KORRIGAN: a Formal ADL with Full Data Types and a Temporal Glue

Pascal Poizat [1], **Jean-Claude Royer** [2]

[1] LaMI - UMR 8042 CNRS, Genopole et Université d'Évry Val d'Essonne
Tour Évry 2, 523 Place des terrasses de l'Agora,
91000 Évry, France
email: poizat@lami.univ-evry.fr

[2] École des Mines de Nantes
4 rue Alfred Kastler, B.P. 20722, 44307 Nantes
Cedex 3, France
email: Jean-Claude.Royer@emn.fr

— SPECIF —

RESEARCH REPORT

N° 88-2003

September 2003

Pascal Poizat, Jean-Claude Royer

KORRIGAN*: a Formal ADL with Full Data Types and a Temporal Glue*

28 p.

Les rapports de recherche du Laboratoire de Méthodes Informatiques sont disponibles
aux formats PostScript® et PDF® à l'URL :
ftp://ftp.lami.univ-evry.fr/pub/publications/reports/index.html

Research reports from the Laboratoire de Méthodes Informatiques are available in PostScript® and PDF® formats at the URL:
ftp://ftp.lami.univ-evry.fr/pub/publications/reports/index.html

KORRIGAN: a Formal ADL with Full Data Types and a Temporal Glue

Pascal Poizat, Jean-Claude Royer

Abstract

The KORRIGAN formalism is devoted to the structured formal specification of mixed systems through a model based on a hierarchy of *views*. In our unifying approach, views are used to describe the different aspects of components (data, behaviours, architecture, communication). While our model is primarily devoted to mixed structured specifications it is a true Architectural Description Language (ADL). As quoted in [40], an ADL must address the component, connector and configuration concepts. In this paper we will demonstrate that KORRIGAN may be used at the formal specification level to describe them. In addition to existing approaches KORRIGAN provides full data types, temporal logic gluing facilities, structuring and readability. We will also discuss related issues such as verifications and tools, and related work.

Additional Key Words and Phrases: Architectural Description Language, Software Component, Mixed Formal Specification, Graphical Notation, Symbolic Transition System, Temporal Glue

Contents

1 Introduction

The need of architectural descriptions, *i.e.* overall, abstract and structured descriptions of software systems, is well-recognized in software engineering, see [56, 57] for instance. An architecture provides the main elements and the main links or communications of the system. 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 of 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/ports, communications, protocol, ...). 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, checker, prover, ...), which is only possible with formal approaches.

An Architectural Description Language (or ADL) is a language that provides features for modelling a software system conceptual architecture, it is distinct from the system implementation. These notions exist since a long time in software engineering, but recently several proposals have been suggested both in the formal specification area and the object-oriented community. Formal software architecture is a level of specification that involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition and constraints on these patterns. Formal ADL must provide concrete syntax and formal semantics for characterising architectures.

The use of formal specifications is now widely accepted in software development to provide abstract, rigorous and consistent 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, communication) appeared. This issue was addressed in approaches combining algebraic data types with other formalisms (*e.g.* LOTOS [31] with process algebras or SDL [7] with State/Transition Diagrams), and also more recently in approaches combining Z and process algebras (*e.g.* OZ-CSP [53] or CSP-OZ [24]). This is also reflected in object oriented analysis and design approaches such as UML [58] 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. Some approaches encompass both aspects within a single framework (*e.g.* LTS [47], rewriting logic [**?**] or TLA [35]). These "homogeneous" approaches ease the verification and the definition of consistency criteria for the integration of aspects, but at the cost of a loss of expressiveness for one of the aspects or a poor level of readability. This is also the case with CCS or CSP based proposals, as WRIGHT ([4]) for example.

In this paper our aim is to present a state of the art of the KORRIGAN approach and its ability to formally design mixed architectures and mixed components. The KORRIGAN formalism is devoted to the structured formal specification of mixed systems through a model based on a hierarchy of *views*. Our approach aims at keeping advantage of the languages dedicated to each 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 and unifying framework accompanied by an appropriate semantic model. Moreover, experience has shown that our formalism leads to expressive and abstract, yet readable specifications. We suggest to reuse some UML notations, but we also define some proper KORRIGAN graphic notations. Since our model is component based we have an approach which is rather different from UML on communications and concurrent aspects. Thus we use specific notations to define the dynamic interface of a component, its communications with others and concurrency. This solves the lack quoted in several work ([39, 17, 43]): the UML notation is not really adequate to design concurrent architectures. We illustrate the KORRIGAN approach (both in textual and graphical notation) on the FM'99 cash-point service case-study benchmark [55].

The paper is organised as follows. Section 2 presents the cash-point service case study. Section 3 gives an overview of the KORRIGAN view model. Section 4 shows how KORRIGAN matches the usual value criteria of

architectural languages and discusses related aspects. Finally, some related work is discussed and a conclusion finishes this presentation.

2 The Case Study

Our case study is an extension of the FM'99 cash-point service benchmark [55]. The system is composed of several tills which can access a central resource containing the detailed records of customers' bank accounts. A till is used by inserting a card and typing in a Personal Identification Number (PIN) which is encoded by the till and compared with a code stored on the card. After successfully identifying themselves to the system, customers may either (i), make a cash withdrawal or (ii), ask for a balance of their account to be printed. Information on accounts is held in a central database and may be unavailable in case of network failure. In such a case, actions (i) and (ii) may not be possible. If the database is available, any amount up to the total in the account may be withdrawn, subject to a daily limit on withdrawals. This means that the amount withdrawn within a day has to be stored on the card. The daily limit is specific to each customer and a part of its bank account record. Another restriction is that the amount of the withdrawal is not greater than the value of the till local stock.

The till may keep "illegal" cards, *i.e.* cards which fail a key checking. Each till is connected to the central by a specific line, which may be down or up. The central handles multiple and concurrent requests. Once a user has initiated a transaction, it is eventually completed and preferably within some real time constraint. A given account may have several cards authorised to use it.

With reference to the original case study we add the following extensions: a management of the local stock of cash within the till, a daily limit which is specific to each customer. In Section 4.4, we will also consider an extension of the network with a multiplexer. We did not take into account the real time constraints.

3 The KORRIGAN Model

In this Section, we will briefly present our formal model, focusing on expressiveness issues. More details on semantics are discussed in [15, 2] and environment (methods and tools) in [45, 14, 17, 18].

Our model is based on 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. KORRIGAN supports both textual and graphical notations which is important for automatic processing, tools and readability.

3.1 The View Hierarchy

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 component may be either a mixed component (integrating different internal structuring views in an *Integration View*), or a composite 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. The whole class diagram for the view model is given in Figure 1.

3.2 Component Interfaces

Components (both basic and composed ones) are represented using *boxes* with well-defined interfaces. These interfaces use *dynamic signatures, i.e.* event ports names with offer parameters (as in LOTOS) and interaction typing (as in SDL). A value emission of type `T` is noted `!T` and a value receipt `?T`. The communication interface symbols we use are described in Figure 2.

The till component interface is described in Figure 3.

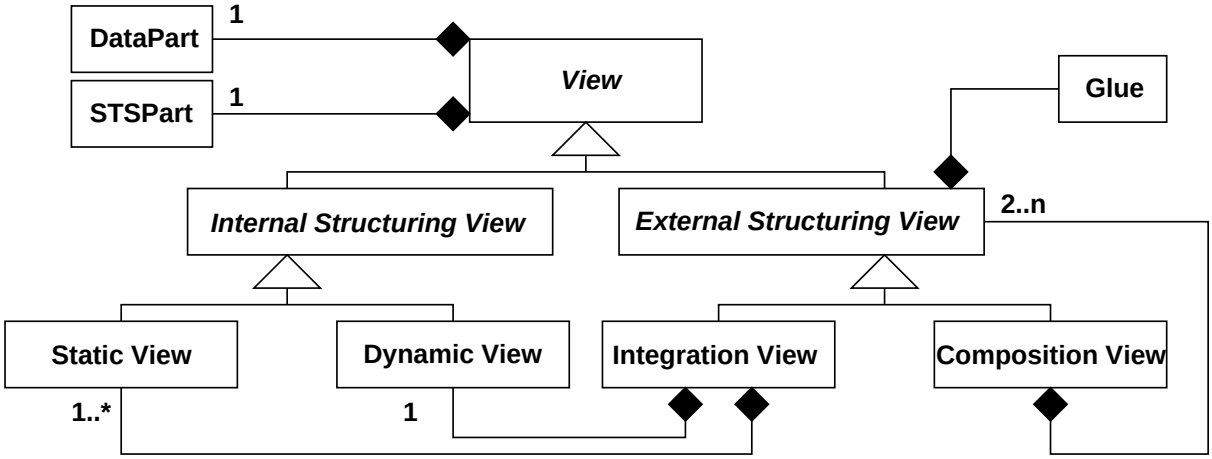

Figure 1: Views Class Hierarchy (UML notation)

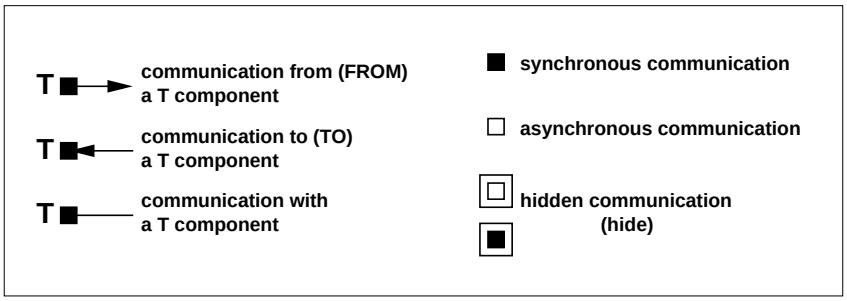

Figure 2: Notations for Dynamic Interfaces

It has several event ports, `card ?Card` to insert the user card, `card !Card` to eject the card, `pin ?PinNumber` to enter the PIN, `cash !Money` to get money, `add ?Money` to add some money, `receipt` to print a ticket, `rec ?Msg` to receive a message from the connection, and `send !Msg` to send a message.

3.3 Genericity and Patterns

Component interfaces may be generic on (possibly constrained) data values (*i.e.* constants, *e.g.* `N,M:Natural {1<N<M}`, Fig. 12), data types (*e.g.* `MSG:Sort`, Fig. 6), event ports (*e.g.* `G`, Fig. 10) and component types (*e.g.* `Server`, `Client`, Fig. 10). As in UML, this is denoted by dashed boxes in the top corners of the components (top left for event ports and top right for data values, data types and component types).

In our example, the till component is generic on a `MsgConnection` component type. This constrains it to be glued, as far as the `send` and `rec` event ports are concerned, with a `MsgConnection` component or any component that inherits from it (`DropMsgConnection`, Figure 9). An example of genericity on data types is given in Figure 6, where the `MsgConnection` and `DropMsgConnection` components are generic on the type of messages which transit in the connection.

In conjunction with genericity on data types and component types, genericity on event ports yields the expressiveness of KORRIGAN to express patterns. As in LOTOS and object-oriented programming some idioms or patterns may be used in KORRIGAN to describe general and common architectures of systems. Patterns will be dealt with by in more details in Section 4 (see Fig. 10 and Fig. 11 for example) where we present the expressiveness of KORRIGAN as an ADL.

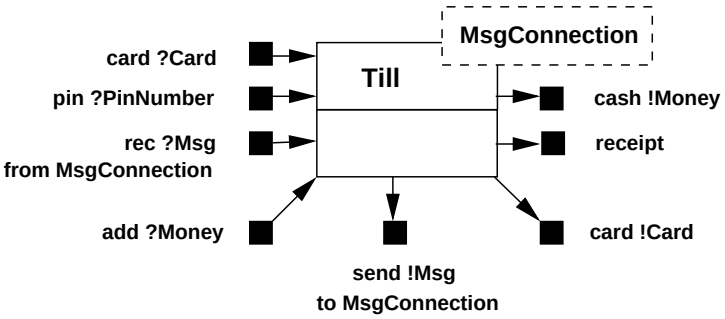

Figure 3: Component Interface Diagram of the Till.

3.4 Simple Components

Simple components or sequential components are described with two aspects: a dynamic behaviour and an abstract data type description. The next subsections illustrate the presentation of these two aspects in KORRIGAN.

3.4.1 Symbolic Transition Systems (STS)

The dynamic behaviour of the till is depicted in the Figure 4. STSs are mainly finite transition systems with guarded transitions and open terms in states and transitions. As in LOTOS non-determinism is possible *via* the `i` action. One transition like `cash !sum ; receipt / giveCash(self, sum)` means that the till emits a sum of money then it prints a ticket. During this transition the `giveCash` operation updates the amount of money of the till data type.

The main interest with these transition systems ([15]) is that (i) they avoid state explosion problems, and (ii) they define equivalence classes (one per state) and hence strongly relate the dynamic and the static (algebraic) representation of a data type. STSs may be related to Statecharts [28] (as least graphically speaking) but for some differences:

- STSs are less expressive than Statecharts. However various simplifying notations have been defined for our STSs, such as activities or hierarchical states, see [44];
- STSs model sequential components, concurrency is done through external structuring and the computation of a structured STS from subcomponents STSs [15];
- STSs are built using conditions which enable one to semi-automatically derive them from requirements (see [45]);
- There is a strong link between the STS and its underlined data type. Their integration with formal data types is formally defined [15, 6]. STSs may be seen as a graphical representation of an abstract interpretation of an algebraic data type.

STSs provide an expressive and abstract means to describe symbolically dynamic behaviours. The description of an STS may be given either in its graphical form or in its textual form [44] (better suited for tool processing).

3.4.2 Abstract Data Types

A data type, `Till` (Fig. 5) in the Till example, is associated with each STS and transitions may execute actions on it. Here we consider a simple approach but more complex situations may be modelled following the Graphic Abstract data Type principles [49]. The data type has two generators, `newTill : Amount -> Till` and `insertCard : Till, Card -> Till`. The operations semantics are described using algebraic axioms. Note that B machines or Z schemas may be used instead or in conjunction with these algebraic specifications following the [6] principles. For example, axiom 6 in Figure 5 means that the amount of money is decreased of

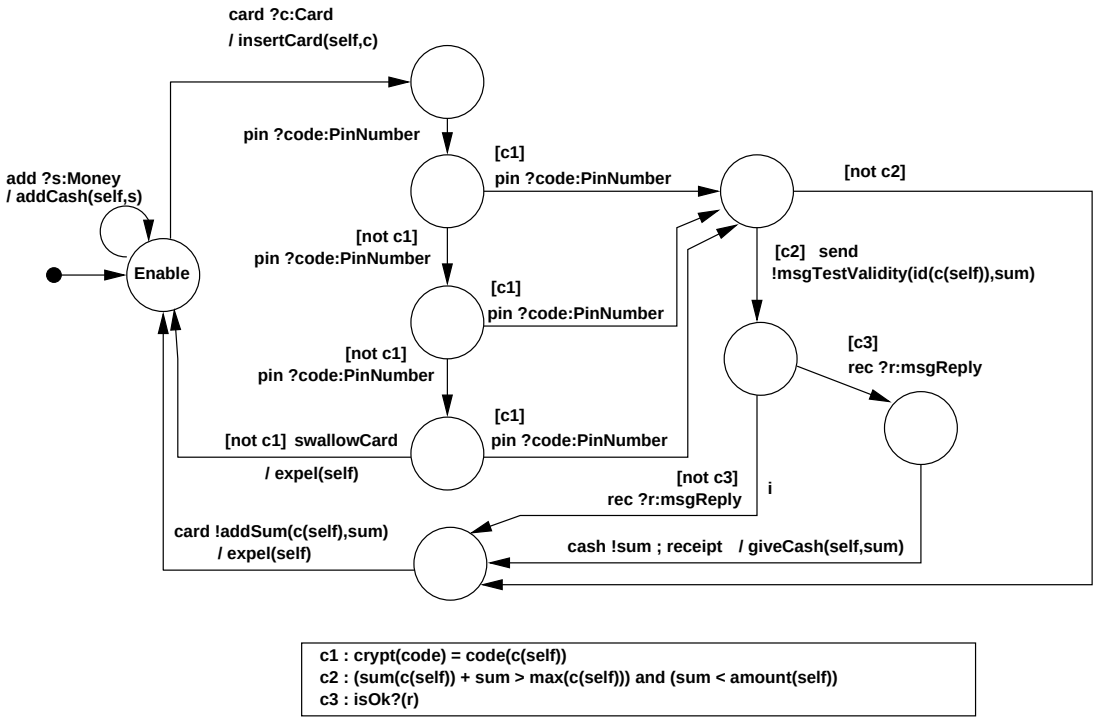

Figure 4: Symbolic Transition System of the Till.

```
-----------------------------------------------------------------------
Till data type
Forall t : Till, c : Card, a, s : Natural

1: card(insertCard(t, c)) = c
2: amount(newTill(a)) = a
3: amount(inserCard(newTill(a), c)) = a
4: expel(insertCard(newTill(a), c)) = newTill(a)
5: addCash(insertCard(newTill(a), c), s) = newTill(a+s)
6: giveCash(insertCard(newTill(a), c), s) = insertCard(newTill(a-s), c)
-----------------------------------------------------------------------
```

Figure 5: Axioms for the Till Data Type.

sum `s`. As in [49] the signature of the data type is computed from the STS. Thus from the STS and the axiom description a complete data type description may be built.

3.5 Inheritance

In software engineering, inheritance is one of the key concepts that enable the reuse of components. Inheritance [41] enables one to add methods to a class, and allows overloading and masking. Inheritance may also be used to add or strengthen constraints. In KORRIGAN we provide a simple form of inheritance for internal structuring (*i.e.* basic) views. Two views related by inheritance must be of the same type, for example a dynamic view can only inherit from another dynamic view. Moreover, our inheritance semantics is restricted to the addition of new conditions and new operations in the subview. It does not allow overloading nor masking since they yield semantic complexity. The subview defines super-states and each super-state inherits the complete dynamic behaviour of the superview. This rather strict inheritance constraints simplify the dynamic descriptions of views, allows subtyping and ensure some kind of behavioural compatibility [42, 54].

An example of inheritance in KORRIGAN is given in Figure 6 where the `MsgConnection` (media) and the `DropMsgConnection` (media with failure) are related.

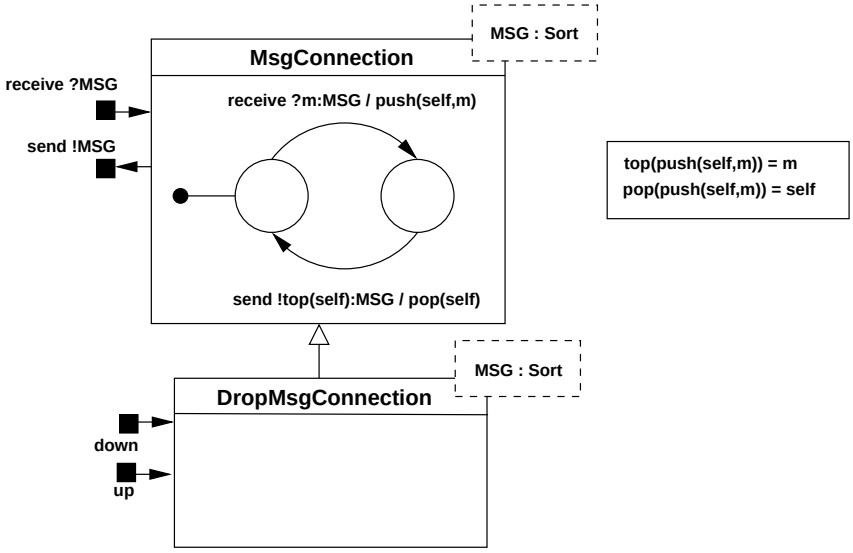

Figure 6: Inheritance Media / Media with Failure

Failure is modelled by a special `down` port. This method may also be used to take into account the creation or deletion of components. This solution, which is acceptable in a synchronous context, is however not realistic with asynchronous communications. This aspect is one possible future improvements of our model.

The inheritance relation between the components behaviours (STSs) is given in Figure 7.

3.6 Compositions

Components are "glued" altogether in External Structuring Views (Fig. 8) using both axioms and temporal logic formulas. This glue corresponds to a generalised form of synchronous product (for STS). Its 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 components with operations defined in other components. The Φ and Φ_0 elements are temporal formulas expressing correct combinations of the components states (Φ) and initial ones (Φ_0). The Ψ element is a set of temporal formulas expressing what transitions have to be triggered at the same time (this expresses *rendez-vous*). The `COMPOSITION` clauses may use a syntactic sugar: the *range* operator (`i:[1..N]` or

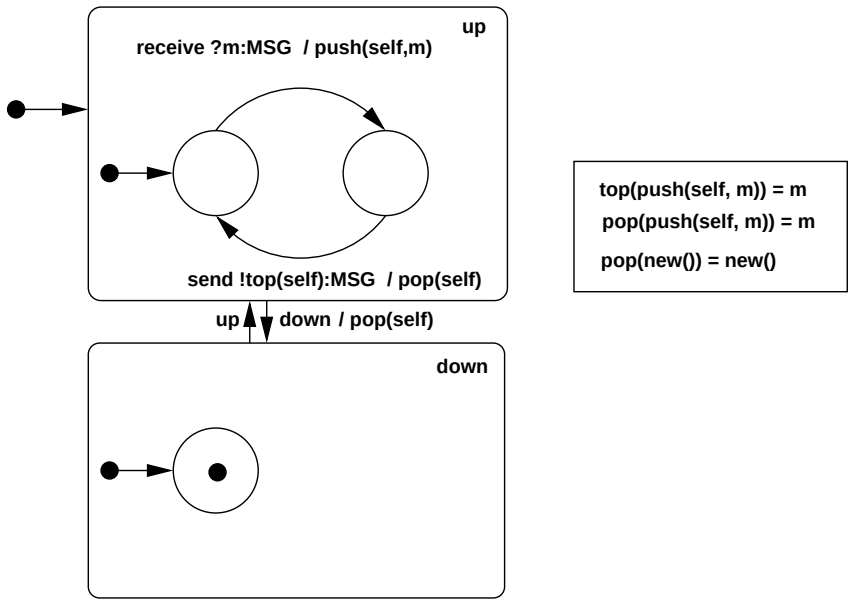

Figure 7: STS of the Media with Failure

`i:{`e_1`,...,`e_n`}`), a bounded quantifier. The range operator may be associated either with a universal quantifier ($\forall$ in formulas, `ALL` in figures) or with a disjunction quantifier ($\oplus$ in formulas, `XOR` in figures). Their meaning is the following one:

$$\begin{array}{lcl} \forall i : [1..N]\phi_i & \Leftrightarrow & \phi_1 \wedge \ldots \wedge \phi_N \\ \oplus i : [1..N]\phi_i & \Leftrightarrow & (\phi_1 \wedge \neg\phi_2 \wedge \ldots \wedge \neg\phi_N) \vee \ldots \vee (\neg\phi_1 \wedge \neg\phi_2 \wedge \ldots \wedge \phi_N) \end{array}$$

Since integration views may be seen as a simple case of composition view, we will only present composition views in the sequel.

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

Figure 8: KORRIGAN Syntax (compositions)

This KORRIGAN glue enables one to express things such as:

- synchronizing: $c_1.a \Leftrightarrow c_2.a$ (c_1 and c_2 must synchronize on a)
- event ports connection: $c_1.a_1 \Leftrightarrow c_2.a_2$ (ports may have different names)
- value passing: $c_1.a?x : T \Leftrightarrow c_2.a!y$
- composite event exportation: $self.a \Leftrightarrow self.sub.a$ where $self$ denotes the current component, being composed of a sub subcomponent
- broadcasting (1 to N, all in the same time): $\forall i : [1..N](server.send \Leftrightarrow client.i.receive)$
- exclusive peer to peer (1 to 1): $server.send \Leftrightarrow \oplus i : [1..N](client.i.receive)$

- exclusive states: $\neg(cooler_1.@on \wedge cooler_2.@on)$, the two coolers may not be in their on state at the same time

The KORRIGAN glue is expressive yet readable and enables us to link symbolic transition systems to denote synchronizations and communications in an abstract and concise way. This glue may represent temporal formulas over state or transitions, but it may also denote complex synchronizings between sequences of states or transitions as in Figure 11. Some formulas have a graphical counterpart (Fig. 10).

A component interface may be associated with a composition by exporting (see example above) some events of its subcomponents. Hence, compositions are components too, and genericity and pattern issues (which we dealt with for basic components in Section 3.3) apply to them.

The system architecture of the till system is described by a composition view, given in its graphical form, in Figure 9. Glue rules for this architecture will be presented in the next Section. The overall system is made up of `N` till lines (*i.e.* a till and its `DropMsgConnection` specialized communication line) and the bank with its bank interfaces. In composition views we reuse aggregation and template notations from UML but with our additional range notation. Note also the inheritance relation between `DropMsgConnection` and `MsgConnection`.

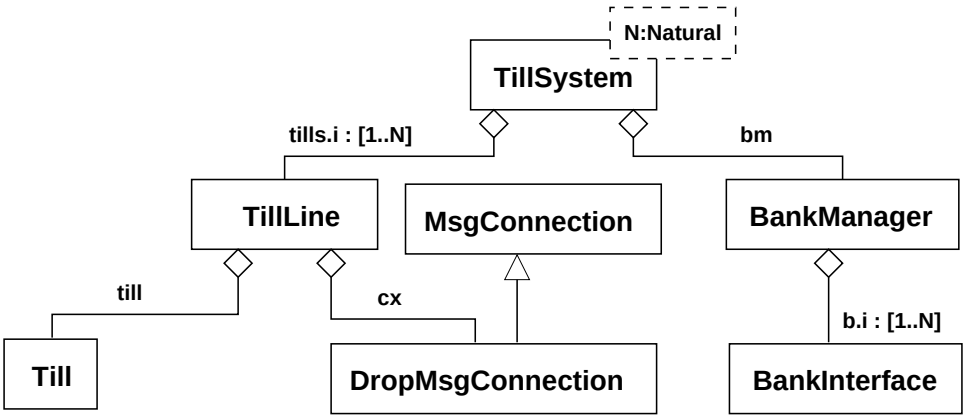

Figure 9: The System Architecture.

4 KORRIGAN : a Formal Architectural Language

The component concept is not new in software engineering. It appeared in Simula [33], in Modula [59] and in the Booch's approach [10, 34]. However it recently re-emerged both in the object-oriented community [23, 22] and in the formal specification area [38, 52, 13, 32, 60, 4, 5, 56]. This concept is strongly related to the architecture concept which is needed to build reusable and "visual" software architectures. An important remark is that components arise in all the steps of the development process, sometimes overlapping:

- Coding: Entreprise Java Beans, COM, Corba Components, ..., [23, 22, 27],
- Design: UML-RT and UML 2.0 [50, 12],
- Specification: Olan, Rapide, [8, 36, 1],
- Formal specification: Darwin [38], WRIGHT [4], KORRIGAN [44]

The aim of this Section is to demonstrate that KORRIGAN is a true architectural language which may be used at the formal specification level. This Section also synthesises and extends several work related to components and KORRIGAN [17, 16, 46]. In the design of KORRIGAN we have focused on clear and typed interfaces, separation of concerns, explicit and abstract glue between components, formality and verification means. In addition to current ADLs, KORRIGAN provides full data types, readability, and expressive structuring and temporal logic features.

Our work does not yet consider other non functional properties such as persistence, security, deployment and others which are generally covered by lower level architectural languages.

Following Medvidovic and Taylor's state of the art [40], we consider that an ADL must have features for first class components, first class connectors, and configurations. Quoting [40], "A *component* is a unit of computation or a data type, they are loci of computation and state". A component has an *interface* which is a set of interaction points between it and the external world. *Connectors* are architectural building blocks used to model interactions among components and rules that govern those interactions. *Architectural configurations*, or topologies, are connected graphs of components and connectors that describes architectural structures. Generally one has the need for more than components, and needs generators of components or *component types*. The same applies as well to connectors and configurations. Component types are abstractions that encapsulate functionality into reusable blocks. They may be parameterized and instantiated several times.

We now address the mechanisms to define components, connectors and configurations in KORRIGAN. We will also discuss several related aspects, mainly dynamic reconfiguration, semantics and verification means, and tool support.

4.1 Components

KORRIGAN is particularly well-suited for the design of components. Like in other architectural languages, KORRIGAN provides the designer with first class components, both basic or composite ones. The notion of elementary components is provided by internal structuring views. Components with both static and dynamic aspects may be defined using integration views. The use of interfaces, STS and abstract data types enable one to give different detailed descriptions of contracts for components. Composite components may be taken into account by composition views.

4.2 Connectors

There is no explicit connector in KORRIGAN which rather defines a powerful means to glue things altogether, enabling specific connectors to be defined as particular (generic) components or component patterns. This treatment of connectors is completely uniform with remarks from [40] which state that connectors are a particular kind of component used to model interactions which, however, may not correspond to a compilation unit in the implemented system. To complete this remark we may quote [5], "explicit connectors are better for intuitive expression of architecture, implicit connectors as components provide a simpler formal semantics". In KORRIGAN we have clear typed interfaces and an abstract glue to express interactions, which provides a good balance between formality and expressiveness without any sacrifice.

A connector may then be seen as a generic pattern resulting form the abstraction over the subcomponents of an architecture. As an example, let us consider a kind of star-topology network where the clients may communicate as in a ring network (Fig. 10) . The architecture is made up of a server and several clients. The clients are concurrent and the i^{th} client communicates with its two neighbours using the two event ports `NEXT` and `PREV`. The server communicates with a client using the specific ports `G` and `H`. The `StarRing` component (Fig. 10) is generic on the number of clients, on the server and the client component types, and on the `G`, `H`, `NEXT`, and `PREV` event ports. The server and the clients sub components are generic on the specific ports needed in the communications. This simplifies the pattern enabling to avoid values emissions and receipts in the synchronizing glue.

Instead of using a rigid encoding of the dynamism in the state behavior of the composition components, KORRIGAN chooses to use a separated glue *i.e.* external to the components and put in an external structuring view (Fig. 8). For example, in the Ψ part, one may write $(s.G \Leftrightarrow c.i.H)$ to synchronize `G` of `s:Server` and `H` of `c.i:Client`. To formally express that communications are exclusive between the server and its clients (*i.e.* one client treated at a time), we extend this formula using the $\oplus$ operator: $s.G \Leftrightarrow \oplus i : [1..N]\ (c.i.H)$. In most cases, this glue has a graphical counterpart which may enrich composition diagrams to lead to true communication diagrams. For example the previous textual synchronizing may be graphically represented, as in Figure 10, by a line joining the two components with a $\oplus$ (textually: `XOR`) quantifier. The `ALL` ($\forall$) quantifier in the same picture denotes that communications between every client and its two neighbours may arrive at any logical instant.

The generic component of Figure 10 allows on to instantiate `StarRing` connectors. One first has to give a concrete server component and several concrete clients components, and then instantiate the different event ports parameters.

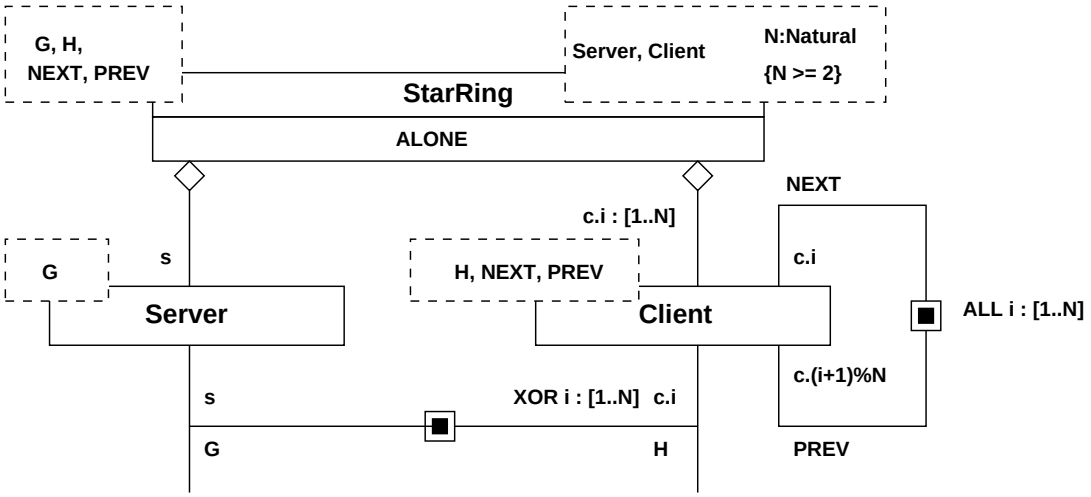

Figure 10: StarRing Connector Pattern in KORRIGAN

This example shows that connectors naturally arise in KORRIGAN thanks to its first class components and its abstract and powerful glue.

4.3 Configurations

The notion of configuration is dealt with by KORRIGAN external views as demonstrated in example Figure 9 which represents the connection between several components. As previously explained the notion of connector is implicit in this figure. A configuration is a set of typed components and corresponding variable names appearing on the aggregation links.

4.4 Dynamic Reconfigurations

Dynamic reconfiguration means to adapt the architecture to changes in a system. A simple example, coming from the literature [5], is the Faulty-Tolerant Client-Server (FTCS). It is a classic client-server architecture with an auxiliary server which will be used whenever the main server goes down. A KORRIGAN architecture for the FTCS is given in Figure 11.

In this example, a complex synchronizing is needed. When the main server is on (*i.e.* the formula $main.@on$ yields) then the server and the client synchronize on G/H to achieve some service. If the main server is off and the auxiliary server is on then the same is achieved using the auxiliary server in place of the main server. The corresponding textual synchronizing connects the three components and glues the corresponding transitions:

$$(main.@on \wedge (main.G \Leftrightarrow \oplus i : [1..N]c.i.H)) \vee$$
$$(main.@off \wedge aux.@on \wedge (aux.G \Leftrightarrow \oplus i : [1..N]c.i.H))$$

Most of the client-server pattern is kept, but we extended it to a faulty-tolerant client-server. However, dynamic reconfiguration often yields more complex situations. There are two non exclusive cases:

1. dynamic modifications of the communications (as in the faulty example),

2. dynamic modifications of the set of components (as in a phone system or a transit node).

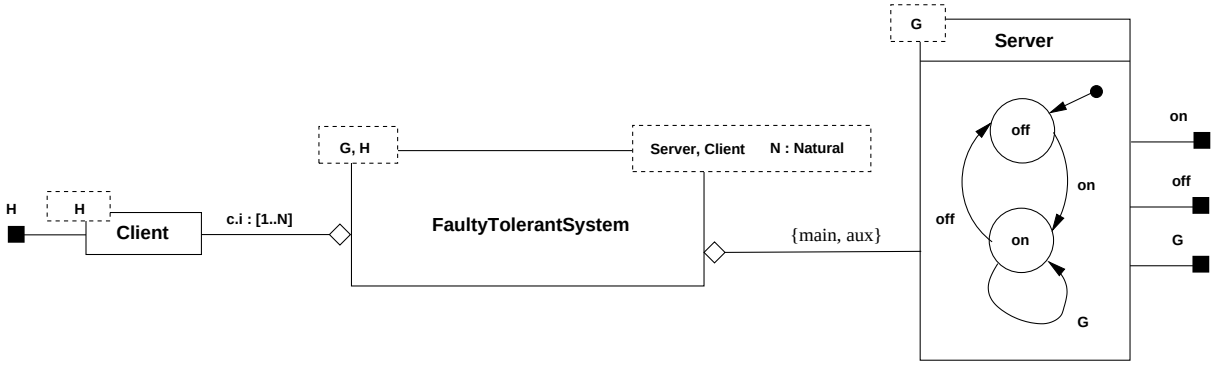

Figure 11: The Faulty-Tolerant Client-Server in KORRIGAN

Case 2 is more complex. To demonstrate the ability of KORRIGAN to represent the second case, let us take a modified version of the till system. The tills may be up or down, we reused the artifact of Section 3.5. $N-1$ tills are connected by a dedicated line. The N^{th} line is used to connect a multiplexer with some additional tills (from N to M). The interface of the multiplexer has to be the same as a till line. However, its dynamic behaviour is different.

Figure 12 represents an architecture with `N-1 TillLines` and the additional multiplexed `Tills` connected to the system.

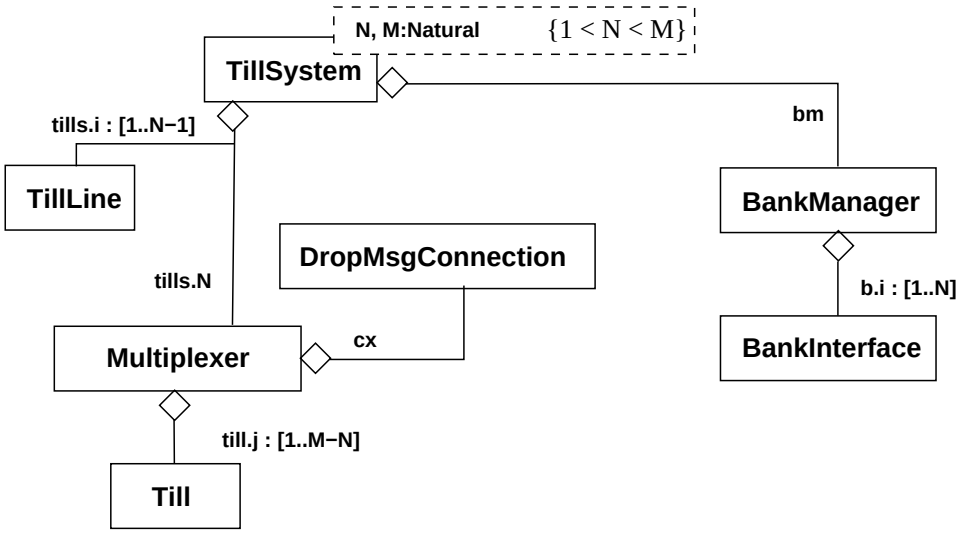

Figure 12: An Architecture with a Multiplexer

The glue in the multiplexer has to connect any of its connected tills with the `b.i.N` interface and this link must be valid during a complete cycle of `send` and `receive` to avoid the interleaving of communications between different tills and the rest of the system. Such treatment of cycles may be dealt with in KORRIGAN using the binding $\overset{@}{\exists}$ operator and the modal $[\_]$ operator.

The glue of the multiplexer for one till (the j^{th}) and without failure is:

$$\Phi_j = \overset{@}{\exists}\, s. \quad \begin{array}{ll} [(self.till.j.send \Leftrightarrow self.cx.receive)] & (\phi 1) \\ [(self.cx.send \Leftrightarrow self.send)] & (\phi 2) \\ [(self.receive \Leftrightarrow self.cx.receive)] & (\phi 3) \\ [(self.cx.send \Leftrightarrow self.till.j.rec)] & (\phi 4) \\ s & \end{array}$$

where $\overset{@}{\exists}\, s.[\phi_1][\phi_2]s$ denotes a cycle starting from a given state (denoted by s), satisfying ϕ_1 and then ϕ_2, and ending in the s state. A graphical representation of formula Φ_j is given in Figure 13.

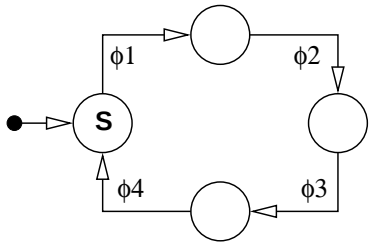

Figure 13: Graphical representation of the gluing formula

Formula Φ_j may be extended to take all the multiplexed tills into account using the exclusive disjunction operator:

$\Phi_{Multiplexer} = \oplus j : [1..M - N].\Phi_j.$

4.5 Semantics and Verification

An operational semantics for KORRIGAN has been proposed in [15]. The main idea is to define a semantics for STS with data types. This is done in a similar way than classic operational semantics for labelled transition systems. The main differences are 1) we need term rewriting and normal form to reduce algebraic terms, 2) we need structured state and transitions to take into account the structure of our STSs. This defines the general semantics for a view, then different cases are studied according to the views hierarchy of KORRIGAN (Fig. 1). For internal views the semantics is straight. For external views we must give a view structure to integration and compositions views. This is done by calculating a global data part and a global behavioural part from the description of the components and the glue. The global data part is based on a data product of the data components. The global behaviour is computed as the synchronous product of the component STSs, the glue defines the synchronizing rules. A complementary proposal [2] presents a denotational semantics for mixed specification languages (hence KORRIGAN, LOTOS and SDL are taken into account) together with a theory for refinement.

We have not yet a verification technique for full KORRIGAN. However, a simpler (as far as the glue is concerned) version of KORRIGAN may be translated into LOTOS, SDL or algebraic specifications for formal verification of the specifications (animation, test, model-checking). The problem is to allow dynamic verifications in presence of data types. Model-checking [19] is a relevant technique but it is limited to labelled finite state machines. Only few approaches [29, 51, 37, 21] are successful in this area. Application of the technique of [48, 3] is possible. The technique is based on symbolic transition systems and partial algebraic specifications. It uses Larch Prover or PVS to prove data properties as well as temporal logic ones.

One problem which is not yet solved is the problem of changing the number of components. Whenever the maximum number of components is known, a possible solution, is to create all the components at system initialisation. The components are then activated when they are needed in the system. However, in some real cases this hypothesis is too constraining. One solution seems to use π-calculus or one of its extensions as in [38, 1].

4.6 Tool Support

In order to promote the use of formal methods in the industrial world, we agree with [11]: *most important properties of specifications methods are not only the underlying theoretical concepts but more pragmatics issues such as readability, tractability, support for structuring, possibilities of visual aids and machine support*. Therefore, we have designed a prototype software environment, ASK [18], for the development of our KORRIGAN specifications. Tools are needed to help users in formal specifications, and [40] also quotes this fact for architectural languages. It considers several features: active specification, multiple views, analysis, refinement, implementation generation and dynamism.

In [18] we present an environment to support the use of KORRIGAN specifications. This work address some of the issues quoted above. The proposed environment supports our specific model, KORRIGAN, to specify mixed systems. This environment follows two principles: openness and extensibility. According to these principles, it provides translation tools to interface with other formalisms, *e.g.* LOTOS, LP, ... The first principle is to *interface*, in an open way, with some existing tools and environments, for example model checking tools (*e.g.* XTL in CADP [25]), theorem provers (*e.g.* the Larch Prover, PVS), and programming languages (*e.g.* JAVA, C++). Since such tools are numerous and evolve, our framework has to be *extensible*. A second principle is to provide *general tools* which can be useful to other environments or formalisms. For example the CLAP tool can be used to compute (a)synchronous compositions of any state-transition diagrams (automata, Petri Nets, symbolic transition systems). To achieve these two principles we reuse some object-oriented features both in the design and in the implementation of our environment. An important feature of this environment is the ability to obtain concurrent object-oriented code (Active Java) from the KORRIGAN specifications [14].

5 Related Work

There is several related work either on formal components and architectures, telecommunication services or object-oriented approaches. We compare KORRIGAN deeply with WRIGHT and UML, which seems to different, but relevant, ways in the area of architecture and specification.

5.1 Comparison with WRIGHT

At this stage it is interesting to compare our approach with WRIGHT [5, 4]. WRIGHT is a formal architectural description language with first class components and connectors. It may be seen as relook of CSP, since the notations and the semantics are inspired by this process algebra. An architecture has three parts. The first part defines component and connector types. A component has a set of ports and a behaviour part. A connector defines a set of roles and a glue specification. Roles are expected to describe the local behaviour of the interacting parts. The glue describes how the local activities of the different parts are coordinated. The second part is an instantiation of the first: *i.e.* component and connector instances. The third part describes how the instances are connected to define the actual system. WRIGHT also handles dynamic reconfigurations. Special events denote when reconfigurations are permitted and they are used in a separate view of the architecture. The configuration program describes how those events trigger reconfigurations.

The semantics of these constructions is defined by a translation into CSP. This has the advantages to get effective model checking for CSP and related work about behavioural refinement. However, most of these verifications are limited by the value-passing state explosion problem hence consider simple data types. WRIGHT proposes a deep analysis about automatic checking for architectural languages. It allows to check connector consistency, configuror consistency, and attachment consistency using mainly techniques to prove deadlock freedom and behavioural refinement.

In WRIGHT a connector explicits the protocol of the connected component and describe the expected global behaviour (the glue) of the system using process algebra expressions. KORRIGAN is simpler here since the glue denotes temporal logic properties of the global system not the global dynamic behaviour. KORRIGAN improves readability by graphic notations, this is important for large scale applications. WRIGHT is not really adequate for this due to several reasons: a poor graphic presentations and no n-ary composition. In our approach we consider both dynamic and functional properties not only dynamic properties with restricted data types. This is

a first important difference but another one, not the least, is the use of symbolic transitions systems for simple component behaviours. We are not restricted to finite dynamic systems as in WRIGHT. However this carries the problem of verifications, we have presented some solutions in Section 4.5.

5.2 Comparison With UML

KORRIGAN is supported by a UML-inspired graphical notation [17]. Indeed, when possible, we suggest 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 [47, 20] ones. However, we think we can reuse UML notations, or partly extend it using stereotypes, rather than defining new notations. Moreover, our approach is dual to the theoretical approaches that try to formalize the UML. Our notations are also more expressive and abstract than [47] as far as communication issues are concerned. One problem with UML 1.3 is that it does not allow a clear architecture of concurrent systems, there is a need for *concurrency diagram*. Such a kind of concurrency diagram was also suggested in [39, 17, 43].

KORRIGAN and UML-RT [50] partly address the same issues than KORRIGAN : 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 active, reactive and proactive systems.

UML 2.0 tries to extend, clean up and clarify the UML 1.x on several points [12]. We only focus on architecture and scalability issues here. There is a notion of component with ports (required and offered services). A port is a first-class concept to denote instantiable connections. The notion of interface of UML 1.x is expanded to allow the offered and required sides of components. There is also protocol state machines (a restricted UML Statechart) which specify service invocation sequences in interfaces. There is a notion of structured class to allow hierarchical structure. These are valuable, but mainly syntactic, improvements to increase the ability and utility of UML to architecture and scalability. Despite this, UML still remains weak on the consistency aspect of data types and dynamic behaviours. It is surely very difficult to get adequate tools for checking and verifications of UML designs.

5.3 Other Related Work

As previously quoted [40] is one of the main reference about ADL. Another important milestone in the area of architecture and formal specification is [52]. Architectural descriptions are important for software engineers. Their first use is to define idioms or patterns to communicate the overall architecture of a system. The second is that they 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 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.

In [13] Büchi and Sekerinski exhibit the benefits of using 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 express syntactical aspects of functional code. The authors propose the notion of formal contract based on pre and post-conditions, but they criticize their popular form (as it appears in [41]) and since they are only checked at run-time, they do not prevent from errors like formal proofs or static analysis.

One important related work is Distributed Feature Composition (DFC) [60] which is a component based architecture for telecommunications services. It is based on pipe-and-filter architecture. The paper proposes a general model to define telecommunication services based on a network (tree) of boxes. Each box is either transparent or processes some inputs and produces outputs. Their applications consider services as well as routing problems. However this model lacks of a general and formal means to describe protocols like state diagrams. They

use Promela and Z specifications [32], we rather use symbolic transition systems and algebraic specifications in KORRIGAN. We argue that our model is a formal model which is able to design the DFC model.

Rapide, [36], is an event-based concurrent, object-oriented language for prototyping system architectures. A main design principle is to get an executable model in which the concurrency, synchronization, dataflow and timing properties are explicit. The execution model is based on partially ordered event sets. The language ensures scalability and dynamism of architectures. Many features of Rapide have a counterpart in KORRIGAN, however we generally provide an equivalent graphic presentation. Event patterns, event mappings and formal constraints of Rapide are useful concepts to define reference architectures and to test conformance of systems. We rather advocate a more abstract approach oriented to proofs while Rapide promotes simulation and tests but with some static analysis aspects.

Our notations for the glue between communicating components may be also related to [26]. The main differences are that our glue is more expressive than LOTOS synchronizations, and that we have a more structured organization of communication patterns. In [56], Turner discusses about architecture of specifications, not the specification of architectures. It advocates an approach based on fundamental information processing concepts and illustrates it with LOTOS. It shows how architectural semantics can be embodied in a library of specification templates as a practical tool for the specifier. We completely agree with this point of view but we improve the LOTOS way of defining component and architecture with KORRIGAN.

Piccola [1] proposes a powerful language to define architectural style. The core features of Piccola is communicating agents and forms. Agents are calculating units which communicate asynchronously through shared channels. Forms are a powerful abstraction of communicated value, record and arguments; it may be seen as a kind of environment of Lisp and Scheme languages. In fact Piccola offers both lexical and dynamic scoping which was the subject of a long debate in the functional programming community. The concept of form is a powerful abstraction of what we called the glue in component models. The syntax of Piccola resembles that of Python and has many common features with lambda-calculus and Lisp. It introduces the nice idea of *component algebra* and it provides user defined architectural styles. KORRIGAN provides only few architectural styles but with more graphical wiring of components. Piccola is based on π L-calculus: a variant of π-calculus which exchanges forms rather than tuples. Piccola may serve has a basic calculus to found and to extend our KORRIGAN model.

Our approach is more abstract than components at the coding level like: Entreprise Java Beans, Active X, Corba Component and so on. There exist some comparisons of these proposals for instance in [30, 9]. However these proposals are at the coding level and they take into account features not generally covered at the specification level. Hence we currently ignore some important aspects like security, persistence, life cycle, and reflexivity among others.

6 Conclusion

We defined in previous work [14, 15, 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. 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. These features have been applied in this paper on a simple cash-point service case study. KORRIGAN provides clear dynamic interfaces and means to structure complex systems with both full data types and behaviours. 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. Our formal ADL provides explicit first class components but not explicit connectors. However thanks to the powerful glue, connectors may be defined as patterns in KORRIGAN. Our language has been designed such that some dynamic reconfiguration is integrated in the core language. This is the result of clear choices which promote separation of concerns, structuring and a powerful glue with axioms and temporal logic.

We are now working on compatibility rules and verification procedures for our KORRIGAN specifications. Due to the use of STS, *i.e.* Symbolic Transition Systems, such procedures have to be adapted [29, 51, 48]. Another

area of future researches is to reconsider asynchronous communications as the basic communication mode to get a more realistic and distributed model. One way to explore is the use of model based on the π-calculus both to express locality and mobility but also to elaborate advanced verification techniques taking into account dynamicity of the architecture and full data types.

References

[1] Franz Achermann and Oscar Nierstrasz. Applications = Components + Scripts – A Tour of Piccola. In Mehmet Aksit, ed., *Software Architectures and Component Technology*, pages 261–292. Kluwer, 2001.

[2] M. Aiguier, F. Barbier, and P. Poizat. A Logic for Mixed Specifications. Technical report 73-2002, LaMI, 2002. Presented at WADT'2002. Available at `ftp://ftp.lami.univ-evry.fr/pub/specif/poizat/documents/RR-ABP02.ps.gz`.

[3] Michel Allemand and Jean-Claude Royer. Mixed Formal Specification with PVS. In *Proceedings of the 15th IPDPS 2002 Symposium, FMPPTA*. IEEE Computer Society, 2002.

[4] Robert Allen, Remi Douence, and David Garlan. Specifying and Analyzing Dynamic Software Architectures. In *Proceedings of the 1998 Conference on Fundamental Approaches to Software Engineering (FASE'98)*, Lisbon, Portugal, March 1998.

[5] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997.

[6] Christian Attiogbé, Pascal Poizat, and Gwen Salaün. Integration of Formal Datatypes within State Diagrams. In M. Pezze, ed., *Fundamental Approaches to Software Engineering (FASE'2003)*, volume 2621 of *Lecture Notes in Computer Science*, pages 341–355. Springer-Verlag, 2003.

[7] Ferenc Belina, Dieter Hogrefe, and Amardeo Sarma. *SDL with Applications from Protocol Specification*. The BCS Practitioner. Prentice Hall, 1991.

[8] L. Bellissard, S. B. Atallah, A. Kerbrat, and M. Riveill. Component-based programming and application management with Olan. *Lecture Notes in Computer Science*, 1107:290–303, 1996.

[9] M. Blay-Fornarino and Anne-Marie Dery, eds.. *L'objet, coopération dans les systèmes à objets*. Vol 7 number 3, Hermès, 2002.

[10] Grady Booch. *Software Engineering with Ada*. Benjamin/Cummings Series in Ada and Software Engineering. The Benjamin/Cummings Publishing Company, Inc., 2nd edition, 1987.

[11] M. Broy. Specification and top down design of distributed systems. In H. Ehrig, C. Floyd, M. Nivat, and J. Thatcher, eds., *TAPSOFT'85*, volume 185 of *Lecture Notes in Computer Science*, pages 4–28. Springer Verlag, 1985.

[12] Bruce Powel Douglass. UML 2.0: Incremental Improvements for Scalability and Architecture. Technical report, I-Logix Inc., 2003. `http://www.ilogix.com/quick_links/white_papers/index.cfm`.

[13] M. Buechi and E. Sekerinski. Formal methods for component software: The refinement calculus perspective. *Lecture Notes in Computer Science*, 1357:332–337, 1997.

[14] Christine Choppy, Pascal Poizat, and Jean-Claude Royer. From Informal Requirements to COOP: a Concurrent Automata Approach. In J.M. Wing and J. Woodcock and J. Davies, ed., *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. Springer-Verlag, 1999.

[15] Christine Choppy, Pascal Poizat, and Jean-Claude Royer. A Global Semantics for Views. In T. Rus, ed., *International Conference on Algebraic Methodology And Software Technology, AMAST'2000*, volume 1816 of *Lecture Notes in Computer Science*, pages 165–180. Springer Verlag, 2000.

[16] Christine Choppy, Pascal Poizat, and Jean-Claude Royer. Formal Specification of Mixed Components with Korrigan. In *Proceedings of the 8th Asia-Pacific Software Engineering Conference, APSEC'2001*, pages 169–176. IEEE, 2001. ISBN: 0-7695-1408-1.

[17] Christine Choppy, Pascal Poizat, and Jean-Claude Royer. Specification of Mixed Systems in KORRIGAN with the Support of a UML-Inspired Graphical Notation. In Heinrich Hussmann, ed., *Fundamental Approaches to Software Engineering. 4th International Conference, FASE 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6. 2001 Proceedings*, volume 2029 of *LNCS*, pages 124–139. Springer, 2001.

[18] Christine Choppy, Pascal Poizat, and Jean-Claude Royer. The Korrigan Environment. *Journal of Universal Computer Science*, 7(1):19–36, 2001. Special issue: Tools for System Design and Verification, ISSN: 0948-6968.

[19] E. Clarke, O. Grumberg, and D. Long. Verification tools for finite-state concurrent systems. In *A Decade of concurrency – Reflections and Perspectives*, volume 603 of *Lecture Notes in Computer Science*. Springer Verlag, 1994.

[20] Eva Coscia and Gianna Reggio. JTN: A Java-Targeted Graphic Formal Notation for Reactive and Concurrent Systems. In *Fundamental Approaches to Software Engineering (FASE'99)*.

[21] Gerardo Costa and Gianna Reggio. Specification of abstract dynamic-data types: A temporal logic approach. *Theoretical Computer Science*, 173(2):513–554, 1997.

[22] Wolfgang Emmerich. Distributed component technologies and their software engineering implications. In *Proceedings of the 24th Conference on Software Engeneering*, volume 26, pages 537–546. ACM Press, 2002.

[23] Wolfgang Emmerich and Nima Kaveh. F2: Component technologies: Java beans, COM, CORBA, RMI, EJB and the CORBA component model. In Volker Gruhn, ed., *Proceedings of the Joint 8th European Software Engeneering Conference and 9th ACM SIGSOFT Symposium on the Foundation of Software Engeneering (ESEC/FSE-01)*, volume 26, 5 of *SOFTWARE ENGINEERING NOTES*, pages 311–312, New York, September 10–14 2001. ACM Press.

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

[25] Hubert Garavel, Mark Jorgensen, Radu Mateescu, Charles Pecheur, Mihaela Sighireanu, and Bruno Vivien. CADP'97 - Status, Applications, and Perspectives. In *Proceedings of the 2nd COST 247 International Workshop on Applied Formal Methods in System Design*, June 1997.

[26] Hubert Garavel and Mihaela 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.

[27] Object Management Group. CORBA Component Model Specification, v3.0. Technical report, 2002. http://www.omg.org/technology/documents/formal/components.htm.

[28] David Harel. Statecharts: A visual formulation for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

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

[30] Marjan Hericko, Matjaz B. Juric, Ales Zivkovic, Ivan Rozman, Tomaz Domajnko, and Marjan Krisper. Java and distributed object models: An analysis. *ACM SIGPLAN Notices*, 33(12):57–65, December 1998.

[31] ISO/IEC. LOTOS: A Formal Description Technique based on the Temporal Ordering of Observational Behaviour. ISO/IEC 8807, International Organization for Standardization, 1989.

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

[33] K. Nygaard K. and O-J. Dahl. Simula an Algol-based Simulation Language. *CACM*, 9:671–678, September 1966.

[34] Wojtek Kozacynski and Grady Booch. Component-based software engineering. *IEEE Software*, 15(5):34–36, September 1998.

[35] Leslie Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

[36] David C. Luckham, John L. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and analysis of system architecture using rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, April 1995.

[37] C. Shankland M. Calder, S. Maharaj. An Adequate Logic for Full LOTOS. In *Proceedings of the FME'2001 Conference*, Lecture Notes in Computer Science. Springer-Verlag, 2001.

[38] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying distributed software architectures. In *Proceedings of ESEC '95 - 5th European Software Engineering Conference*, pages 137–53, Sitges, Spain, 25–28 September 1995. IEEE.

[39] Michael J. McLaughlin and Alan Moore. Real-time extensions to UML. *Dr. Dobb's Journal of Software Tools*, 23(12):82, 84, 86–93, December 1998.

[40] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, January 2000.

[41] Bertrand Meyer. *Object-Oriented Software Construction, 2nd Ed*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, second edition, 1997.

[42] Oscar Nierstrasz. Regular Types for Active Objects. In *ACM Proceedings OOPSLA*, pages 1–15, 1993.

[43] Liang Peng, Annya Romanczuk, and Jean-Claude Royer. A Translation of UML Components into Formal Specifications. In Theo D'Hondt, ed., *TOOLS East Europe 2002*, pages 60–75. Kluwer Academic Publishers, 2003. ISBN: 1-4020-7428-X.

[44] Pascal Poizat. KORRIGAN*: a Formalism and a Method for the Structured Formal Specification of Mixed Systems*. Doctoral thesis, Institut de Recherche en Informatique de Nantes, Université de Nantes, 2000. in French. `http://www.lami.univ-evry.fr/ poizat/publications-fr.php`.

[45] Pascal Poizat, Christine Choppy, and Jean-Claude Royer. Concurrency and Data Types: a Specification Method. An Example with LOTOS. In J. Fiadero, ed., *Recent Trends in Algebraic Development Techniques, Selected Papers of the 13th Workshop on Algebraic Development Techniques, WADT'98*, volume 1589 of *Lecture Notes in Computer Science*, pages 276–291. Springer-Verlag, 1999.

[46] Pascal Poizat and Jean-Claude Royer. Une proposition de composants formels. In M. Dao et M. Huchard, ed., *Actes de LMO'2002*, pages 231–245, 2002. ISBN 2-6462-0093-7.

[47] Gianna Reggio and Mauro Larosa. A graphic notation for formal specifications of dynamic systems. In John Fitzgerald, Cliff B. Jones, and Peter Lucas, eds., *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods*, volume 1313 of *Lecture Notes in Computer Science*, pages 40–61, Graz, Austria, September 1997. Springer-Verlag. ISBN 3-540-63533-5.

[48] Jean-Claude Royer. Formal Specification and Temporal Proof Techniques for Mixed Systems. In *Proceedings of the 15th IPDPS 2001 Symposium, FMPPTA*, San Francisco, USA, 2001. IEEE Computer Society.

[49] Jean-Claude Royer. The GAT Approach to Specify Mixed Systems. *Informatica*, 2003. to appear, ISSN 0350-5596, `http://ai.ijs.si/informatica/`.

[50] Bran Selic and Jim Rumbaugh. Using UML for Modeling Complex Real-Time Systems. Technical report, Rational Software Corp., 1998.

[51] 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.

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

[53] 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, eds., *Formal Methods Europe (FME'97)*, volume 1313 of *Lecture Notes in Computer Science*, pages 62–81. Springer-Verlag, 1997.

[54] Jean-Louis Sourrouille. A framework for the definition of behavior inheritance. *Journal of Object-Oriented Programming*, 9(1):17–21, 1996.

[55] Springer Verlag, ed.. *The Cash-Point (ATM) Problem*, volume 12 of *Formal Aspects of Computing Science*. 2000.

[56] Kenneth J. Turner. Relating architecture and specification. *Computer Networks and ISDN Systems*, 29(4):437–456, March 1997.

[57] Kenneth J. Turner. Specification architecture illustrated in a communications context. *Computer Networks and ISDN Systems*, 29(4):397–412, March 1997.

[58] UML Consortium. The OMG Unified Modeling Language Specification, Version 1.3. Technical report, June 1999. `ftp://ftp.omg.org/pub/docs/ad/99-06-08.pdf`.

[59] Niklaus Wirth. *Programming in Modula-2*. Springer-Verlag, Berlin, Germany, 1983.

[60] Pamela Zave and Michael Jackson. A component-based approach to telecommunication software. *IEEE Software*, 15(5):70–78, September /October 1998.

LABORATOIRE DE MÉTHODES INFORMATIQUES

KORRIGAN: a Formal ADL with Full Data Types and a Temporal Glue

Pascal Poizat, Jean-Claude Royer

Abstract

The KORRIGAN formalism is devoted to the structured formal specification of mixed systems through a model based on a hierarchy of *views*. In our unifying approach, views are used to describe the different aspects of components (data, behaviours, architecture, communication). While our model is primarily devoted to mixed structured specifications it is a true Architectural Description Language (ADL). As quoted in [40], an ADL must address the component, connector and configuration concepts. In this paper we will demonstrate that KORRIGAN may be used at the formal specification level to describe them. In addition to existing approaches KORRIGAN provides full data types, temporal logic gluing facilities, structuring and readability. We will also discuss related issues such as verifications and tools, and related work.

Additional Key Words and Phrases: Architectural Description Language, Software Component, Mixed Formal Specification, Graphical Notation, Symbolic Transition System, Temporal Glue

LaMI, UMR 8042 CNRS, Genopole & Université d'Évry Val d'Essonne
Tour Évry 2, 523 place des terrasses de l'Agora
F-91000 ÉVRY