friendliness, Expressiveness and Abstraction. It is possible to define planes by combining two axis such as the Power (expressiveness and abstraction) and the Notation (readability and abstraction) planes.

We call "power" a good adequation between abstraction and expressiveness levels that deals with the ability to do proofs and generate code for the specifications. To end, let us agree with [2] that formalisms and methods should be closely related.

Our formalism (and the corresponding specification language called KORRIGAN) is intended to be used at a design level (between the end of the requirement analysis, that may be done in UML for example, and the code generation). A method associated to our formalism has been defined in [16] and subsequently used to generate LOTOS and SDL specifications. In [17] we explained how to generate object-oriented code from our specifications, while in this paper we shall deal with the formal aspects of our specification method.

In this paper, we present the different view structures that compose our model. The use of STS is of great interest as it enables one to define components in a very abstract and readable way. Therefore, we give the mappings between the two different structures (using or not STS) possible for our views. Then, we define inheritance for basic views (modelling static and dynamic aspects).We also define how a view structure may be retrieved for the composition of several components views. This retrieval is based (i) on the retrieval of the algebraic part of view structures, and (ii) the computation of a global STS for the composition. We illustrate our view model on a phone service example.

2 Case Study

Our formalism is explained using a simplified phone service example. The specification will be sketched in the following Section. The full specification may be found in Appendix A.

The EasyPhone society wants to develop a computerised phone service on a distributed system. This phone service should enable the clients communication through a central server. The requirements are the following.

2.1 The clients

A client or `USER` is identified by a 10 digits personal number. A client may have two possible statutes: (s)he may be a simple client or a subscriber.

When a communication takes place, the *caller* is the one who requested the communication, and the *party called* is the one who is called. A *potential called party* is a client who may be communicating with a calling subcriber.

The client activities list depends on whether (s)he is a ordinary client or a subscriber.

The ordinary client.

1. (S)he may ask the server to become a subscriber only when not involved in a communication.
2. (S)he may be called by the server when a calling subscriber requests a communication.
3. (S)he may refuse a communication request.
4. (S)he may accept a communication request with a calling subscriber.
5. (S)he may interrupt the communication (end of transmission).

The subscriber. A subscriber has the same rights as a client, and in addition:

1. (S)he may go back to the ordinary client statut only when not requesting or not involved in a communication.
2. (S)he may ask the server to register another client as one of his potential called parties.
3. (S)he may ask the server to delete one client from his(her) potential called parties.
4. (S)he may ask the server to start a communication with another client.
5. (S)he may withdraw a communication request previously made.
6. Once the communication is taking place, (s)he may interrupt it at any time.

2.2 The central server

The central server is composed with four units : a connection unit (UC), a management unit (UM), a disconnection unit (UD) and a database (UDB).

The management unit.

1. This unit handles all status change requests (ordinary client/subscriber).
2. It manages all the updates of subscribers potential called parties.

The connection unit.

1. This unit handles all communication requests
2. It sends a request to the party called to indicate that a communication request to him/her was issued.
3. It establishes the connection between the caller and the party called.

The disconnection unit.

1. This unit handles the disconnection requests.
2. The central server may decide to interrupt a connection.

The database unit.

1. This database contains all subscribers numbers, and for each subscriber its potential called parties.
2. The database unit registers all statut changes requests (ordinary client/subscriber), and all potential called party modifications
3. The database unit provides a knowledge on which are the current communications, and who are the caller and the party called.

A communication between a caller i and a party called j is possible when:

1. The caller has to be a subscriber (but this is not necessary for the party called).
2. i $\neq$ j.
3. j has to be registered as a potential party called by i.
4. A communication between i and j is not already taking place.
5. i first issued a communication request for j.

3 The View Model

A specification framework. We will here present the different views that compose our view model. We will first present two concepts our model has the need for: identifiers and symbolic transition systems.

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

Symbolic Transition Systems. Symbolic transition systems are a form of guarded automata with free variables appearing in guards and transitions. Our STS originate from symbolic transition graphs [9].

Definition 1 (Symbolic Transition Graphs). *A symbolic transition graph is a directed graph in which every node n is labeled by a set of free variables $fv(n)$ and every branch is labeled by a guarded action (a boolean guard b and an action α that may bind variables - denoted by bv) such that if a branch labeled by (b, α) goes from node m to n, [...] then $fv(b) \cup fv(\alpha) \subseteq fv(m)$, and $fv(n) \subseteq fv(m) \cup bv(\alpha)$. The set of bound and free variables of the actions are defined in the obvious manner: $fv(c!e) = fv(e)$, $bv(c?x) = \{x\}$ and otherwise both $fv(\alpha)$ and $bv(\alpha)$ are empty.*

STS enable one to reduce state-explosion problems by providing a good (high) level of abstraction.

Suppose that we have an unbounded buffer of naturals with operations put and get (only when the buffer is not empty). With the usual transition systems approach, there is a state explosion since the reception variables and the process parameters are instantiated (Figure 1); both vertical (due to receptions - there is an infinity of different naturals to be received) and horizontal (due to process parameters - there is an infinity of buffer contents - size and/or values in it).

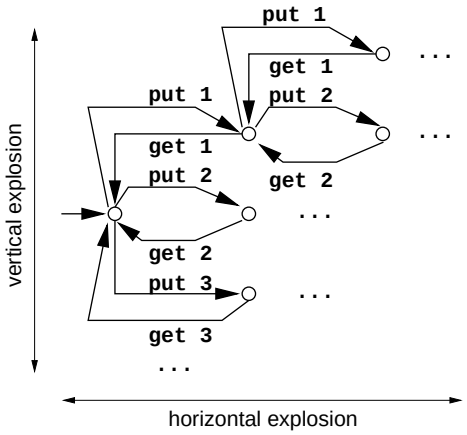


Fig. 1. Unbounded buffer state explosion with (usual) transition systems

With STS, one may abstract away from the buffer contents and receptions, and focus only on the fact that the buffer is empty or not (Figure 2). To quote [20] we may say that "STS are transition systems which separate the data from the process behaviour".

Moreover, STS are a very readable formalism and ease the work on our views structures (by defining them using operations on states and transitions).

However, whereas automata and closed transition systems are a commonly accepted notation that are equipped with tools for design or verification – see

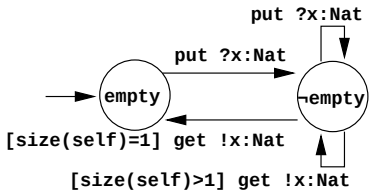


Fig. 2. Unbounded buffer with symbolic transition systems

[8, 14] for example – STS are not yet equipped with tools such as those dealing with temporal model checking [18, 13, 12] or bisimulations [9, 20].

3.1 The Different View Classes

We define an *Internal Structuring View* abstraction 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).

We therefore have here a first structuring level (*internal structuring*) for objects. Another structuring level is the *external structuring*, 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*).

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 3.

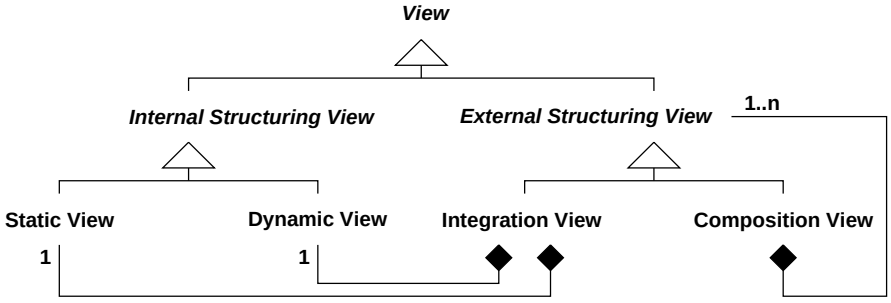


Fig. 3. The View model class diagram

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

Definition 2 (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 3 (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.*

Definition 4 (Dynamic Signatures). *Dynamic signatures are given in a LOTOS-like form [4],* 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 `comm` gate)
  constructor `comm : T, Msg, PId → T`
- `comm !m:Msg to u:Pid`
  (emission of a message `m` to `u` on the `comm` gate)
  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. Hence, with each operation, its preconditions and postconditions will be associated.

Definition 5 (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.

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 6 (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 4. The `SPECIFICATION` parts 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).

An STS (S, S_0, T) may be obtained in a unique way from the abstraction and operation parts.

- $S = \{s \in 2^{|C|} | \forall \phi \in \Phi . s \models \phi\}$
- $S_0 = \{s_0 \in S | s_0 \models \Phi_0\}$
- $\forall o \in O - \overline{O}, \forall s.s \models o.P_C, \forall s'.s' \models o.Q_C \ . \ t : s \xrightarrow{[o.Q_{Cl}] \ \{o\} \ [o.P_{io} \wedge o.Q_{io}]} s' \in T$
- $\forall o \in \overline{O}, \forall s.s \models o.P_C, \forall s'.s' \models o.Q_C \ . \ t : s \xrightarrow{[o.Q_{Cl}] \ i_{\{o\}} \ [o.P_{io} \wedge o.Q_{io}]} s' \in T$

State are composition of conditions that satisfies the Φ relations between conditions. Initial states are states satisfying the initial formula given in Φ_0. Finally, the transitions are retrieved automatically using the pre and postconditions.

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

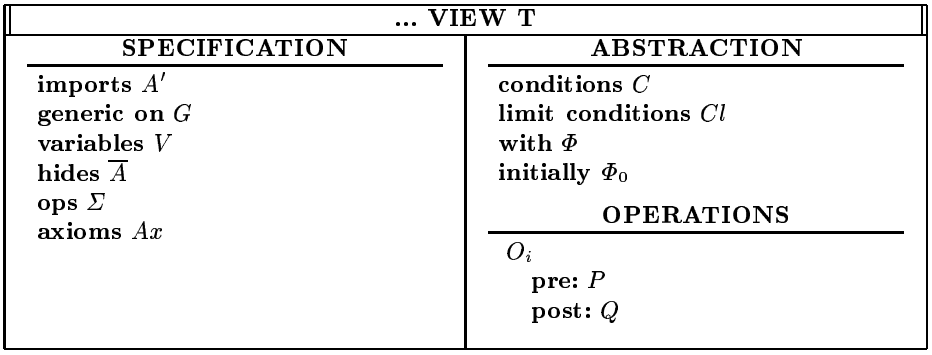


Fig. 4. The KORRIGAN syntax (views)

Conversely, trivial abstraction and operation parts may then be derived from the STS. The idea is to give one condition for each state, consider these conditions as being exclusive (using Φ), and then define the pre and postconditions using them.

Therefore, views may be defined in an alternative way using either Symbolic Transition Systems (STS) or the abstraction and operation parts. The alternative KORRIGAN syntax is given in Figure 5. Note that STS parts of views are given using dynamic signatures (see Definition 4). These parts are a textual representation of the STS (STS have also a graphical - *i.e.* a more readable - representation, see Figure 6 for example).

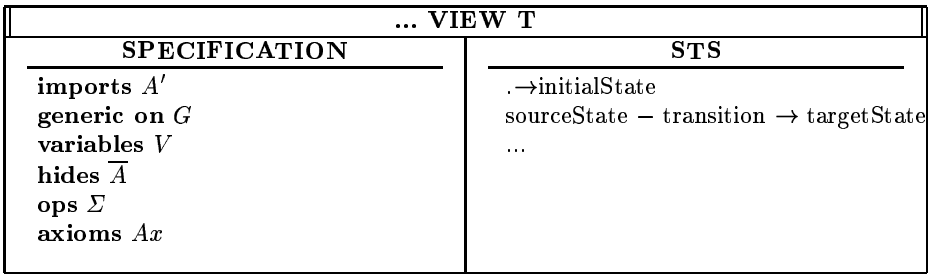


Fig. 5. The KORRIGAN alternative syntax (views)

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

– no full algebraic specification (only signatures matter)

– the `from/to` elements (using PId types)

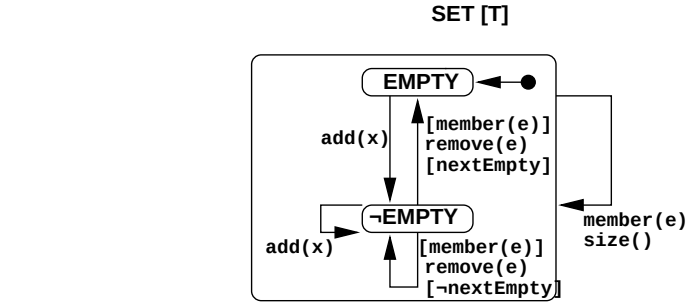
This goes with the fact that dynamic views focus only on the communication abstraction.

Static views. Static views (SV) describe the static aspect of a component.

Definition 7 (Static view). *A* static view SV_T *of a component* T *is a view.*

The `FULL-USER` static view corresponds to a (basic) set of known `USERS` (PIdUser). The `SET` static view is given in Figure 6. It defines the usual set operations: `member`, `size`,... It is abstracted as being either `empty` or not (using a corresponding condition).

The `FULL-USER` dynamic view will be defined as an extension (Fig. 10) of the `BASIC-USER` dynamic view (Fig. 7).



STATIC VIEW SET

SPECIFICATION

imports NAT
generic on T
ops
new : → SET
add : SET × T → SET
member : SET × T → BOOL
size : SET → NAT
nextEmpty : SET → BOOL
remove : SET × T → SET
empty : SET × BOOL
axioms
nextEmpty(s) == size(s)=1;
member(new,e) == false;
...

ABSTRACTION

conditions empty
limit conditions nextEmpty
initially empty

OPERATIONS

member, size
pre: true
post: {empty':empty}
add
pre: true
post: {empty':false}
remove
pre: ¬ empty ∧ member(e)
post: {empty':nextEmpty}

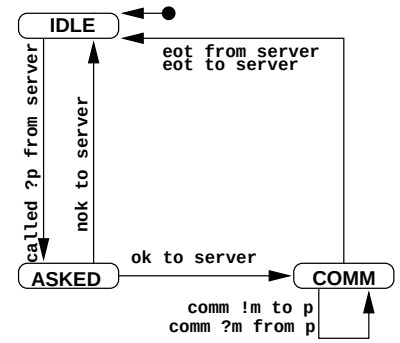
Fig. 6. The `SET` Static View

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 emitters 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.

Definition 8 (Dynamic view). *A* dynamic view DV_T *of a component* T *is a view where:*

- Σ_T *are dynamic signatures for sort* T
- $Ax_T = \Phi \cup \Phi_0$

The BASIC-USER dynamic view is given in Figure 7.



DYNAMIC VIEW D-BASIC-USER

SPECIFICATION

imports PId, MSG
generic on server:PIdServer
ops ...

STS

• → IDLE
IDLE – called ?p:PIdUser **from** server →ASKED
ASKED – ok **to** server→COMM
ASKED – nok **to** server→IDLE
COMM – comm ?m:Msg **from** p→COMM
COMM – comm !m:Msg **to** p→COMM
COMM – eot **to** server→IDLE
COMM – eot **from** server→IDLE

Fig. 7. The BASIC-USER Dynamic View

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 9 (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 10 (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 11 (ESV glue part). *An ESV* glue part Θ *(in an ESV structure (A_T, Θ, K_T)) is a tuple (Φ, Ψ, Φ_0) where:*

- *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 $\vdash \Phi_0 \Rightarrow \bigwedge_{id} id.\Phi_0(Obj_{id})$).*

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 $AG \bigwedge_i \psi_{i_s} \Leftrightarrow \psi_{i_d}$. State and transition formula are simple ones: there are no fixpoint or special modalities like in more expressive temporal logics such as CTL, μCRL or XTL [14]. 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 [13] approach). State and transition formula are defined in [5].

Definition 12 (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.*

Issues on ESVs, and their global semantics (components having all aspects defined) are addressed in [5]. Here, we focus only on the obtaining of views structures for compositions of components. Hence, Integration and Composition Views are out of the scope of this paper (see [5]). The syntax for external structuring views in KORRIGAN is given in Figure 8.

An example of ESV is given in Figure 9 where we define the `SERVER` as the composition the four units (defined in 2.2).

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

[3] Where AG denotes "always globally".

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

Fig. 8. The KORRIGAN syntax (external structuring views)

EXTERNAL STRUCTURING VIEW SERVER	
COMPOSITION	
is	
uc : UC [udb]	; connection unit
um : UM [udb]	; management unit
ud : UD [udb]	; disconnection unit
udb : UDB [uc,ud,um]	; database unit
with true,{	
(ud.askSomeIn, udb.askSomeIn)	
(ud.getSomeIn ?p:PIdUser, udb.getSomeIn !p:PIdUser)	
...}	
initially true	

Fig. 9. The SERVER ESV

3.2 Inheritance of Views.

By *inheritance* we only mean a basic form of inheritance that consists in adding new conditions and new operations (that may use these conditions) in static or dynamic views. There is no overriding.

Furthermore, the subclass still has the superclass operations (and conditions). For these reasons, we will have new states for each combination of the new conditions in the subclass. These states are hierarchical states in which the STS of the superclass is implicitly present (only new transitions are explicitly given).

This form of inheritance is actually quite simple indeed but of practical use and well defined.

As an example, with the phone service, we have simple (basic) users that may only be called (Figure 7) and complex users that may also call other users (Figure 10).

4 View Structures for ESVs

We want to give view (either in their "normal" (A_T, α, D_T) or their alternative (A_T, STS) form) structures for ESV in order to be able to compose them in turn in ESVs. There are two ways to give view structures for ESVs (Fig. 11).

The first one is to use the ESV components view structures, and obtain (arrow A in Fig. 11) a view structure for the ESV. The second one is to use the

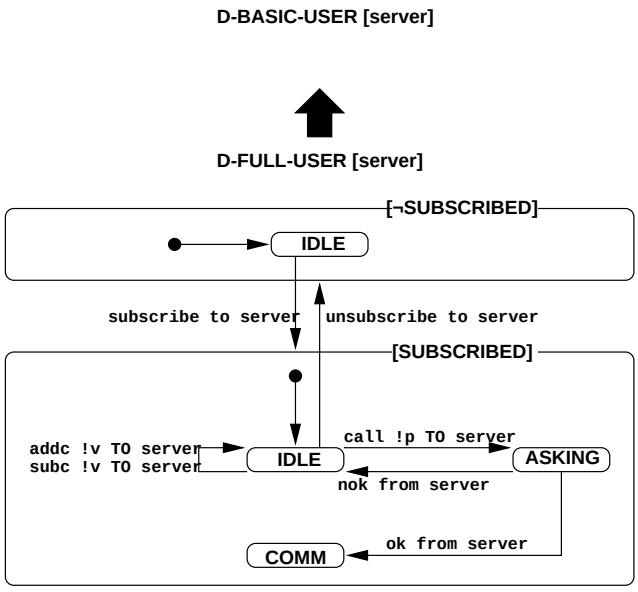


Fig. 10. The `FULL-USER` Dynamic View

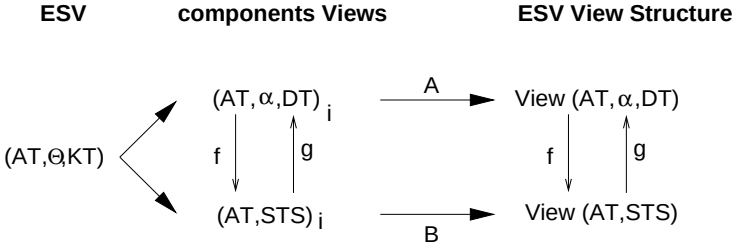


Fig. 11. Obtaining of a view structure for an ESV

ESV alternative view structures (using STS), and obtain (arrow B in Fig. 11) an alternative view structure for the ESV. Given the functions mapping view structures and alternative view structures (say f and g), the diagram should commute. Therefore, for the simplicity sake, we use the B arrow since working on the STS part is easier than working on the (α, D_T) one.

4.1 A_T part obtaining

The rules for the $A_T = (A', G, V, \Sigma_T, Ax_T, \overline{A})$ part 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 $V(Obj_i(ESV))$

- $\Sigma_T = \bigcup_i i.\Sigma(Obj_i(ESV))$
- $Ax_T = \bigcup_i i.Ax(Obj_i(ESV))$. Some renaming and indexing has to be done on the axioms.
- $\overline{A} = \overline{A} \bigcup_i \overline{A}(Obj_i(ESV))$

These rules use indexing to build a product (static) type for A_T. The renaming on the axioms consists in indexing every operation where the type of interest (the type of the component whose axioms we are indexing) is in first position ("receiver").

4.2 *STS* part obtaining

The rules for the STS part use an extension of the synchronous product [1]. The synchronous vector is replaced by the Θ part of the ESVs. These mechanisms have been implemented in [15] where generic classes for (parameterized) automata and a generic synchronous product function (parameterized by Θ) have been defined.

The rules for the $STS = (S, S_0, T)$ part are:

- $S = \{s \in \Pi_i S_i \mid s \models \Phi\}$
- $S_0 = \{s_0 \in S \mid s_0 \models \Phi_0\}$
- $T = \{t \in \Pi_i T_i \mid t \models \Psi\}$

Afterwards, one has to remove any non accessible states.

5 Conclusion

We presented a formalism to express both the static and the dynamic parts of mixed specifications. In order to provide a framework to integrate both parts, we defined a formal structure that we call *view*. We make use of *symbolic transition systems* (STS) to specify systems at an abstract level and to avoid state explosion. We also allow for the structuring of specifications by means of collections of objects (with identities). A temporal logic is used to glue the components altogether and expresses a generalized form of synchronous product [1]. One of the interesting points in this work being to provide a framework for the combination of both static-dynamic integration and structuring. We define the mappings between the two different structures (using or not STS) possible for our views and define how a view structure may be retrieved for the composition of several components views. We also define inheritance for the basic views (static and dynamic aspects).

We think this formalism is well suited to achieve a good balance between the readability, expressiveness and abstraction properties for the specifications.

In [5] we present the strong links that exist between all aspects of a single component, and that all together build its global semantics. We also provide a new set of rules for the STS part, taking into account the composition of all aspects.

A next step would be to define proof mechanisms (algebraic, model checking or tableaux) for our view-based specifications.

References

1. André Arnold. *Systèmes de transitions finis et sémantique des processus communicants.* Etudes et recherches en informatique. Masson, 1992.
2. E. Astesiano and G. Reggio. Formalism and method. In M. Bidoit and M. Dauchet, editors, *TAPSOFT'97*, volume 1214 of *Lecture Notes in Computer Science*, pages 93–114. Springer-Verlag, 1997.
3. Paul E. Black, Kelly M. Hall, Michael D. Jones, Trent N. Larson, and Phillip J. Windley. A Brief Introduction to Formal Methods. In *Proceedings of the IEEE 1996 Custom Integrated Circuits Conference*, 1996.
4. Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–29, January 1988.
5. Christine Choppy, Pascal Poizat, and Jean-Claude Royer. A Global Semantics for Views. Rapport de Recherche 189, IRIN, 1999. /papers/rr189.ps.gz in Poizat's web page.
6. Christine Choppy, Pascal Poizat, and Jean-Claude Royer. Control and Datatypes using the View Formalism. presented at WADT'99, 1999.
7. Jan Ellsberger, Dieter Hogrefe, and Amardeo Sarma. *SDL : Formal Object-oriented Language for Communicating Systems.* Prentice-Hall, 1997.
8. Jean-Claude Fernandez. *ALDEBARAN : un système de vérification par réduction de processus communicants.* Thèse de Doctorat, Université Joseph Fourier, Grenoble, Mai 1988.
9. M. Hennessy and H. Lin. Symbolic Bisimulations. *Theoretical Computer Science*, 138(2):353–389, 1995.
10. ISO/IEC. ESTELLE: A Formal Description Technique based on an Extended State Transition Model. ISO/IEC 9074, International Organization for Standardization, 1989.
11. ISO/IEC. LOTOS: A Formal Description Technique based on the Temporal Ordering of Observational Behaviour. ISO/IEC 8807, International Organization for Standardization, 1989.
12. Carron Kirkwood and Muffy Thomas. Towards a Symbolic Modal Logic for LOTOS. In *Northern Formal Methods Workshop NFM'96*, eWIC, 1997.
13. Radu Mateescu. *Vérification des propriétés temporelles des programmes parallèles.* PhD thesis, Institut National Polytechnique de Grenoble, 1998.
14. 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.
15. Gaël Nedelec, Marielle Papillon, Christelle Piedsnoirs, and Gwen Salaün. CLAP: a Class Library for Automata in Python. Projet de maîtrise informatique, Université de Nantes, 1999. http://www.sciences.univ-nantes.fr/info/recherche/mgl/FRANCAIS/ThemesetProjets/SHES/CLAP/.
16. 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.
17. 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.
18. J. Rathke and M. Hennessy. Local Model Checking for Value-Passing Processes (Extended Abstract). In Martín Abadi and Takayasu Ito, editors, *Third International Symposium on Theoretical Aspects of Computer Software TACS'97*, volume 1281 of *Lecture Notes in Computer Science*, pages 250–266. Springer-Verlag, 1997.
19. Amílcar Sernadas, Christina Sernadas, and José Felix Costa. Object Specification Logic. *Journal of Logic and Computation*, 5(5):603–630, 1995.
20. 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.
21. 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.
22. Rational Software and al. Unified Modeling Language, Version 1.1. Technical report, Rational Software Corp, http://www.rational.com/uml, September 1997.
23. The Raise Method Group. *The RAISE Development Method*. The Practitioner Series. Prentice-Hall, 1995.

A The Phone Service Specification

We give in this appendix the full KORRIGAN specification of the phone service. The phone service is specified (in a decompose-and-recompose way) following the method we developed for mixed specifications [16]. Here, we will have first some basic (common) specifications. Then, we will specify the phone service users (first the basic ones, and secondly the full users using the inheritance principle). Finally, we will specify each of the server units, their composition (the server), and the full composition (Figure 12) of both the users and the server that build the phone service. Communications between these components will be studied in Figures 37 and 38.

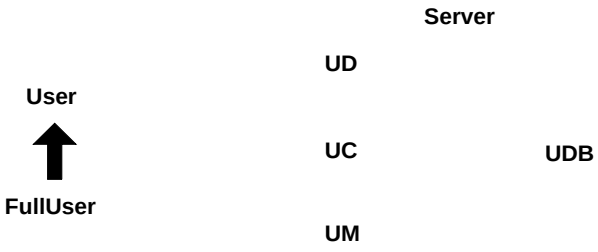


Fig. 12. The Components of the Phone Service

A.1 Basic Specifications

We will first give the specifications of some basic elements that are of current use: the empty (static) view, a (generic) static view for sets, and the algebraic specification of couples. Both are not specific to this case study, therefore we group them under the term "basic specifications".

The Static View `EMPTY`. This view (Figure 13) is used whenever the components have no static part. It is built over an IDLE initial state that enables no transitions.

STATIC VIEW S-EMPTY	
SPECIFICATION	**STS**
	• → IDLE

Fig. 13. The Static View `EMPTY`

The Static View of Sets. Figure 14 gives the static view of sets in KORRIGAN. The set operation signatures and axioms are defined. Sets are abstracted as being empty or not (using a `empty` condition), and then operations preconditions and postconditions are defined by expressing their effect on the `empty` condition. Simple (non primed) condition names in preconditions and postconditions stand for the corresponding condition value before the operation takes place. Primed condition names stand for the corresponding condition value after the operation takes place. Axioms may be obtained *from* the STS using derivation principles [6].

STATIC VIEW SET

SPECIFICATION

imports NAT
generic on T
ops
new : → SET
add : SET × T → SET
member : SET × T → BOOL
size : SET → NAT
nextEmpty : SET → BOOL
remove : SET × T → SET
empty : SET × BOOL
axioms
nextEmpty(s) == size(s)=1;
member(new,e) == false;
(e1=e) ⇒ member(add(s,e1),e)
== true;
not(e1=e) ⇒ member(add(s,e1),e)
== member(s,e);
size(new) == 0;
size(add(s,e)) == size(s)+1;
remove(new,e) == new;
(e1=e) ⇒ remove(add(s,e1))
== remove(s,e);
not(e1=e) ⇒ remove(add(s,e1))
== add(remove(s,e),e1);

ABSTRACTION

conditions empty
limit conditions nextEmpty
initially empty

OPERATIONS

member, size
pre: true
post: {empty':empty}
add
pre: true
post: {empty':false}
remove
pre: ¬ empty ∧ member(e)
post: {empty':nextEmpty}

Fig. 14. The Static View of Sets

Figure 15 gives a graphical representation for the STS part of the static view of sets. The arguments of the operations corresponding to the component (*type of interest*) are implicit. Hence, they are omitted. Since some operations (namely, the `member` and `size` operations) are defined on each state, and do not modify the conditions values (such an operation is called a *non conditioned observator* in [6]), we use a super state in order to simplify the STS graphical representation.

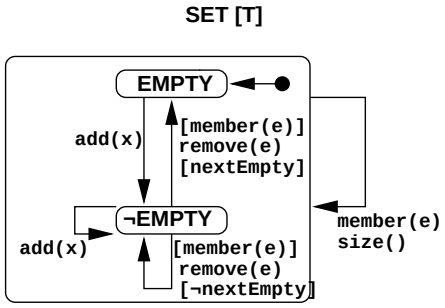


Fig. 15. A Graphical Representation of the SET STS

The Algebraic Specification of Couples. Figure 16 gives the algebraic specification of (generic) couples. This specification will be used in further KORRIGAN specifications for example when modelling couples of users being currently in communication between one another. Operations p1 and p2 correspond (respectively) to the first and second projection of the couple.

Couple	
generic on T	
imports BOOL	
ops	
(_ , _):T × T → Couple	; couple constructor
_ = _: Couple → BOOL	; equality of couples
p1 : Couple → T	; first element selector
p2 : Couple → T	; second element selector
axioms	
(a,b) = (c,d) == (a=c) ∧ (b=d);	
p1((a,b)) == a;	
p2((a,b)) == b;	

Fig. 16. The Algebraic Specification of Couples

A.2 Basic Users

The Dynamic View of Basic Users. Figure 17 gives the dynamic view of basic users in KORRIGAN.

Figure 18 gives a graphical representation for the STS part of the dynamic view of basic users.

The Integration View of Basic Users. Basic users have no static view. Figure 19 gives the integration view of basic users in KORRIGAN.

DYNAMIC VIEW D-BASIC-USER

SPECIFICATION	STS
imports PId, MSG	• → IDLE
generic on	IDLE − called ?p:PIdUser **from** server
server:PIdServer	→ASKED
ops	ASKED − ok **to** server→COMM
called ?p:PIdUser **from** server	ASKED − nok **to** server→IDLE
ok **to** server	COMM − comm ?m:Msg **from** p→COMM
nok **to** server	COMM − comm !m:Msg **to** p→COMM
comm ?m:Msg **from** PIdUser	COMM − eot **to** server→IDLE
comm !m:Msg **to** PIdUser	COMM − eot **from** server→IDLE
eot **to** server	
eot **from** server	

Fig. 17. The Dynamic View of Basic Users

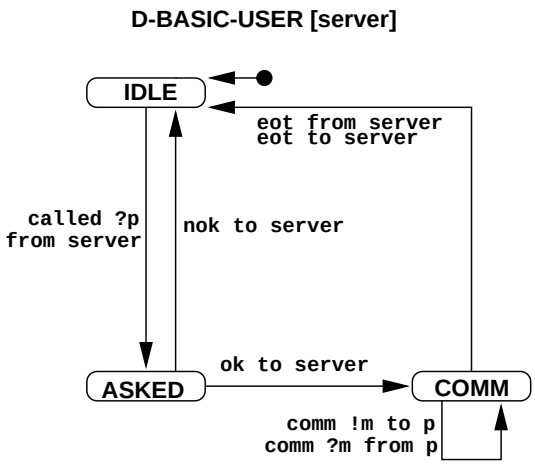


Fig. 18. A Graphical Representation of the `BASIC USER` Dynamic View STS

INTEGRATION VIEW BASIC-USER

SPECIFICATION

generic on
server:PIdServer

COMPOSITION

is
s : S-EMPTY
d : D-BASIC-USER [server]
with true, true
initially true

Fig. 19. The Integration View of Basic Users

A.3 Full Users

The Dynamic View of Full Users. Figure 20 gives the dynamic view of full users in KORRIGAN. This view inherits from the basic user dynamic view. Hence, it may only add new conditions and operations defined using these new conditions. Here, we have a new condition, `subscribed` that is true when the full user has subscribed, and false if (s)he has not.
The STS is defined using hierarchical states for each possible values of the new conditions. Here, we have only one new condition, hence the STS contains two hierarchical states (`subscribed` and `not subscribed`). Each of the hierarchical states contains the whole STS of the dynamic view its view inherits from. Hence, the states in these hierarchical states may be referred to using our usual prefix notation (*e.g.* `subscribed.IDLE` for the `IDLE` state in the `subscribed` hierarchical state). Moreover, one may use the inherited STS state names in place of the corresponding formula over the (inherited) conditions, and the state names may be used in preconditions and postconditions. For example, the `subscribe` operation preconditions states that the `subscribed` condition must be false and the status IDLE. When conditions are unchanged (*i.e.* when there is `c':c` in a postcondition for some `c` condition), we have them implicit (*i.e.* we omit it).

DYNAMIC VIEW D-FULL-USER EXTENDS D-BASIC-USER

SPECIFICATION

ops
subscribe **to** server
unsubscribe **to** server
ok **from** server
nok **from** server
addc !v:PIdUser **to** server
subc !v:PIdUser **to** server
call !p:PIdUser **to** server

ABSTRACTION

conditions subscribed
initially ¬ subscribed

OPERATIONS

subscribe
pre: IDLE ∧ ¬ subscribed
post: {subscribed':true}
unsubscribe
pre: IDLE ∧ subscribed
post: {subscribed':false}
ok
pre: ASKING ∧ subscribed
post: {COMM':true}
nok
pre: ASKING ∧ subscribed
post: {IDLE':true}
addc !v:PIdUser, subc !v:PIdUser
pre: IDLE ∧ subscribed
post: {}
call !p:PIdUser
pre: IDLE ∧ subscribed
post: {ASKING':true}

Fig. 20. The Dynamic View of Full Users

Figure 21 gives a graphical representation for the STS part of the dynamic view of full users.

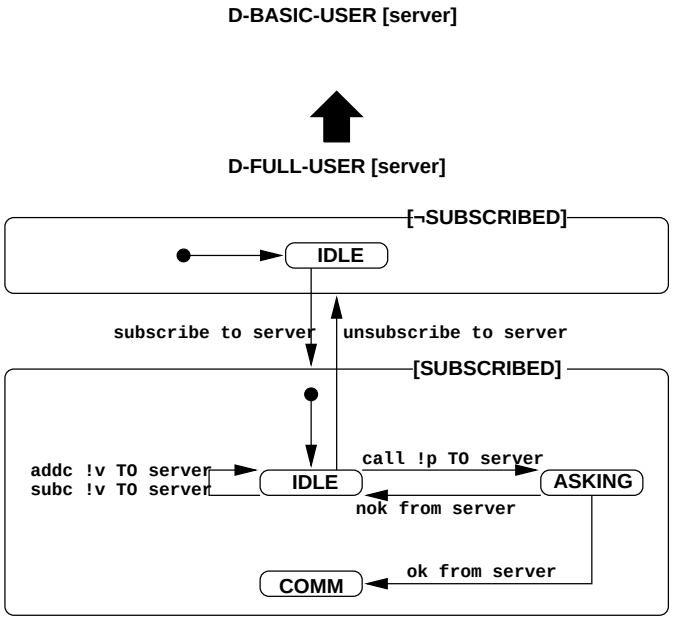


Fig. 21. A Graphical Representation of the `FULL USER` Dynamic View STS

The Integration View of Full Users. Figure 22 gives the integration view of full users in KORRIGAN. The static part of full users is a set of PIdUser. This corresponds to the set of (simple or full: since there is a subsort relation between sorts PIdFullUser and PIdUser, any PIdFullUser is also a PIdUser) users the full user knows. The glue part states that the full user may only add (or remove) from its authorization list, or communicate with users (s)he knows.

A.4 Connection Units

The Dynamic View of Connection Units. Figure 23 gives the dynamic view of connection units in KORRIGAN.

Figure 24 gives a graphical representation for the STS part of the dynamic view of connection units.

The Integration View of Connection Units (UC). Connection units have no static view (an empty one). Figure 25 gives the integration view of connection units in KORRIGAN.

A.5 Disconnection Units

The Dynamic View of Disconnection Units. Figure 26 gives the dynamic view of disconnection units in KORRIGAN.

INTEGRATION VIEW FULL-USER

SPECIFICATION

imports PId
generic on
server:PIdServer

COMPOSITION

is
s : SET [PIdUser]
d : D-FULL-USER [server]
with true, {
(true,d.addc !v to server ⇒ s.member(v),),
(true,d.subc !v to server ⇒ s.member(v)),
(true,d.call !p to server ⇒ s.member(p))}
initially true

Fig. 22. The Integration View of Full Users

DYNAMIC VIEW D-CONNECTION-UNIT

SPECIFICATION

imports PId
generic on udb : PIdDB
ops
call ?p:PIdUser **from** PIdUser
newCx !p:PIdUser !p2:PIdUser
to udb
ask !p:PIdUser !p2:PIdUser
to udb
ok **from** udb
nok **from** udb
ok **to** PId
nok **to** PId
called !p:PIdUser **to** PId
ok **from** PId
nok **from** PId

STS

• → IDLE
IDLE – call ?p:PIdUser **from** user → TESTING
TESTING – ask !p:PIdUser !user:PIdUser
to udb → WAIT1
WAIT1 – ok **from** udb → CALL
WAIT1 – nok **from** udb → CANCEL
CALL – call !user:PIdUser **to** p → WAIT2
WAIT2 – ok **from** p → $\bullet_1$
WAIT2 – nok **from** p → CANCEL
$\bullet_1$ – ok **to** user → $\bullet_2$
$\bullet_2$ – newCx !user:PIdUser !p:PIdUser **to** udb
→ IDLE
CANCEL – nok **to** user → IDLE

Fig. 23. The Dynamic View of Connection Units

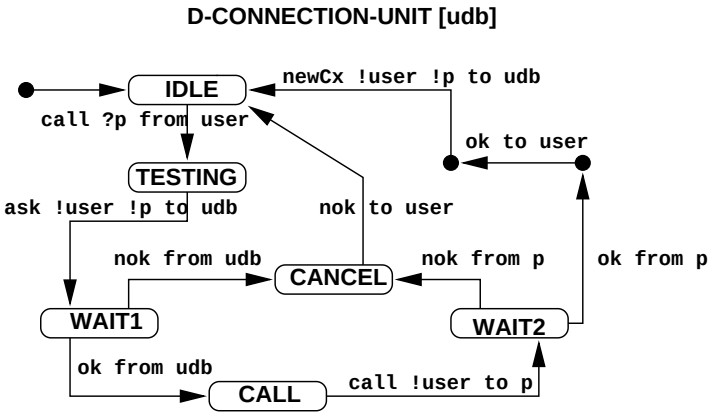


Fig. 24. A Graphical Representation of the `CONNECTION UNIT` Dynamic View STS

INTEGRATION VIEW CONNECTION-UNIT (UC)

SPECIFICATION

generic on
udb:PIdDB

COMPOSITION

is
s : S-EMPTY
d : D-CONNECTION-UNIT [udb]
with true, true
initially true

Fig. 25. The Integration View of Connection Units

DYNAMIC VIEW D-DISCONNECTION-UNIT

SPECIFICATION

imports PId
generic on udb : PIdDB
ops
eot **from** PIdUser
eot **to** PIdUser
askOther !p:PIdUser **to** udb
getOther ?p:PIdUser **from** udb
askSomeIn **to** udb
getSomeIn ?p:PIdUser **from** udb

STS

$\bullet \rightarrow$ IDLE
IDLE $-$ eot **from** user1 $\rightarrow$ ASK
IDLE $-$ askSomeIn **to** udb $\rightarrow \bullet_1$
$\bullet_1 -$ getSomeIn ?user1:PIdUser **from** udb $\rightarrow \bullet_2$
$\bullet_2 -$ eot **to** user1 $\rightarrow$ ASK
ASK $-$ askOther !user1:PIdUser **to** udb $\rightarrow \bullet_3$
$\bullet_3 - \rightarrow$ getOther ?user2:PIdUser **from** udb $\rightarrow \bullet_4$
$\bullet_4 -$ eot **to** user2 $\rightarrow$ IDLE

Fig. 26. The Dynamic View of Disconnection Units

Figure 27 gives a graphical representation for the STS part of the dynamic view of disconnection units.

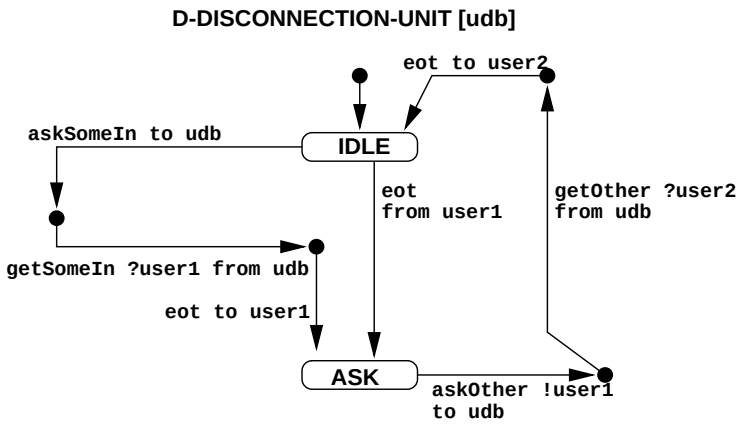


Fig. 27. A Graphical Representation of the `DISCONNECTION UNIT` Dynamic View STS

The Integration View of Disconnection Units (UD). Disconnection units have no static view (an empty one). Figure 28 gives the integration view of disconnection units in KORRIGAN.

INTEGRATION VIEW DISCONNECTION-UNIT (UD)

SPECIFICATION

generic on
udb:PIdDB

COMPOSITION

is
s : S-EMPTY
d : D-DISCONNECTION-UNIT [udb]
with true, true
initially true

Fig. 28. The Integration View of Disconnection Units

A.6 Management Units

The Dynamic View of Management Units. Figure 29 gives the dynamic view of management units in KORRIGAN.

DYNAMIC VIEW D-MANAGEMENT-UNIT	
SPECIFICATION	**STS**
imports PId **generic on** udb : PIdDB **ops** addc ?p:PIdUser **from** PIdUser subc ?p:PIdUser **from** PIdUser addc !u:PIdUser !p:PIdUser **to** udb subc !u:PIdUser !p:PIdUser **to** udb sub **from** PIdUser unsub **from** PIdUser sub !u:PIdUser **to** udb unsub !u:PIdUser **to** udb	• → IDLE IDLE − addc ?p:PIdUser **from** user → $\bullet_1$ IDLE − subc ?p:PIdUser **from** user → $\bullet_2$ $\bullet_1$ − addc !user:PIdUser !p:PIdUser **to** udb → IDLE $\bullet_2$ − subc !user:PIdUser !p:PIdUser **to** udb → IDLE IDLE − sub **from** user → $\bullet_3$ IDLE − unsub **from** user → $\bullet_4$ $\bullet_3$ − sub !user:PIdUser **to** udb → IDLE $\bullet_4$ − unsub !user:PIdUser **to** udb → IDLE

Fig. 29. The Dynamic View of Management Units

Figure 30 gives a graphical representation for the STS part of the dynamic view of management units.

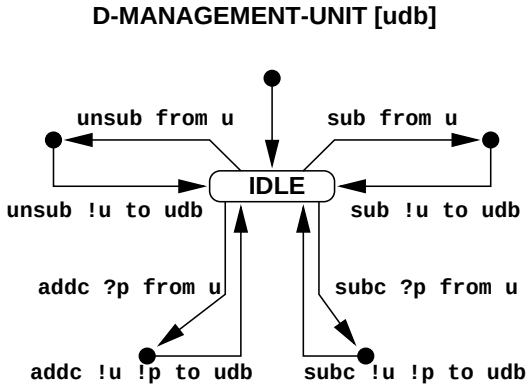


Fig. 30. A Graphical Representation of the `MANAGEMENT UNIT` Dynamic View STS

The Integration View of Management Units (UM). Management units have no static view (an empty one). Figure 31 gives the integration view of management units in KORRIGAN.

A.7 Database Units

The Dynamic View of Database Units. Figure 32 gives the dynamic view of database units in KORRIGAN.

INTEGRATION VIEW MANAGEMENT-UNIT (UM)

SPECIFICATION

generic on
udb:PIdDB

COMPOSITION

is
s : S-EMPTY
d : D-MANAGEMENT-UNIT [udb]
with true, true
initially true

Fig. 31. The Integration View of Management Units

DYNAMIC VIEW D-DATABASE-UNIT

SPECIFICATION

imports PId
generic on
uc:PIdUC,
ud:PIdUD,
um:PIdUM
ops
addc ?u:PIdUser ?u2:PIdUser
from um
subc ?u:PIdUser ?u2:PIdUser
from um
sub ?u:PIdUser **from** um
unsub ?u:PIdUser **from** um
newCx ?u:PIdUser ?u2:PIdUser
from uc
askSomeIn **from** ud
getSomeIn !p:PIdUser **to** ud
askOther ?u:PIdUser **from** ud
getOther !u2:PIdUser **to** ud
ask ?u1:PIdUser ?u2:PIdUser
from uc
ok **to** uc
nok **to** uc

STS

• → IDLE
IDLE − addc ?u:PIdUser ?u2:PIdUser
from um → IDLE
IDLE − subc ?u:PIdUser ?u2:PIdUser
from um → IDLE
IDLE − sub ?u:PIdUser **from** um → IDLE
IDLE − unsub ?u:PIdUser **from** um → IDLE
IDLE − newCx ?u:PIdUser ?u2:PIdUser
from uc → IDLE
IDLE − askSomeIn **from** ud → REPL3
REPL3 − getSomeIn !p:PIdUser **to** ud → IDLE
IDLE − askOther ?u:PIdUser **from** ud
→ REPL2
REPL2 − getOther !u2:PIdUser **to** ud → IDLE
IDLE − ask ?u1:PIdUser ?u2:PIdUser **from** uc
→ REPL1
REPL1 − [auth(u1,u2)] ok **to** uc → IDLE
REPL1 − [¬ auth(u1,u2)] nok **to** uc → IDLE

Fig. 32. The Dynamic View of Database Units

Figure 33 gives a graphical representation for the STS part of the dynamic view of database units.

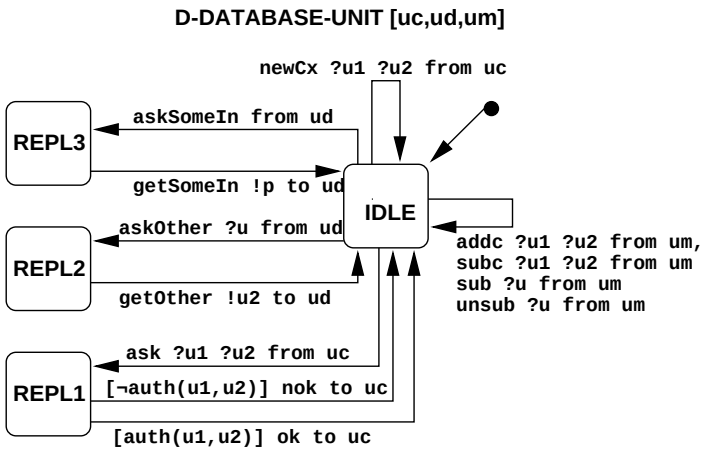


Fig. 33. A Graphical Representation of the `DATABASE UNIT` Dynamic View STS

The Integration View of Database Units (UDB). Figure 34 gives the integration view of database units in KORRIGAN. Its static part is made up of three static views. `sl` is the set of the subscribed users PId. `al` corresponds to authorizations, *i.e.* if a couple `(u1,u2)` is in `al`, then `u2` is an authorized caller for `u1`. `cl` contains couples of users communicating.

This is an extension of the (usual) integration view where there is only one static and one dynamic view. This is equivalent to the (single) static view that corresponds to the free (`ALONE`) composition [5] of the three static views. Another solution would have been to make a trivial (*i.e.* with dynamic operations corresponding to the static ones) dynamic view for each one of the three static views, integrate the dynamic view of databases with an empty static view, and then compose both (*i.e.* the four integration views) using a composition view.

A.8 Server

Figure 35 gives the graphical representation of the composition of the Server in terms of components and communications between these components.

Figure 36 gives the composition view of the server in KORRIGAN. Instances of UC, UM, UD and UDB are created. The corresponding parameters of processes are instanciated.

INTEGRATION VIEW DATABASE-UNIT (UDB)

SPECIFICATION

generic on
uc:PIdUC, ud:PIdUD, um:PIdUM

COMPOSITION

is
sl : SET [PIdFullUser]
al : SET [Couple [PIdUser]]
cl : SET [Couple [PIdUser]]
d : D-DATABASE-UNIT [uc,ud,um]
with true, {
(sl.add(u), d.sub ?u **from** um)
(sl.remove(u), d.unsub ?u **from** um)
(al.add((u1,u2)), d.addc ?u1 ?u2 **from** um)
(al.remove((u1,u2)), d.subc ?u1 ?u2 **from** um)
(al.member((u1,u2)), d.auth(u1,u2) **from** um)
(cl.add((u1,u2)), d.newCx ?u1 ?u2 **from** uc)
(cl.member(u), d.getSomeIn !p1(u) **to** ud ∨ d.getSomeIn !p2(u) **to** ud)
(cl.member((u,u2)) ∨ cl.member((u2,u)),
d.(− askOther ?u **from** ud > − getOther !u2 **to** ud >))}
initially true

Fig. 34. The Integration View of Database Units

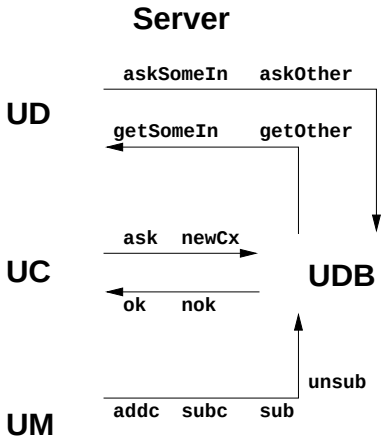


Fig. 35. A Graphical Representation of the Server Composition

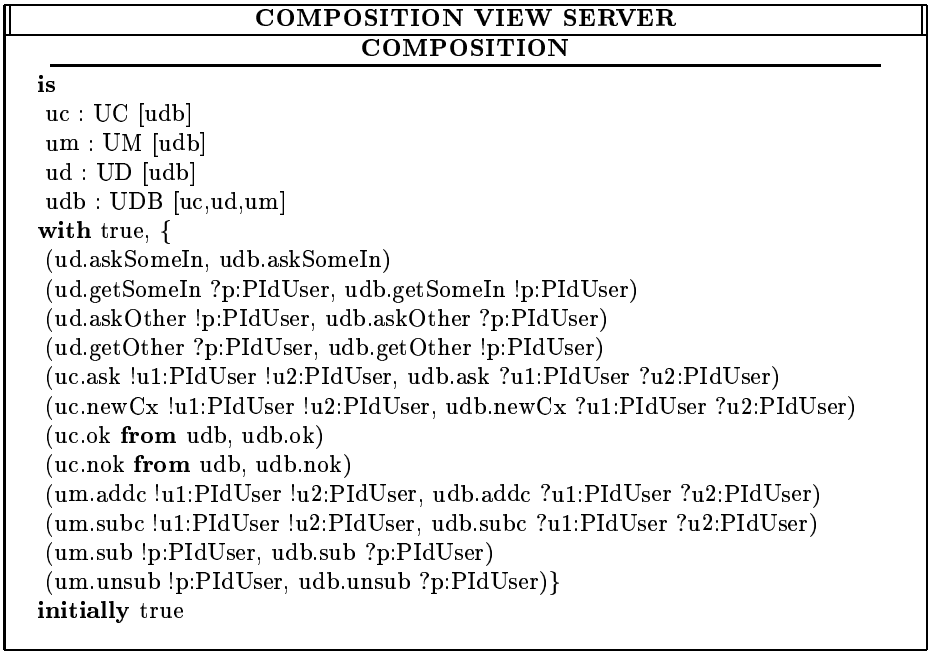


Fig. 36. The Composition View of the Server

A.9 The Composition View of the Phone Service

Figure 37 gives the graphical representation of the composition of the Phone Service in terms of components and communications between these components.

Figure 38 gives the composition view of the phone service in KORRIGAN. The phone service is generic on `n`, the number of basic users, and `m`, the number of full users. Users are created using the range operator. This operator is a shorthand for a (finite) set of PId (built using naturals). For example,
`i:[1..3]*user.i : USER`
is used as a shorthand for:

user.PId(1) : USER
user.PId(2) : USER
user.PId(3) : USER

The range operator may also be used in glue formulas.

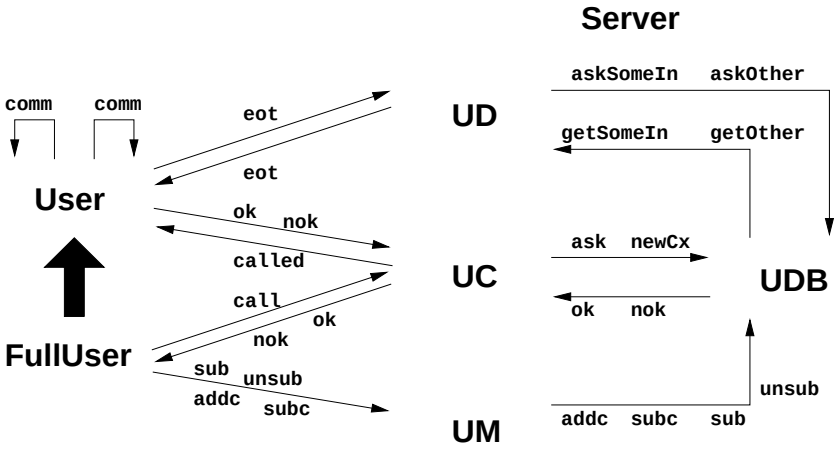


Fig. 37. A Graphical Representation of the Composition of the Phone Service

COMPOSITION VIEW PHONE-SERVICE

SPECIFICATION

generic on n,m : NATURAL

COMPOSITION

is
s : SERVER
i:[1..n]*user.i : USER [s]
j:[n+1..n+m]*user.j : FULL-USER [s]
with true, {
(user.i.comm ?m:Msg **from** user.j, user.j.comm !m:Msg **to** user.i)
(ud.eot **to** user.i, user.i.eot **from** ud)
(ud.eot **from** user.i, user.i.eot **to** ud)
(um.sub **from** user.i, user.i.sub **to** um)
(um.unsub **from** user.i, user.i.unsub **to** um)
(um.addc ?p:PIdUser **from** user.i, user.i.addc !p:PIdUser **to** um)
(um.subc ?p:PIdUser **from** user.i, user.i.subc !p:PIdUser **to** um)
(uc.ok **from** user.i, user.i.ok **to** uc)
(uc.nok **from** user.i, user.i.nok **to** uc)
(uc.ok **to** user.i, user.i.ok **from** uc)
(uc.nok **to** user.i, user.i.nok **from** uc)
(uc.called !p:PIdUser **to** user.i, user.i.called ?p:PIdUser **from** uc)
(uc.call ?p:PIdUser **from** user.i, user.i.call !p:PIdUser **to** uc)}
initially true

Fig. 38. The Composition View of the Phone Service