Formal Specification of Mixed Components with Korrigan

Christine Choppy
LIPN, Université Paris XIII
Institut Galilée
avenue J.-B. Clément
93430 Villetaneuse – France
choppy@lipn.univ-paris13.fr

Pascal Poizat
LaMI, Université d'Évry
Tour Evry 2
523 place des terrasses de l'Agora
91000 Évry – France
poizat@lami.univ-evry.fr

Jean-Claude Royer
IRIN, Université de Nantes
2 rue de la Houssinière
B.P. 92208
44322 Nantes – France
royer@irin.univ-nantes.fr

Abstract

Formal specifications are now widely accepted in software development. Recently, the need for a separation of concerns with reference to static and dynamic aspects appeared. Furthermore, in order to be able to design complex architectures and communication patterns, we need a structured approach which allows us to decompose a system into subparts and to recompose it from them. Finally, a notion of formal reusable component which is suitable to describe both functional and dynamic aspects is also required.

This paper presents the Korrigan model. It allows one to specify in a uniform and structured way both datatypes and behaviours using Symbolic Transition Systems and algebraic specifications. We demonstrate its ability to describe reusable components in a formal way with well defined interfaces. We also demonstrate that Korrigan is relevant to describe the architecture and communication schemes of systems that may present a complex structure.

keywords: *Formal specification, mixed specification, components, reuse, Korrigan specification language*

1. Introduction

The use of formal specifications is now widely accepted in software development to provide abstract, rigorous and complete descriptions of systems. Formal specifications are also essential to prove properties, to prototype the system and to generate tests.

In the last few years, the need for a separation of concerns with reference to static (data types) and dynamic aspects (behaviours, communications) appeared. This issue was addressed in approaches combining algebraic data types with other formalisms (*e.g.* LOTOS with process algebras or SDL with State/Transition Diagrams), and also more recently in approaches based on Z (*e.g.* with Promela [10], or with CSP [19, 5]). This is also reflected in object oriented analysis and design approaches such as UML where static and dynamic aspects are dealt with by different diagrams (class diagrams, interaction diagrams, Statecharts). However, the (formal) links and consistency between the aspects are not defined, or trivial. This limits either the possibilities of reasoning on the whole component or the expressiveness of the formalism.

To design complex architectures and communication patterns, we need a structured approach which allows us to decompose a system into subparts and recompose it from them. A notion of formal component which is suitable to describe functional and dynamic aspects is also required.

Software engineering projects need techniques to be effective in-the-large. But first of all they must be effective in-the-small, *i.e.* at the component level of a system. It is probably difficult to give a general definition for the notion of component. Our notion of component is a defined unit of functionality which can be taken out of the current context and reused in other contexts.

Component interfaces are of crucial importance because often a designer builds a component and another one reuses it. Formal specifications are essential to abstract, simplify and disambiguate interfaces. Several kinds of interfaces are useful, the most common are signatures (static interfaces) and dynamic interfaces (gates, communications, ...). A good architecture must be drawn at least for the overall structure, but some other parts are only feasible via formal textual descriptions. Furthermore we must provide methods and tools (code generation, ...).

The Korrigan formalism is devoted to the structured formal specification of reusable mixed components through a model based on a hierarchy of *views*. Our approach aims at keeping advantage of the languages dedicated to both aspects (*i.e.* Symbolic Transition Systems for behaviours, algebraic specifications derived from these diagrams for data parts, and a simple temporal logic and axiom based glue for compositions) while providing an underlying unifying

framework accompanied by an appropriate semantic model. Moreover, experience has shown that our formalism leads to expressive and abstract, yet readable specifications.
In this paper we present the adequation of the Korrigan approach to the specification of reusable components on a simple telephony case study.

The paper is organized as follows. Section 2 presents an overview of the Korrigan model, its view principles, and its textual and graphical notations. Then, in Section 3 we present the Korrigan architecture and apply its principles for the definition of components on a phone service case study. We also detail three aspects related to the reusability of components in Korrigan: inheritance, instantiation and communication patterns. Finally, Section 4 discusses some related works, and a conclusion summarizes the main points of the paper.

2. Korrigan and the view model

In this Section, we will briefly present our model and the Korrigan specification language.
Our model [12] is based upon the structured specification of communicating components (with identifiers) by means of structures that we call *views* which are expressed in Korrigan, the associated formal language. A formal operational semantics for Korrigan has been defined in [2].
Views are used to describe in a structured and unifying way the different aspects of a component using “internal” and “external” structuring. We define an *Internal Structuring View* abstraction that expresses the fact that, in order to design a component, 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). Another structuring level is achieved through the *External Structuring View* abstraction, expressing that a component may be composed of several subcomponents. Such a composite component may be either an integration component (integrating different internal structuring views in an *Integration View*), or a concurrent component (*Composition View*). Integration views follow an *encapsulation principle*: the static aspect (*Static View*) may only be accessed through the dynamic aspect (*Dynamic View*) and its identifier (`Id`). The whole class diagram (UML notation) for the view model is given in Figure 1.

2.1. Basic components

Basic components are specified using *Internal Structuring Views*. Such a view may be given as a triple (`SPECIFICATION`, `ABSTRACTION` and `OPERATIONS` parts) as in Figure 2, or using derivation principles [13], as a couple (a `SPECIFICATION` part and a *Symbolic Transition System*[1]).

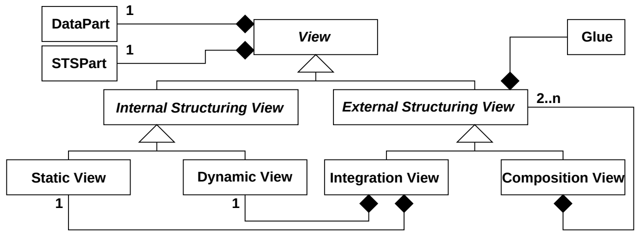

Figure 1. Views class hierarchy.

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

Figure 2. Korrigan syntax (basic components).

A set of conditions defines an abstract point of view for components. STS, *i.e. Symbolic Transition Systems* are built using the conditions. The main interest with these transition systems is that (i) they avoid state explosion problems, and (ii) they define equivalence classes (one per state) and hence strongly relate the behavioural and the algebraic representations of a data type. STSs may be related to Statecharts [7] but for some differences:

- STSs are simpler (but less expressive) than Statecharts;
- STSs model sequential components (concurrency is done through external structuring and the computation of a structured STS from subcomponents STSs [2]);
- STSs are built using conditions which enable one to semi-automatically derive them from requirements;
- STSs may be seen as a graphical representation of an abstract interpretation of an algebraic data type.

Basic components may also be described using a UML inspired graphical notation.

[1]Mainly a transition system with guarded transitions and open terms in states and transitions, see Figure 11 or [2].

Interface diagrams. The communication interface symbols we use are described in Figure 3.

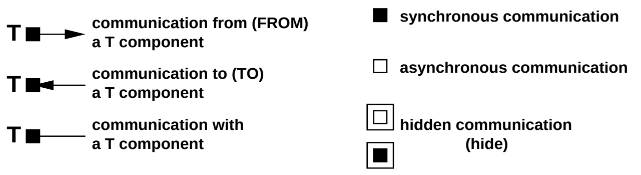

Figure 3. Communication interfaces symbols.

The components are represented with *boxes* with well-defined interfaces. These interfaces use *dynamic signatures*, *i.e.* event names with offer parameters (as in LOTOS) and interaction typing (as in SDL). As for the textual representation seen earlier on, one may either use STS (Figure 4) or not (Figure 5).

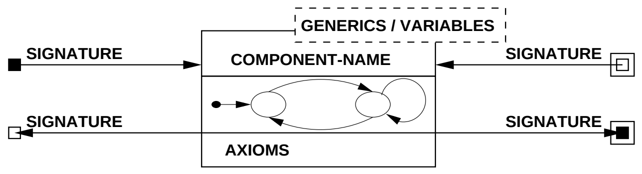

Figure 4. Component interface diagram (with STS).

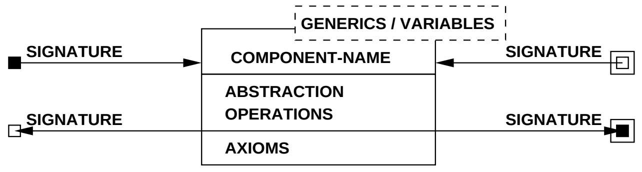

Figure 5. Component interface diagram (without STS).

Various simplifying notations have been defined for our STS, such as activities or hierarchical states, see [12].

2.2. Composite Components

There exists different kinds of compositions within the set-theoretic or the object-oriented framework. We restrict ourselves to the strong composition of critical systems and distributed applications, that is a composition with dependence, exclusivity and predominance. This can be compared with composition in UML (black diamond). However, our approach is more component-oriented than UML since we explicitly address communication issues in interfaces (Figure 4) and concurrency in communication diagrams (Figure 8) whereas in UML they are embedded within Statecharts and Message Sequence Charts.

Components are "glued" altogether in external structuring views (Figure 6) using both axioms and temporal logic formulas. This glue expresses a generalized form of synchronous product (for STS) and may be used to denote different concurrency modes and communication semantics. The δ component may be either `LOOSE`, `ALONE` or `KEEP` and is used in the operational semantics to express different concurrency modes (synchronous or asynchronous modes) and communication schemes. The `axioms` clause is used to link abstract guards that may exist in some components with operations defined in other components. The Φ and Φ_0 elements are state formulas expressing correct combinations of the components conditions (Φ) and initial ones (Φ_0). The Ψ element is a set of couples of transition formulas expressing which transitions have to be triggered at the same time (this expresses synchronization and communication). The `COMPOSITION` clauses may use a syntactic sugar: the *range* operator (`i:[1..N]` or `i:[e`$_1$`,...,e`$_n$`]`), a bounded universal quantifier.

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

Figure 6. Korrigan syntax (composite components).

Composition diagrams. We introduce composition diagrams (Figure 7) to decompose a component into subcomponents. We denote by (de)composition both the separation/integration of the different aspects of a component (static and dynamic), and the (de)composition into concurrent communicating subcomponents. The Korrigan model enables us to describe both in a unifying way using specific *External Structuring Views, i.e. Integration* and *Composition Views*.

Communication diagrams. The communication diagrams (Figure 8) are used to complement the composition diagrams with the inter-component communication and concurrency schemes. They use a graphical notation of the Korrigan glue rules (`COMPOSITION` parts in external structuring views, Figure 6).

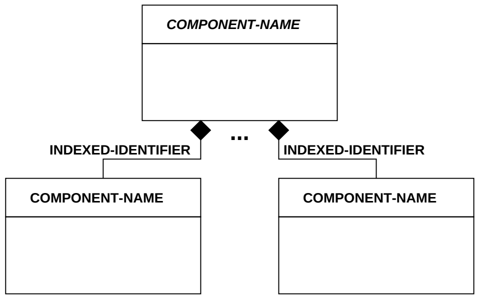

Figure 7. Composition diagrams.

The axiomatic part of the glue (the `axioms` clause), the state temporal formulas (Φ and Φ_0), and the concurrency mode (δ) are put in the aggregating component. Each element of the transition couples (Ψ) is treated by linking the involved components with a node (square boxes in Figure 3). The parts of the couple relative to each of these components is put on the lines: the elements above denoting the components participating in the synchronizations, the elements under the lines denoting the events. Here we may use again range operators as a syntactical shorthand (one link using the range operator is used in place of several ones that would be used without it). Indexed operators correspond to paths in the decomposition trees built using composition diagrams (*e.g.* in a car component, $wheels.4$ would stand for the fourth subcomponent of a $wheels$ subcomponent, *i.e.* the fourth wheel of the car).

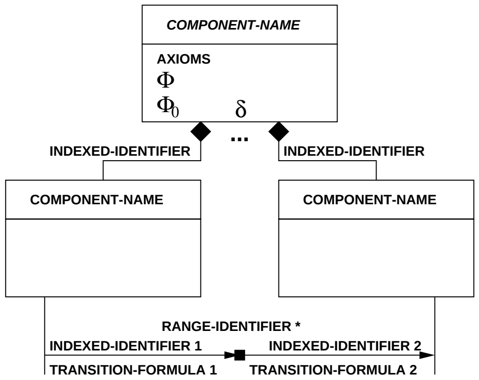

Figure 8. Communication diagrams.

When components at different levels are involved we adopt a structured communication scheme and do not add more links on the communication nodes. We add the communication information on parents (in the decomposition tree) of the concerned subcomponents. Moreover, this corresponds to the Java code-generation mecanisms we defined for our specifications in [13].

The obtaining of a textual Korrigan specification from its diagram is straightforward [12].

This representation of the communication is expressive enough to describe different kinds of communication, for example, both point-to-point communication (`ptp`) and broadcast communication (`broadcast`) in a client-server communication pattern, Figure 9.

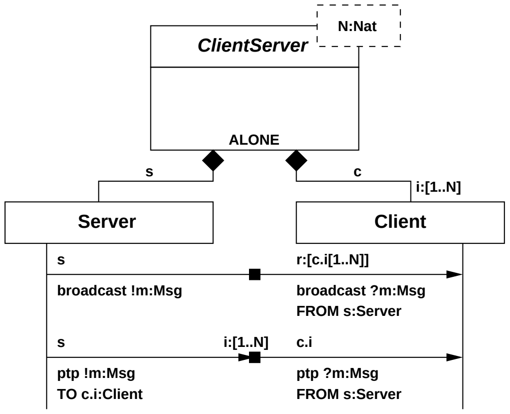

Figure 9. Client-Server pattern.

3. Specification and reuse of components

3.1. Architecture

The overall architecture of the system is presented in the Figure 10. The phone service is a component composed with users and a server. It is generic on `N`, the number of users. The notations we use are close to UML compositions and parameterized classes. The structure of the server is a composition of four components which are `UC` (Connection Unit), `UDB` (DataBase Unit), `UD` (Disconnection Unit) and `UM` (Management Unit).

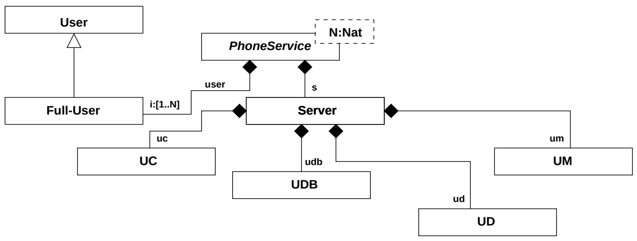

Figure 10. Phone Service architecture.

3.2. STS

The dynamic behaviour of the basic user is depicted in the Figure 11. This is a finite symbolic transition diagram without guard. Such a component specification is generic because we must glue it with a server which appears as a formal sender or emitter of messages. Such STSs are really good ways to describe middle size symbolic dynamic behaviours. They are not directly adequate for model-checking but solutions exists [8, 17, 15].

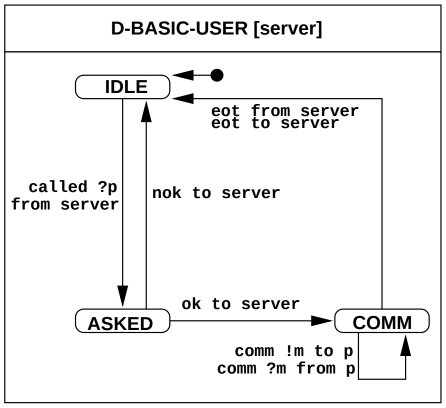

Figure 11. Basic User STS.

3.3. Reuse: inheritance

Generally inheritance [11] adds methods to a class, and allows overloading and masking. But inheritance may also be used to add constraints.

Here we only provide a simple form of inheritance for internal structuring views. Furthermore two views related by inheritance must be of the same type, for example a dynamic view can only inherit from another dynamic view. Our inheritance semantics is restricted to the addition of new conditions and new operations in the subview. It does not allow overloading and there is no masking because it actually carries too many difficulties and it is too complex for specifiers.

3.4. Reuse: instantiation

In presence of generic components, we may use instantiation diagrams to relate concrete components to their generic parent. Such views are reusable. Generally, the static views are sets, lists or buffers, *i.e.* the description in dynamic terms of the inputs and outputs of a storage element. As far as the case study is concerned, instantiation is used in the Database Unit.

The dynamic view of database units. Figure 12 gives a graphical representation for the STS part of the dynamic view (*i.e.* their communication protocol) of database units. The effect of the dynamic events in this STS on the datatypes owned by the databases are described when this dynamic aspect is integrated with its static aspects.

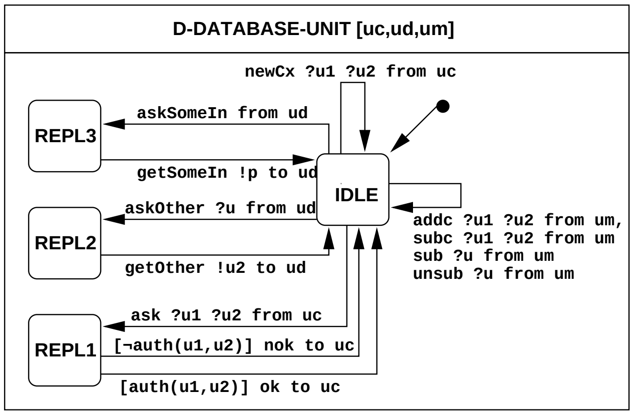

Figure 12. Database Unit dynamic view STS.

The integration view of database units. Figure 13 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 communicating users.

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 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.

The `with` clause of the integration is used to synchronize dynamic events defined in the Database Units protocol (*i.e.* their dynamic view) with operation on sets that constitute their datatype (*i.e.* their static view). Each couple of this clause gives a correspondence. For example, the `(cl.add((u1,u2)), d.newCx ?u1 ?u2` **from** `uc)` couple in the `with` clause states that whenever a `newCx ?u1 ?u2` communication is received from `uc` in the Database Unit dynamic view then, the `add` operation is performed on the `cl` element (couples of communicating users) of its static view with `u1` and `u2` as arguments.

3.5. Reuse: communication patterns

As in LOTOS and object-oriented programming some idioms or patterns may be used to describe general and common architectures of systems. Here the phone service is a kind of client-server architecture. Such a pattern is designed in the Figure 9.

Applying this pattern to our system yields an architecture skeleton which has been adapted and completed to our

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
$\vee$ d.getSomeIn !p2(u) **to** ud)
(cl.member((u,u2)) $\vee$ cl.member((u2,u)),
d.($-$ askOther ?u **from** ud $>$
$-$ getOther !u2 **to** ud $>$))}
initially true

Figure 13. Database Unit integration view.

specific purposes. The overall architecture with communications and synchronizations is split into Figures 15 and 16. We may note that we have a powerful and readable glue which allows us to link symbolic transition systems to denote synchronizations and communications in an abstract and concise way.

Some details on the connection units have to be given before we may present the communication diagrams corresponding to the case study. Figure 14 gives a graphical representation for the STS part of the dynamic view of connection units. The labels of its transitions will be used in the communication diagrams. We do not give details for the other components here by lack of place, see [12] for the full specification.

4. Related work

There are several related works either on formal components and architectures or on telecommunication services.

One important related work is Distributed Feature Composition (DFC) [20] which is a component based architecture for telecommunications services. It is based on a pipe-and-filter architecture and proposes a general model to define telecommunication services based on a network (tree) of boxes. Each box is either transparent or processes some

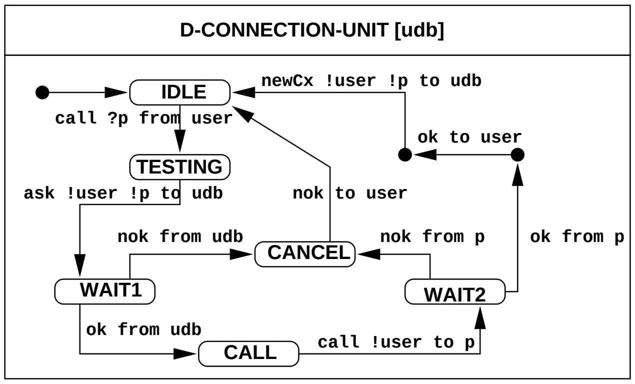

Figure 14. Connection Unit dynamic view STS.

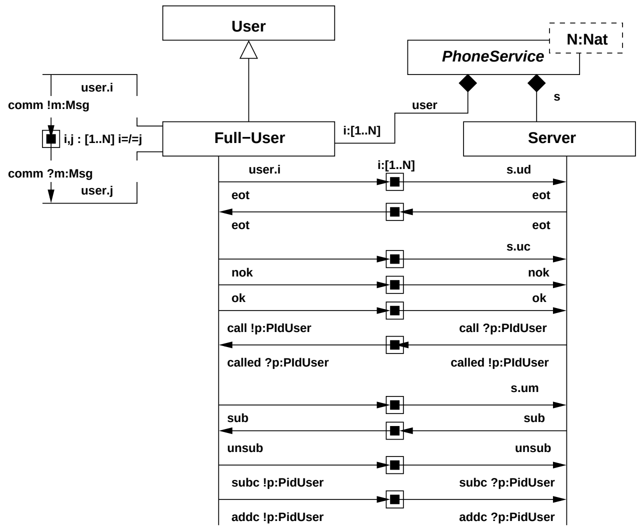

Figure 16. Phone Service composition.

inputs and produces outputs. Its applications consider services as well as routing problems. However this model lacks of a general and formal means to describe protocols like state diagrams. It uses Promela and Z specifications [10], we rather use symbolic transition systems and algebraic specifications in Korrigan. Our model is a formal model which is able to design the DFC model.

D. Jackson and M. Jackson in [9] argue that decomposition is twofold: mastering the complexity of systems and reusing well-known solutions of some subproblems. They also claim the need of a separation of concerns. We completely agree with these ideas. Their model of views is different from ours because it is at a higher level (identify problems and choose adequate solutions). An improvement of our approach would be to exploit their views.

In [1] Büchi and Sekerinski exhibit the benefits of using

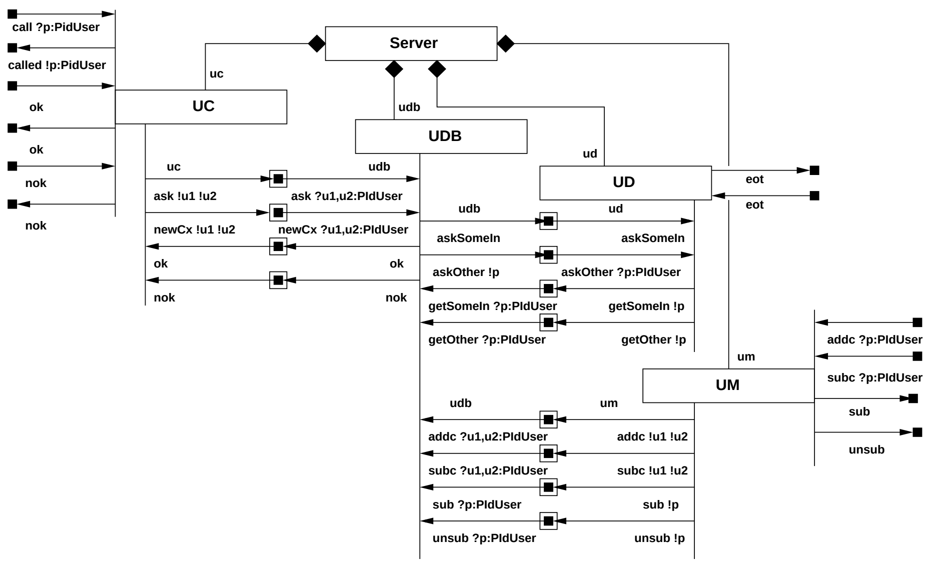

Figure 15. Server composition.

formal methods for constructing and documenting component software: *Formal specifications provide concise and complete descriptions of black-box components and, herewith, pave the way for full encapsulation*. We completely agree with the fact that current interface-description languages (such as IDL for CORBA) are limited to expressing syntactical aspects of functional code. The authors propose the notion of formal contract based on pre and post-conditions, but they critisize their popular form (as appear in [11]) and since they are only checked at run-time, they do not prevent from errors like formal proofs or static analysis. Moreover, they are checked at run-time during test runs, but for the production version these checks are often disabled for efficiency reasons. Note that Korrigan uses pre and post-conditions in this way but contrary to this work we do not have yet a notion of formal refinement.

An important milestone in the area of architecture and formal specification is [18]. Architectural descriptions are important for software engineers. The first use is to define idioms or patterns to communicate the overall architecture of a system. The second is that it serves as a skeleton to express and prove system properties. Our approach stands in the formal language for architecture category. One important task in defining an Architectural Description Language (ADL) is to choose the underlying model. As we saw earlier our ADL has multiple views and the notion of connector correspond to the glue in Korrigan.

Korrigan is supported by a UML-inspired graphical notation. We suggest, when possible, to reuse the UML notation, but we also have some proper extensions. Since our model is component-based, we have an approach which is rather different from UML on communications and concurrent aspects. Thus we also describe specific notations to define dynamic interfaces of components, communications patterns and concurrency.

Our concerns about methods and graphical notations for formal languages are close to [14, 4] ones. However, we think we can reuse UML notations, or partly extend it using stereotypes, rather than defining new notations. Moreover, our approach is complementary to the theoretical approaches that try to formalize the UML. Our notations are also more expressive and abstract than [14] as far as communication issues are concerned.

Korrigan and UML-RT [16] partly address the same issues : architectural design, dynamic components and reusability. However, UML-RT is at the design level whereas Korrigan is rather concerned about (formal) specification issues. There are also some other differences, mainly at the communication level, but the major one is that, to the contrary of UML-RT, Korrigan provides a uniform way to specify both datatypes and behaviours.

Our notation for the glue between communicating components may be also related to [6]. The main differences are that our glue is more expressive than LOTOS synchronizations, and that we have a more structured organization of communication patterns.

5. Conclusion

We defined in previous works [13, 2] a formal approach based on view structures for the specification of mixed systems with both control, communications and data types. The corresponding formal language, Korrigan, is based on Symbolic Transition Systems, algebraic specifications and a simple form of temporal logic. It allows one to describe systems in a structured and unifying way.

We wish to promote the use of formal methods in the industrial world. Therefore, we have built a software environment, ASK [3], for the development of our Korrigan specifications. An important feature of this environment is the ability to obtain concurrent object-oriented code (Active Java) from the Korrigan specifications [13].

In this paper we have shown that Korrigan is well suited to the definition of formal components. Moreover, it promotes the specification of reusable components. We have means to compose heterogeneous components, a simple form of inheritance, a powerful notion of genericity and the ability to describe communication patterns. Korrigan provides clear dynamic interfaces and means to structure complex systems with both data parts and behavioural parts. Hence, it is also suited to the definition of the architecture of complex systems. Korrigan supports both textual and graphical notations which is important for automatic processing, tools and readability.

We are now working on validation and verification procedures for our Korrigan specifications. Due to the use of STS, *i.e.* Symbolic Transition Systems, such procedures have to be adapted [8, 17]. We also investigate the automatic translation of Korrigan specifications into PVS.

References

[1] M. Buechi and E. Sekerinski. Formal Methods for Component Software: The Refinement Calculus Perspective. In J. Bosch and S. Mitchell, editors, *Object-Oriented Technology*, volume 1357, pages 332–337. Springer-Verlag, 1998.

[2] C. Choppy, P. Poizat, and J.-C. Royer. A Global Semantics for Views. In T. Rus, editor, *International Conference on Algebraic Methodology And Software Technology (AMAST'2000)*, volume 1816 of *Lecture Notes in Computer Science*, pages 165–180. Springer-Verlag, 2000.

[3] C. Choppy, P. Poizat, and J.-C. Royer. The KORRIGAN Environment. *Journal of Universal Computer Science*, 7(1):19–36, 2001. Special issue on Tools for System Design and Verification.

[4] E. Coscia and G. Reggio. JTN: A Java-Targeted Graphic Formal Notation for Reactive and Concurrent Systems. In J.-P. Finance, editor, *Fundamental Approaches to Software Engineering (FASE'99)*, volume 1577 of *Lecture Notes in Computer Science*, pages 77–97. Springer-Verlag, 1999.

[5] C. Fischer. CSP-OZ: a combination of Object-Z and CSP. In H. Bowman and J. Derrick, editors, *Proc. 2nd IFIP Workshop on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pages 423–438, Canterbury, UK, 1997. Chapman & Hall.

[6] H. Garavel and M. Sighireanu. A Graphical Parallel Composition Operator for Process Algebras. In *Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification (FORTE/PSTV'99)*, 1999.

[7] D. Harel. On Visual Formalisms. *Communications of the ACM*, 31(5):514–530, 1988.

[8] M. Hennessy and H. Lin. Symbolic Bisimulations. *Theoretical Computer Science*, 138(2):353–389, 1995.

[9] D. Jackson and M. Jackson. Problem decomposition for reuse. *IEEE/BCS Software Engineering Journal*, 11(1):19–30, Jan. 1996.

[10] M. Jackson and P. Zave. Distributed feature composition: A virtual architecture for telecommunications services. *IEEE Transactions on Software Engineering*, 24(10):831–847, Oct. 1998.

[11] B. Meyer. *Object-oriented Software Construction*. International Series in Computer Science. Prentice Hall, 1988.

[12] P. Poizat. KORRIGAN*: a Formalism and a Method for the Structured Formal Specification of Mixed Systems*. PhD thesis, Institut de Recherche en Informatique de Nantes, Université de Nantes, Dec. 2000. in French.

[13] P. Poizat, C. Choppy, and J.-C. Royer. From Informal Requirements to COOP: a Concurrent Automata Approach. In J. 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.

[14] G. Reggio and M. Larosa. A graphic notation for formal specifications of dynamic 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 40–61. Springer-Verlag, 1997.

[15] J.-C. Royer. Formal Specification and Temporal Proof Techniques for Mixed Systems. In *Proceedings of the IPDPS Conference, FMPPTA*, San Francisco, USA, 2001. IEEE.

[16] B. Selic and J. Rumbaugh. Using UML for Modeling Complex Real-Time Systems. Technical report, Rational Software Corp., 1998.

[17] C. Shankland, M. Thomas, and E. 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.

[18] M. Shaw and D. Garlan. Formulations and Formalisms in Software Architecture. In J. vanLeeuwen, editor, *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 307–323. Springer-Verlag, 1995.

[19] G. 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.

[20] P. Zave and M. Jackson. A component-based approach to telecommunication software. *IEEE Software*, 15(5):70–78, Sept./Oct. 1998.