Integration of Control and Datatypes using the View Formalism

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. In this paper we present a specification formalism that provides a coherent framework for the integration of static and dynamic aspects of a system. In our approach, a system is specified as the composition of several components. Our framework is based on the concept of view which is a formal structure used to describe the static and dynamic parts of a component and their integration. Views are also the structuring mechanism for components. Each basic component has two views, a dynamic and a static view. Those two views are glued together by axioms and temporal logic formulas.
Our formalism uses *Symbolic Transition Systems* (STS) [10], a kind of guarded state/transition diagrams that finitely represent potentially infinite systems, to increase the readability and abstraction of the specification. The STS diagrams are an alternative to the textual representation of operations in terms of pre/post-conditions.
The system is structured by means of collections of objects (with identities). Axioms and temporal logic formulas are again used to glue the components altogether and express a generalized form of synchronous product [2].
We show how a global view structure may then be retrieved.
The formalism is explained using a simplified phone service example.
Keywords: mixed specifications, view, structuring, integration, datatypes, control, formalism, symbolic transition systems.

1 Introduction

The complexity of "real life" industrial systems often requires that several viewpoints or aspects are considered in order to specify and develop them adequately. Among the different identified aspects, one may usually find information about the static aspects (type, signature, architecture), about the dynamic ones (control and communication, states, lifetime, etc.), and the functional ones (constraints, invariants, operation descriptions, etc.) [23]. In our approach, we shall denote, for a given component, the types and functional aspects by the *static* *view*, the communications and the component states by the *dynamic view*, the integration of static and dynamic aspects by the *integration view*, architecture and concurrency by the *composition view*. In order to provide a uniform way to express, integrate and compose these different kinds of views, we require that they all contain three parts, the data part, the behaviour part, and the abstraction part. The three parts are strongly linked together.

We are interested in mixed systems where these different kinds of information are needed (such as those addressed by LOTOS [12], SDL [8], Estelle [11], UML [23], Raise [24], works presented in [7], extension of non mixed formalisms as ObjectZ/CSP [22]). While the description of each (kind of information) is usually well expressed using specific languages (such as state/transition systems for dynamic aspects, or algebraic specifications for data types), there is a need to provide a *unifying coherent framework* to *integrate* both. Therefore, the issues addressed in this framework that we would like to take into account are:

- a formalism appropriate for mixed systems specification, providing both dynamic and static aspects, and *structuring* means for specifications
- a *methodology* to help writing specifications for these systems [18]
- means to insure the *coherence* of the different aspects of the specification [6]
- an open and extensible *environment* for specifying and verifying such systems [4].

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). Integration of both static and dynamic parts of the specification as well as structuring between the different components is achieved through a formal structure that we call *views*. A method associated to our formalism has been defined in [18] and subsequently used to generate LOTOS and SDL specifications. In [19] we explain 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 symbolic transition systems (STS) [10] is of great interest as it enables one to define components in a very abstract and readable way. Therefore, we provide 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 global 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 The phone service example

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 statuses: (s)he may be an ordinary 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 (`BASIC-USER`).

1. (S)he may ask the server to become a subscriber only when (s)he is 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.

The subscriber (`FULL-USER`). A subscriber has the same rights as a client, and in addition:

1. (S)he may go back to the ordinary client status 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.
7. (S)he may be given the information of a new known client.

2.2 The central server

The `CENTRAL` server is composed with five units : a reception unit, a connection unit, a management unit, a disconnection unit and a database.

The reception unit.

1. The reception unit receives all the clients requests, and these requests are put in a queue.
2. Depending on their kind, the requests are sent to the connection, management, or disconnection units according to their arrival order.

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 status changes requests (ordinary client/subscriber), and all potential called party modifications.
3. The database unit provides a knowledge on (i) which are the current communications, and (ii) for each communication, 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. $\mathtt{i} \neq \mathtt{j}$.
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 A view based model

The central concept of our framework is the abstract concept of *view*. This view concept is used to provide a unifying approach to specify the different aspects of the components, e.g. the various structuration levels as well as inheritance. More generally, this concept provides a means for specifying and for integrating the various aspects of the components.

In order to obtain a good level of both abstraction and legibility, we use *Symbolic Transition Systems* (STS) [10] to represent the behavioral parts of views. In the following, we first describe the characteristics of STS, before introducing the view components.

3.1 Symbolic transition systems

STS are based on Symbolic Transition Graphs (STG) [10, 20] that are used as to finitely represent potentially infinite systems. STGs are a kind of guarded automaton where free variables may be used in guards and transitions. They were used to express a global semantics for part of the full-LOTOS language [21, 13].

Other approaches, such as TAG [1] use STS in order to provide an interpretation of algebraic specifications in terms of a finite state-transition system, while keeping the fully static feature (but with not attempt of structuring as in our view model).

Definition 1 (Symbolic Transition Graphs [10]). *A symbolic transition graph (STG) is a directed graph* (S, S_0, T) *in which every node* n *is labelled by a set of variables* $fv(n)$ *and every branch is labelled by a guarded action (a boolean guard* b *and an action* α *that may bind variables, denoted by* bv*) such that if a branch labelled by* (b, α) *goes from node* m *to node* 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 usual way: $fv(c!e) = fv(e)$*,* $bv(c?x) = \{x\}$ *and otherwise both* $fv(\alpha)$ *and* $bv(\alpha)$ *are empty.*

Here we use an extension of STGs where states are not sets of variables but equivalence classes[1] of (algebraic) terms (in the following we shall also consider states as tuples of *conditions* used to build the *abstraction* level of the described component). In this way, STS enable one both to avoid the state explosion by expressing the appropriate abstraction level, and to establish a link with the algebraic part of our views.

Let us consider an unbounded buffer containing natural numbers, and the operations put and get (get is allowed only when the buffer is not empty). In the usual transition systems approach, a state explosion is due to the fact that all variables (i.e., the receptions and process parameters) are instanciated (Figure 1); this explosion is both vertical (due to receptions - there is an infinity of different values that may be received) and horizontal (due to process parameters since there is an infinite number of possible states for the buffer content, both w.r.t. its size and the values contained).

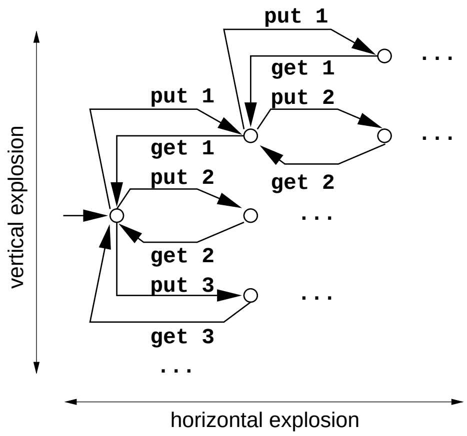


Fig. 1. An unbounded buffer state explosion with (usual) transition systems

[1] w.r.t. the conditions introduced in the abstraction part (Def. 6)

STS allow one to abstract away from the buffer content and the values received, and to focus only on whether the buffer is empty or not (Figure 2). Quoting [21], "STS are transition systems which separate the data from the process behaviour".

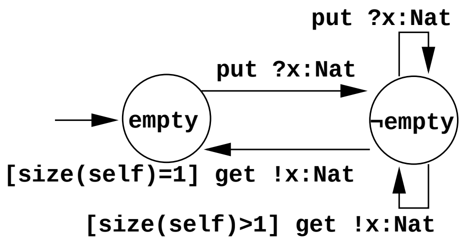


Fig. 2. An unbounded buffer modelled by a symbolic transition system

Moreover, to the contrary of STG, we find it adequate to distinguish between two sets of guards on transitions:

- on the left, a guard restricting the set of possible concrete states (members of the equivalence class
corresponding to the abstract source state) that enable the transition - on the right, a guard restricting the set of values of the free variables of the transition that enable the transition.

This distinction exists also in languages like LOTOS where there are guards (on the source state, or variables already bound) and action predicates (a restriction over the possible values for the free variables of the transitions).

We think that the STS formalism provides a very good legibility and eases the work (and integration) on our views when using operations that are defined in terms of states and transitions.

Whereas automata and closed transition systems are a commonly accepted notation equipped with design or verification tools (e.g. [9, 16]), STS were more recently developed and are not yet equipped with tools such as [20, 15, 13] for temporal model checking or [10, 21] for bisimulations.

3.2 The view classes hierarchy

While the definition of a view and its components will be given in the next section, we introduce here the different view classes that will be derived from it to ensure structuring.

The *Internal Structuring View* provides a uniform interface for both static and dynamic aspects of components. The *Integration View* is the first external structuration level where these aspects are integrated.

Another structuring level provided by the external structuring view is expressing the fact that an object may be composed of several other objects (*Composition View*). An object may then be either a simple one (integrating a *Static*

View and a *Dynamic View* 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 (`PId`).

The whole class diagram for the view model is given in Figure 3.

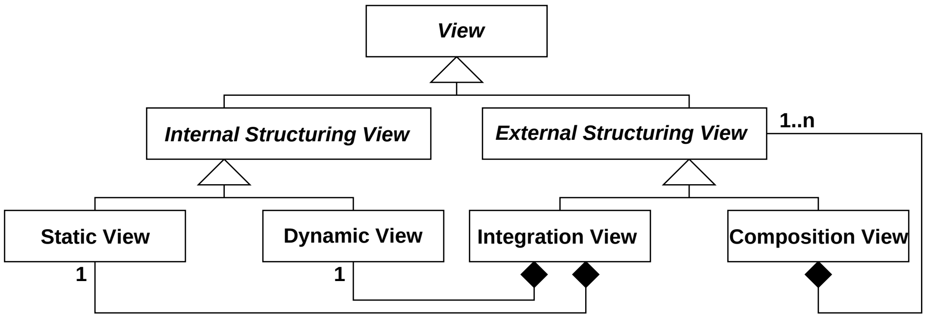


Fig. 3. The View model class diagram

3.3 Views components

As mentioned above, the view concept is used to provide a unifying approach to specify the different aspects of the components.

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 *is the behavioural part defined above* A_T*.*

The *data part* addresses the functional issues for the component, while the *behavioural part* describes the possible sequences of operations the component may perform. For a given aspect of a component, there may exist several different abstraction levels based on (selected) conditions and the *abstraction* reflects the selected level.

Back to the buffer example slightly modified, we now have the `put` *operation allowed when the buffer is not full, and the* `get` *operation allowed when it is not empty. We may consider four different levels of abstraction, one without any conditions, one with a condition about the emptyness of the buffer, one with a condition about it being full or not, and one with both conditions.*

This notion of different levels of abstraction for a given component is used in [14] for proofs and documentation. Our approach is similar. However we base our abstractions, and more generally the components construction, on conditions (called states predicates in [14]). In our approach, these conditions are central both for our formalism and our method, while in [14] they are auxiliary concepts for proofs and documentation.

The data part and the abstraction part are linked by the fact that it is possible to derive parts of the axioms formulas from the abstraction [1,18]. The behavioural and abstraction parts are also strongly linked since the operation semantics is expressed in terms of the abstraction.

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 [3] (i.e. with communication offers), enriched with arguments expressing the explicit communication scheme (using process identifiers PId).*

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` $\rightarrow$ `T`
- `comm !m:Msg to u:Pid`
  (emission of a message `m` to `u` on the `comm` gate)
  constructor `comm_c : T` $\rightarrow$ `T`
  observer `comm_o_m : T` $\rightarrow$ `Msg`
  observer `comm_o_id : T` $\rightarrow$ `PId`

The view behavioural part 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 behavioural part). *A* behavioural part 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[2] (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.
- $S = \{s \in 2^{|C|} | \forall \phi \in \Phi . s \models \phi\}$
- $S_0 = \{s_0 \in S | s_0 \models \Phi_0\}$
- $\forall o \in O - \overline{O}, \forall j, \forall s.s \models o.\mathcal{P}_c, \forall s'.s' \models o.\mathcal{Q}_c \ . \ t : s \xrightarrow{[o.\mathcal{Q}_{cl}] \ \ \{o\} \ \ [o.\mathcal{P}_{jo} \wedge o.\mathcal{Q}_{jo}]} s' \in T$
- $\forall o \in \overline{O}, \forall j, \forall s.s \models o.\mathcal{P}_c, \forall s'.s' \models o.\mathcal{Q}_c \ . \ t : s \xrightarrow{[o.\mathcal{Q}_{cl}] \ \ i_{\{o\}} \ \ [o.\mathcal{P}_{jo} \wedge o.\mathcal{Q}_{jo}]} s' \in T$

States are composition of conditions that satisfy 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.

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

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

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

Fig. 4. The KORRIGAN syntax (views)

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 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).

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

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.

4 The different view classes

In the following, we describe the different views, and illustrate them on the phone service example (cf. [5] for the complete case study specification).

The different aspects of components are specified using *Internal Structuring Views* (*i.e.* static views for static aspects, and dynamic views for dynamic aspects). In our unifying approach, dynamic views are the same as static views but for some differences:

– no full algebraic specification (only signatures matter)
– use of the `from/to` elements in the dynamic signatures.

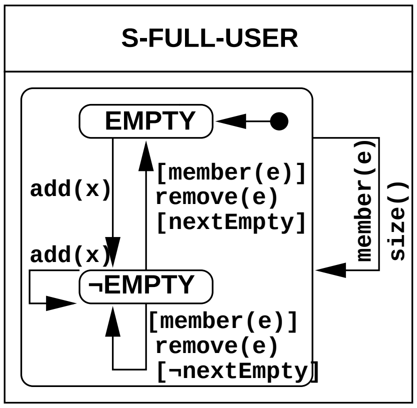
This goes with the fact that dynamic views focus only on the communication abstraction.

4.1 Static views

Static views (SV) describe the static aspect of a component. A *static view* SV_T of a component T has the same structure as simple views (cf. Definition 2).

In our case study, the ordinary client is modelized by the `BASIC-USER` and the subscriber by the `FULL-USER`. The `FULL-USER` static view is given in Figure 6. It corresponds to a set of known users (PId). Therefore, it defines the usual set operations: `member`, `size`, ... It is abstracted as being either `empty` or not (using a corresponding condition).

Its dynamic view will be defined as an extension (Fig. 8) of the `BASIC-USER` dynamic view (Fig. 7).



STATIC VIEW S-FULL-USER (SFU)	
SPECIFICATION	**ABSTRACTION**

imports PId, NAT
ops
new : → SFU
add : SFU × PId → SFU
member : SFU × PId → BOOL
size : SFU → NAT
nextEmpty : SFU → BOOL
remove : SFU × PId → SFU
axioms
nextEmpty(s) == size(s)=1;
member(new,e) == false;
...
	conditions empty
	limit conditions nextEmpty
	initially ¬ empty
	OPERATIONS
	add
	pre: true
	post: {empty':false}
	remove
	pre: ¬ empty ∧ member(e)
	post: {empty':nextEmpty}
	...

Fig. 6. The `FULL-USER` static view

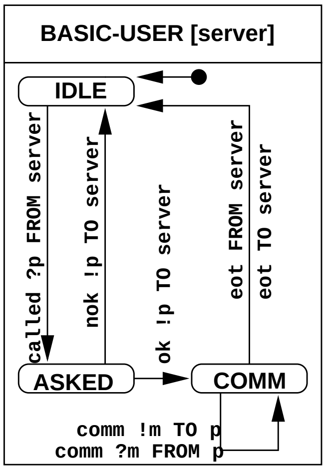
4.2 Dynamic views

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. Moreover dynamic views may use abstract guards. The semantics of these guards will be defined when the integration with a static view is achieved.

Definition 7 (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` communication protocol is specified in the Figure 7 dynamic view.



DYNAMIC VIEW BASIC-USER
SPECIFICATION

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

STS

.→IDLE
IDLE − called ?p : PId FROM server→ASKED
ASKED − ok !p : PId TO server→COMM
ASKED − nok !p : PId 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

4.3 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 ordinary clients that may only be called (Figure 7) and subscribers that may also call other users (Figure 8). The corresponding specification is given in Figure 9 where the writing of the postconditions is simplified, we omit conditions such as `X':X` which do not change.

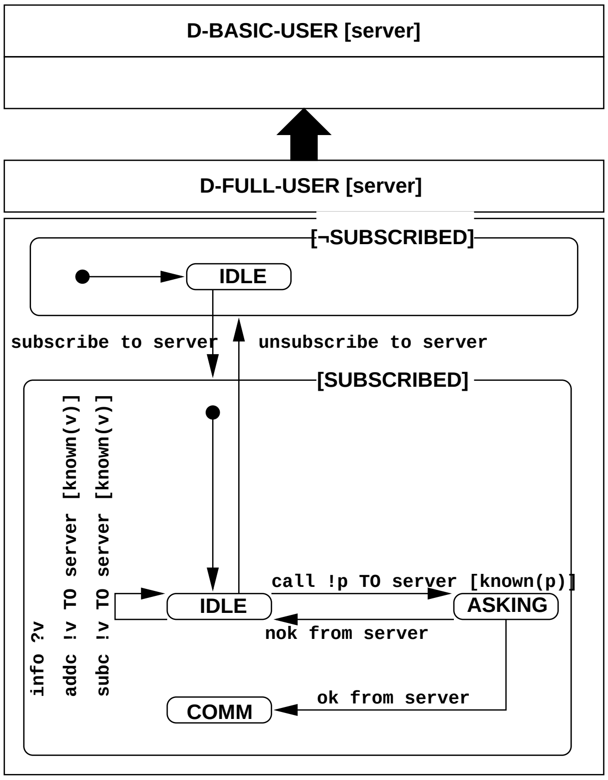


Fig. 8. The `FULL-USER` dynamic view

4.4 External structuring views (ESV)

Aspects of components specified using internal structuring views have no global semantics. Integration views are used to integrate the different aspects within a single component. More generally, external structuring views (*i.e.* either integration views or composition of ESV using composition views) are the only views that have a global semantics.

DYNAMIC VIEW D-FULL-USER EXTENDS D-BASIC-USER	
SPECIFICATION	**OPERATIONS**
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 info ?v:PIdUser known ?x:PIdUser !v:Bool **ABSTRACTION** **conditions** subscribed **initially** ¬ subscribed	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, info ?v:PIdUser **pre:** IDLE ∧ subscribed **post:** {} call !p:PIdUser **pre:** IDLE ∧ subscribed **post:** {ASKING':true}

Fig. 9. The `FULL-USER` dynamic view (KORRIGAN syntax)

Definition 8 (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.*

The optional data part is used whenever some new parameters and/or algebraic data types are needed in the ESV.

Definition 9 (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 integrated (glued) altogether.

Definition 10 (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.*
- *Φ is a (state) formula built over id-indexed C members of the components views[3].*

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

- *Ψ is a (transition) id-indexed formula relating (gluing) transitions from several components STS. This is a generalized form of synchronized product vector [2].*
- *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 5.1).*

Φ and Ψ are implicitly universally quantified over time (*i.e.* we have[4] $AG\Phi$ 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 formulas are simple ones: there are no fixpoint or special modalities like in more expressive temporal logics such as CTL, μCRL or XTL [16]. 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 [15] approach). State and transition formulas are detailed in [5].

Definition 11 (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 [6]. Here, we focus only on the obtaining of views structures for compositions of components. The syntax for external structuring views in KORRIGAN is given in Figure 10.

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

Fig. 10. The KORRIGAN syntax (external structuring views)

ESV is an abstract view, with two subclasses, one for integrating a component with a static and a dynamic view, and the other to structure the complex components. The integration view of `FULL-USER` is given in Figure 11. In this integration, the abstract guard “known” in the dynamic part is related to the concrete observer “member” of the static part using an axiom. Moreover, the effect of the dynamic transition ”info” on the static part is given by relating this transition with the ”add” transition of the static view (with clause).

[4] Where AG denotes “always globally”.

INTEGRATION VIEW FULL-USER

SPECIFICATION

imports PId
generic on
server:PIdServer

COMPOSITION

is	
s : S-FULL-USER	**axioms** $d.known(x) == s.member(x)$
d : D-FULL-USER [server]	**with** true, { (d.info ?v, s.add(v)) }
	initially *true*

Fig. 11. The `FULL-USER` integration view

5 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. 12).

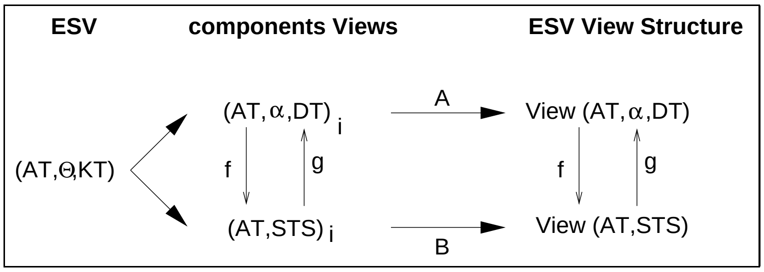


Fig. 12. Obtaining of a view structure for an ESV

The first one is to use the ESV components view structures $((A_T, \alpha, D_T))$, and obtain (arrow A in Fig. 12) a view structure for the ESV. The second one is to use the ESV alternative view structures (using STS with $STS = (S, S_0, T)$), and obtain (arrow B in Fig. 12) an alternative view structure for the ESV. Given the functions mapping view structures and alternative view structures (say `f` and `g` if a choice of conditions is made), 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. This is done in two parts:

- build a composed and indexed STS part
- build a composed and indexed data part.

5.1 Synchronization constraints

The ESV glue part Θ (in ESV structures, Definition 10) provides 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 Θ. 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).

The retrieval rules (Figures 13 and 14) have been implemented in [17].
ESV view structures are a special kind of views 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.

5.2 STS part obtaining

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 Φ and Φ_0 components of the ESV structure. The set of states is built from the initial states ($STATE_0$ rule) and the target states of the transitions ($STATE_1$ rule).

$$\frac{s_{0_s} \in S_0(Obj_s),\, s_{0_d} \in S_0(Obj_d),\, \langle\ s.s_{0_s}/d.s_{0_d}\ \rangle \models \Phi \wedge \Phi_0}{\langle\ s.s_{0_s}/d.s_{0_d}\ \rangle \in S_0}\ INIT$$

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

Fig. 13. State related retrieval rules

A couple of transitions (one from the static view, the other from the dynamic view) that satisfies some couple of transition formulas from the Ψ 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 Φ component of the ESV structure. The composition of states and transitions uses indexing in order to keep the system structure.

5.3 A_T part obtaining

The rules for the $A_T = (A', G, V, \Sigma_T, Ax_T, \overline{A})$ part (from [5]) use indexing to build a product (static) type for A_T:

- $A' = A'(ESV) \cup I_{G_i} \bigcup_i A'(Obj_i(ESV))$, I_{G_i} being instantiations I_i of $G(Obj_i(ESV))$

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

Fig. 14. Transition related retrieval rule

- $G = \bigcup_i i.G(Obj_i(ESV))$, 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 [5]).
- $\overline{A} = \overline{A} \bigcup_i \overline{A}(Obj_i(ESV))$

Using these two sets of rules (for STS and A_T parts) a global view that corresponds to the composition of views may be retrieved. The relationship between the abstract guards of the dynamic view and observers of the static view leads, e.g. for "addc" (Fig.8 and Fig.9) to a transition of the form
[⟨ s.true/d.true ⟩]⟨ s.ϵ/d.addc !v **to** server ⟩[⟨ s.true/d.known(v) ⟩].
The effect of some dynamic transition on the static part leads, e.g. for "add" (Fig.6) and "info" (Fig.8 and Fig.9), to
[⟨ s.true/d.true ⟩]⟨ s.add(v)/d.info ?v ⟩[⟨ s.true/d.true ⟩].
The global semantics for the composed and indexed views is described in [6].

6 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* 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). Temporal logic formulas and axioms are used to glue the components altogether and express a generalized form of synchronous product. 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).

In [6], we present the strong links that exist between all aspects of a single component, and that all together build its operational global semantics.

We already have defined an environment (the Korrigan environment) [4] to support our model of views.

We think this formalism is well suited to achieve a good balance between the readability, expressiveness and abstraction properties for the specifications. To test our approach, we have specified different case studies: hospital admissions [18], a transit node in a communication network [19], a password management system [6] amongst others. In this paper we present a phone service example and we apply our model to its specification (cf. [5] for the full specification).

One important remaining problem is to extend our composition mechanism to a finite (but unknown) number of components. This needs to define a synchronous product of a variable number of automata. A next step is to define proof mechanisms (algebraic, model checking or tableaux) for our view-based specifications.

References

1. Pascal André and Jean-Claude Royer. An Algebraic Approach to the Specification of Heterogeneous Software Systems. In *14th Workshop on Algebraic Development Techniques*, Bonas, France, September 1999. http://www.sciences.univ-nantes.fr/info/perso/permanents/royer/papers/rrwadt.ps.gz.
2. André Arnold. *Systèmes de transitions finis et sémantique des processus communicants*. Etudes et recherches en informatique. Masson, 1992.
3. Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–29, January 1988.
4. Christine Choppy, Pascal Poizat, and Jean-Claude Royer. The Korrigan Environment. FM-TOOLS 2000, 4th Workshop on Tools for System Design and Verification, Reisensburg Castle, Germany, July 2000.
5. 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.
6. Christine Choppy, Pascal Poizat, and Jean-Claude Royer. A Global Semantics for Views. In *International Conference on Algebraic Methodology And Software Technology, AMAST'2000*, volume 1816 of *Lecture Notes in Computer Science*, 2000.
7. H. Ehrig, M. Grosse-Rhode, and F. Orejas, editors. *INT'2000, Workshop on Integration of Specification Techniques with Applications in Engineering*, Technische Universitat Berlin, Germany, 2000. Bericht-Nr. 2000/04, ISSN 1436-9915.
8. Jan Ellsberger, Dieter Hogrefe, and Amardeo Sarma. *SDL : Formal Object-oriented Language for Communicating Systems*. Prentice-Hall, 1997.
9. Jean-Claude Fernandez. *ALDEBARAN : un système de vérification par réduction de processus communicants*. Thèse de Doctorat, Université Joseph Fourier, Grenoble, Mai 1988.
10. M. Hennessy and H. Lin. Symbolic Bisimulations. *Theoretical Computer Science*, 138(2):353–389, 1995.
11. ISO/IEC. ESTELLE: A Formal Description Technique based on an Extended State Transition Model. ISO/IEC 9074, International Organization for Standardization, 1989.
12. ISO/IEC. LOTOS: A Formal Description Technique based on the Temporal Ordering of Observational Behaviour. ISO/IEC 8807, International Organization for Standardization, 1989.

13. Carron Kirkwood and Muffy Thomas. Towards a Symbolic Modal Logic for LOTOS. In *Northern Formal Methods Workshop NFM'96*, eWIC, 1997.
14. Leslie Lamport. TLA in Pictures. *IEEE Transactions on Software Engineering*, 21(9):768–775, 1995. Available at http://www.research.digital.com/SRC/personal/Leslie_Lamport/tla/papers.html.
15. Radu Mateescu. *Vérification des propriétés temporelles des programmes parallèles.* PhD thesis, Institut National Polytechnique de Grenoble, 1998.
16. 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.
17. 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.
18. 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.
19. 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.
20. 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.
21. 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, 1997.
22. 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.
23. Rational Software. Unified Modeling Language, Version 1.1. Technical report, Rational Software Corp, http://www.rational.com/uml, September 1997.
24. The Raise Method Group. *The RAISE Development Method.* The Practitioner Series. Prentice-Hall, 1995.