

# The KORRIGAN Environment

**Christine Choppy<sup>1</sup>, Pascal Poizat<sup>2</sup>, Jean-Claude Royer<sup>2</sup>**

<sup>1</sup> LIPN, Institut Galilée, Université Paris XIII,  
Avenue Jean-Baptiste Clément, F-93340  
Villetaneuse, France  
Christine.Choppy@lipn.univ-  
paris13.fr

<sup>2</sup> IRIN, Université de Nantes  
2 rue de la Houssinière, B.P. 92208, F-44322 Nantes  
cedex 3, France  
Jean-Claude.Royer@irin.univ-  
nantes.fr

— *Génie Logiciel, Méthodes et Spécifications Formelles* —



**RESEARCH REPORT**

**N<sup>o</sup> 00.10**

**September 2000**

Christine Choppy, Pascal Poizat, Jean-Claude Royer

*This work was presented at FMTOOLS'2000 in Reisenburg, Germany, July 2000.*

17 p.

Les rapports de recherche de l'Institut de Recherche en Informatique de Nantes sont disponibles aux formats PostScript® et PDF® à l'URL :

<http://www.sciences.univ-nantes.fr/irin/Vie/RR/>

*Research reports from the Institut de Recherche en Informatique de Nantes are available in PostScript® and PDF® formats through the URL:*

<http://www.sciences.univ-nantes.fr/irin/Vie/RR/indexGB.html>

© September 2000 by Christine Choppy, Pascal Poizat, Jean-Claude Royer

# The KORRIGAN Environment

Christine Choppy, Pascal Poizat, Jean-Claude Royer

## Abstract

This paper presents an environment to support the use of specification for mixed systems, *i.e.* systems with both dynamic (behaviour) and static (data type) aspects. We provide an open and extensible environment based on the KORRIGAN specification model. This model uses a hierarchy of view concepts to specify data types, behaviours and compositions in a uniform way. The key notion behind a view is the symbolic transition system. A good environment supporting such a model needs to interface with existing languages and tools like JAVA, LOTOS, LP or PVS. The core of our environment is the CLIS library which is devoted to the representation of our view concepts and existing specification languages. This environment is implemented with the object-oriented language PYTHON. Actually, it provides an integration process for new tools, a specification library, a parser library, LOTOS generation and object-oriented code generation for KORRIGAN specification.

Categories and Subject Descriptors: D.2 [**Software Engineering**]: Design Tools and Technique

General Terms: Environment, Specification, Tools

Additional Key Words and Phrases: Formal Specification, Software Environment, Specification Library, Mixed System, View, Symbolic Transition System, Dynamic Behaviour, Data Type.



# 1 Introduction

In this paper, we present an environment to support the use of specification for mixed systems, *i.e.* systems with both a dynamic aspect (behaviour, communication, concurrency) and a static aspect (data type). While the importance of mixed formal specifications is widely accepted, there is still a need for open and extensible tools and environments. “Open” meaning here that it should be possible to link them with other existing tools. Another need is to integrate the formal specification into a software process (*e.g.* an object-oriented one). Therefore, in our environment, we need editing and formatting tools, verification means, as well as prototyping and code generation tools. Moreover, object-orientation is often advocated for programming, and we would like to extend this to specifications and specification developments.

To specify real-life size and complex systems, one needs to use several languages dedicated to the different system parts. In several methods (for instance UML [10]), we have to describe functional aspects, dynamic behaviours and static parts. The main problem is how to glue all of these descriptions and to get a consistent and global semantics. The KORRIGAN model is an attempt to overcome this problem. Our model of mixed systems is based on the notion of views [9]. This model aims at keeping advantage of the languages dedicated to both aspects (algebraic specifications for data types, and state-transitions diagrams for dynamic behaviour) while providing a unifying model with an operational semantics.

We want to provide a specification language with a global semantics and also with guidelines for the specifiers. Such a language must be equipped with various tools: parsing, editing, and formatting. We also want to integrate our approach into a software development process and to provide prototyping, code generation and verification tools. This is of course an ambitious and a long term task.

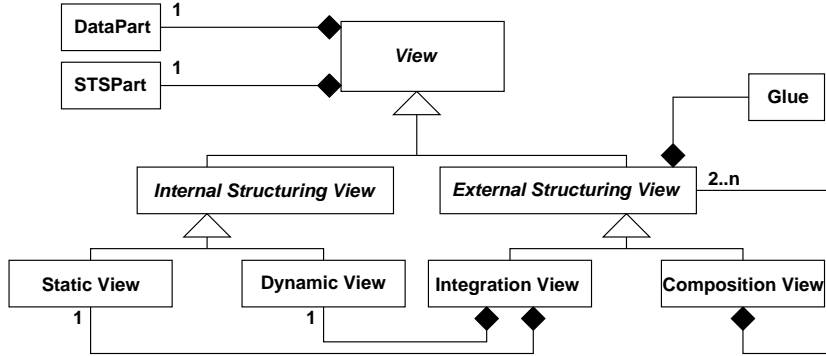
In this paper we start with a presentation of the KORRIGAN specification model and its notion of view. The second Section is devoted to the description of our environment: goals, principles, and architecture. The third Section describes the libraries: a set of components for the KORRIGAN specifications but also for other specification languages. Section four presents different tools which have developed and integrated in our environment.

## 2 The KORRIGAN Specification Model

Our model focuses on the specification of systems with both static and dynamic aspects and that feature a certain level of complexity that requires the definition of structuring mechanisms. We use two ways to ensure structuring and modularity: a simple form of inheritance and the composition of specification components.

Our model is based on the notion of *view*, an interface to describe components. A view [Fig. 1] has a static part and a state-transition or behavioural part. The key concept behind this notion is the *Symbolic Transition System* (STS) concept. STSs [14] are a general form of finite state-transition diagrams which provides an appropriate level of abstraction and avoids state explosion by the use of guards and open (*i.e.* not ground) terms in states and transitions. STSs are more powerful and more readable than classic state-transition diagrams. However the difficult counterpart remains about verifications. The dynamic aspects of components are described in *dynamic views* (cf. [Fig. 1]). The static aspects are described using *static views*. The integration of all aspects of a given component is done using *integration views*. Finally the (concurrent) compositional aspects of components are described by *composition views*. Both integration and composition views use a mixed “glue” (algebraic first-order axioms and temporal formulas) to express the interface of composition as a whole. A great part of the semantics of the model is devoted to explain how to compute a global view structure for the different compositions [9].

We illustrate the different views using a simple password manager of the Unix system. As an example of a static view, see [Fig. 2], we have the data type which memorizes information about the users (its abstracts the `/etc/passwd` file). Basically a static view describes a data type, it is an algebraic specification with a STS point of view. The graphical description of the STS appears in [Fig. 3]. [Fig. 4] is an example of the STS part of a KORRIGAN dynamic view. It describes the activities and the communications of the password manager. There are some syntactic features to denote emissions and receptions of (may be) complex typed data. It has states and transitions but they do not necessarily represent simple and finite entities (rather equivalence classes). This is due to guards and variables which may appear on transitions and inside states. For example the transition from the `BeC` state to the `cGp2` state means: if some validity condition on the received user identifier is satisfied, then we may

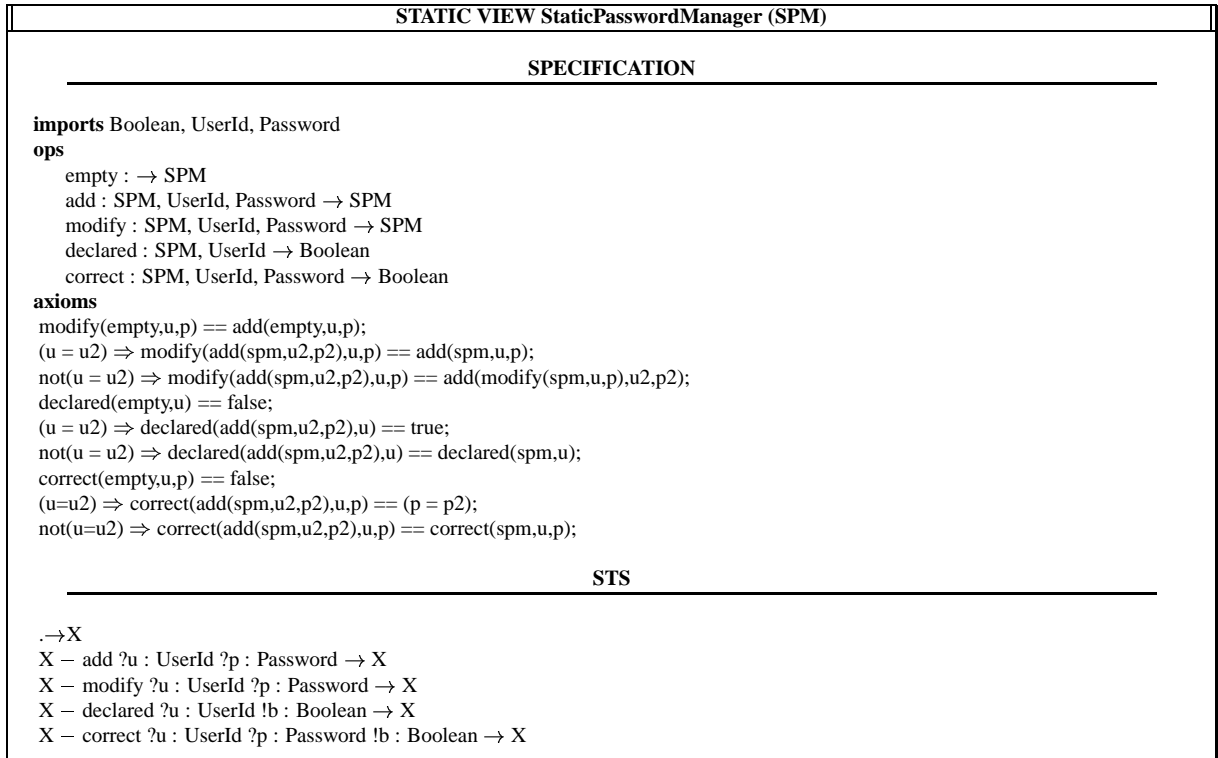
Figure 1: *the view model class diagram*

trigger this transition and we get a password from the last process the component received a message from (the root process). The `cValid` guard is an abstract guard, it means that the user is declared. This has to be mapped out of a particular static view. Then we must link this guard with an operation of this data type: this is done by the concept of integration view.

To describe views we use KORRIGAN textual descriptions. The textual description for the integration view of our password manager example is depicted in [Fig. 5]. One can see the two components: the static view and the dynamic view. They are glued with four sets of formulas. The first one expresses the correspondence of predicates between the two views (static and dynamic). For example the `cValid` guard corresponds to the negation of the declared operation in the static view. The second and the third sets are used to synchronize respectively states and transitions of the two STSs. Here we synchronize the `ic` transition of the dynamic part with the `add` transition of the static part. The last set defines the initial state of the component as a restriction of the free composition of the two subcomponents initial states. An operational semantics exists and was presented in [9]. This semantics is based on the extraction of a global STS and a global data part specification from a view. From the textual description of [Fig. 5], we may build the global STS view of [Fig. 6]. This global STS, was obtained by computing a general form of the synchronous product of the two STS components. It represents the global activity of the two components of the password manager: its static and its dynamic part. Within this global STS, the transition from the `cVal` state to the `Wa` state means: if the state of the static part satisfies `true` and if the two passwords (in the dynamic part) are equal, then the static part triggers its `add` transition and the dynamic part triggers its `ic` transition. The [Fig. 6] illustrates that in our model we have complex state transition diagrams with compound transitions and states. This complexity comes from the use of state and transition formulas and also from the product of STSs. This also illustrates the need of graphical and textual presentations of the same entity. This fact is now widely accepted, for example in the SDL [4] or UML [10] languages. Composition is achieved in a similar way, by gluing several views with the same glue than in an integration view. By lack of space we do not describe an example of a composition view, see [23] for examples.

### 3 The KORRIGAN Environment

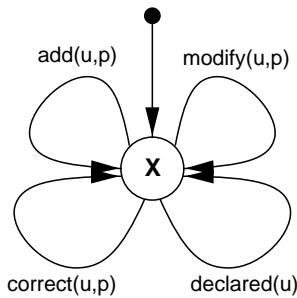
The current environment architecture is organized as in [Fig. 7]. This environment is currently still under development. We have implemented some concepts and tools, but we do not have yet a graphical user interface. We also provide a library (CLIS) of interface formats, for example to target the Larch Prover tool [12], LOTOS [5] or Xfig. CLIS is a set of classes to define our concepts like views and STSs but also interface classes. We have developed our proper simple package for conditional rewriting. In the future, if we need more efficient rewriting we will interface the KORRIGAN environment with an external rewriting system (ELAN [6] for example). We have also defined our proper classes to describe various kind of algebraic terms (terms, equations, offers, guards, and conditional axioms). The parsing package is a set of parsers to read descriptions from files and to generate the corresponding class instances. We may produce various formats for documentation and editing tools. For example

Figure 2: *the PasswordManager static view*

we have defined generation of Xfig as a part of the CLAP Library. We also expect to target some verification tools and some object-oriented languages.

### 3.1 Design Principles

The design of our environment follows several principles. The first principle is to interface with some existing tools and environments, for example model checking tools (*e.g.* XTL in CADP [11]), theorem provers (*e.g.* the Larch Prover [12], KIV [27], ELAN [6], PVS [22], HOL-CASL [17]), and programming languages (*e.g.* JAVA [13], C++ [28]). 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 library (detailed below) 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

Figure 3: *the STS of the PasswordManager static view*

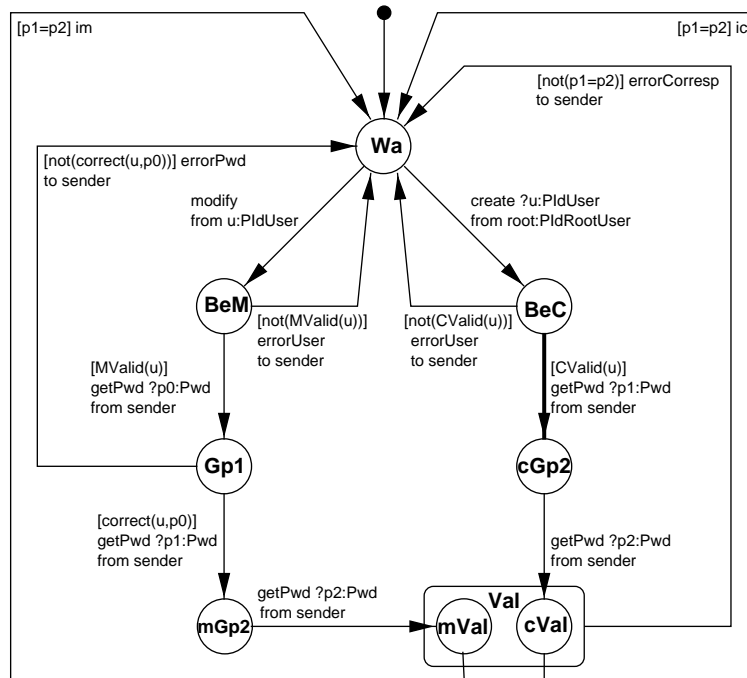


Figure 4: the STS of the PasswordManager dynamic view

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

Figure 5: the PasswordManager integration view

both in the design and in the implementation of our environment. We have chosen object-oriented programming because one cannot ignore the quality of such code, and also because we had already a good experience of this programming model in a formal context [1].

### 3.2 The PYTHON Language

The implementation is done in PYTHON [16]. The PYTHON language is an interpreted object-oriented language, hence it is really useful to produce both quick scripts and prototypes of complex environments. One important feature is that PYTHON is free, open source and portable across several platforms (Unix, Linux, Windows ...). It is closely related to Lisp, Perl, and C++, but it is much more legible. It is dynamically type-checked, functional and object-oriented. It has a simple meta-object protocol and provides exceptions, powerful built-in data structures and module libraries (parser generation, CORBA programming, and XML parsing).

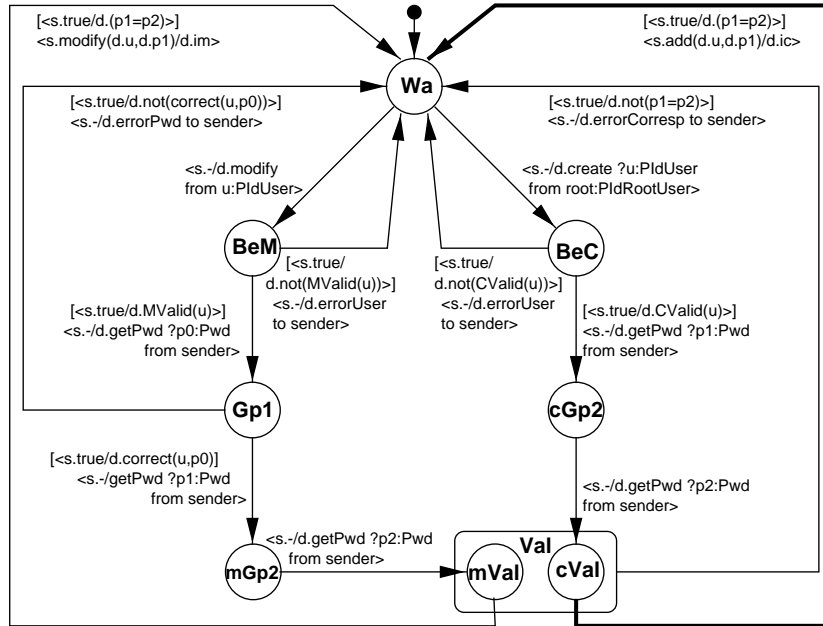


Figure 6: the global STS corresponding to the integration view

### 3.3 Integration Process

Based on this object-oriented framework, we have defined a general process to integrate new languages and new tools in our environment [Fig. 8]. From the abstract syntax description of a given specification we get an Abstract Syntax Tree (AST) instance using the parsing mechanism (see [Section 5.1]). From that an interpreter builds an instance of a class in the CLIS hierarchy. Some transformations may therefore be done on this instance to get another instance of the CLIS Library. For example one may want to transform the data part of a static view into a CASL [21] or a Larch Prover specification. This is done by defining a method from the source class to the target class. Finally there is a print method in each specification class which is able to print out the required specification format. Once parsing and printing for a language are implemented, the main task of the designer is to implement methods to convert one CLIS class to another CLIS class. This process was used for example with the generation of Xfig documentation for CLAP instances. It also has been used to generate LOTOS specifications (see [Section 5.3]). Note also that this process can be done in another language (C for example) and interface PYTHON with it (using SWIG -*Simplified Wrapper and Interface Generator*- [3]).

## 4 The CLIS Library

CLIS (*Class Library for Specification*) is an extensible hierarchy mapping the specification classification. It contains classes for the KORRIGAN model, but also for other formalisms. We provide a general hierarchy for specifications, with subclasses corresponding to data types or dynamic specifications. We try to classify the different approaches, but this is a difficult task and this hierarchy is only for design support. As an example of data type specification we have the Larch Prover tool language and as an example of dynamic specification we have members of the CLAP library (see [Section 4.1]). Our KORRIGAN model is a subclass of mixed specification languages, LOTOS is another example of mixed specification language. In the future, we expect to integrate in this hierarchy some other specification languages (SDL [4], CASL [18] and CASL-LTL [26]).

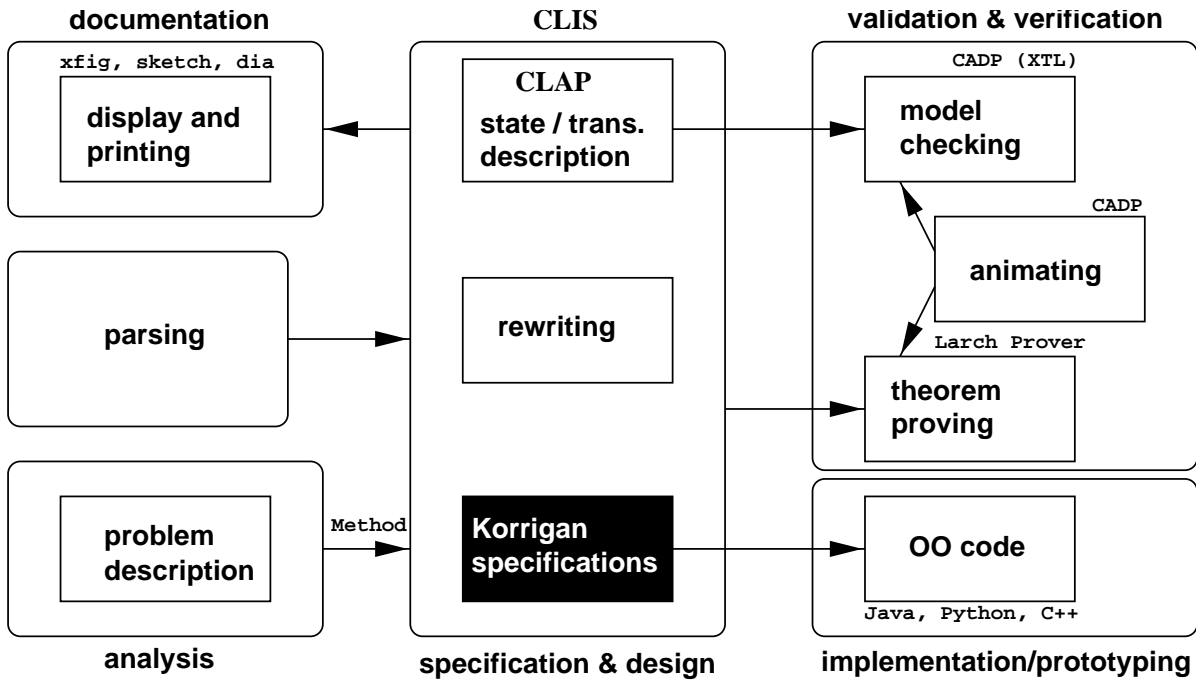


Figure 7: the KORRIGAN architecture

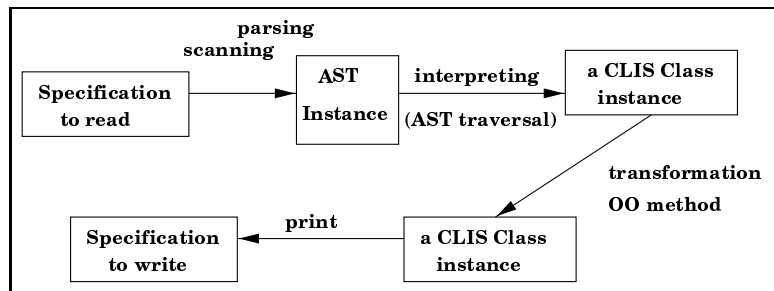


Figure 8: the integration process

### 4.1 The CLAP Hierarchy

State transitions diagrams are important in KORRIGAN, and more generally to represent models of the dynamic systems, thus a special part of CLIS is devoted to them. CLAP (*Class Library for Automata in Python* [20]) enables one to define different kinds of state-transition diagrams by providing an extensible hierarchy of classes. For example, there are classes for automata with state or transition parameters (initial states, labels, emissions, receipts, colours), and Petri Nets. It is easy to add a new class corresponding to some new kind of state-transition diagram by subclassing. The state-transition diagrams are stored in files following a generic internal format. A parser for this format is provided, see [Section 5.1]. The diagrams may be automatically transformed to displaying formats (Xfig or DOT) following the [Fig. 8] schema.

The original part concerns symbolic transition systems, because there do not exist packages devoted to this kind of systems. We do not have a very efficient implementation for STSs, but we think this is not the prominent problem with that concept for the moment. STSs are abstract, hence they have generally a small size. Therefore classical graph algorithms are sufficiently efficient.

A part of the current CLAP hierarchy is presented in [Fig. 10]. Different kinds of symbolic transitions systems are represented by the `GATDiagram` and the `STSKorriganDiagram` classes. The `GATDiagram` class

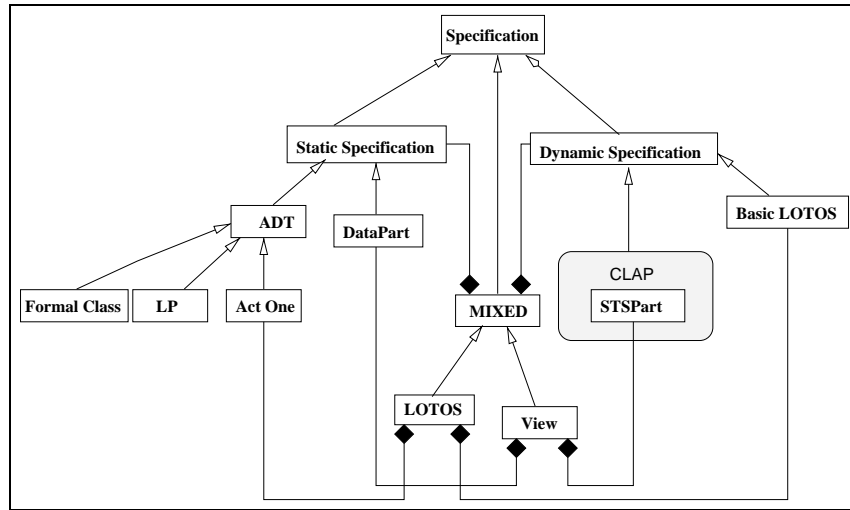


Figure 9: the CLIS hierarchy (part of)

describes general STSs without any offer for emission and reception, it may be used to describe static views. The `STSKorriganDiagram` class is the one used in the KORRIGAN model to define dynamic views.

## 5 KORRIGAN Environment Tools

We have implemented some tools in our current environment. The focus is rather put on generality and rapid prototyping, efficiency is not a main goal for the moment. Note that all these mechanisms are implemented by classes and they are easy to extend. Other tools have been experimented in different context (Smalltalk, CLOS) and will be integrated soon.

### 5.1 Parsing

We choose to reuse a simple and general approach based on SPARK [2]. It defines a package with four levels devoted to scanning, parsing, semantic analysis and code generation. Each of these levels provides a general class and some methods. To define a proper system (*i.e.* devoted to a given language), one may subclass the general classes and redefine some methods. For example to produce a scanning for a new language we subclass the `GenericScanning` class. We also redefine the methods which declare a regular expression and the associated action. The main advantage of SPARK is to use object-oriented programming which provides extensibility and reusability of components. [Fig. 12] presents the part of actual class diagram of the parsing package related to the view model parsing. There are three parallel hierarchies in this package, one for parsing, another one for scanning, and a last one for interpreters. Interpreters are used to transform an AST instance into an instance of a CLIS class. These CLIS class instances may then serve as internal formats.

### 5.2 Automata Related Operations

Since our semantics uses the synchronous product of diagrams we need to implement it. We may note that the product of two simple STSs is no longer a simple STS, as we saw with the STS of the integration view of the password manager [Fig. 6]. This is the reason why the notion of structured STSs appears in our hierarchies. This also requires to define structured identifiers, states or transitions. Together with the hierarchy, we have some basic functionalities. CLAP allows one to define temporal formulas in order to compute initial states, states and transitions reachable from the initial states, and parameterized synchronous products. The most interesting is the synchronous product of diagrams. It allows one to build the synchronous product of any number of state transition

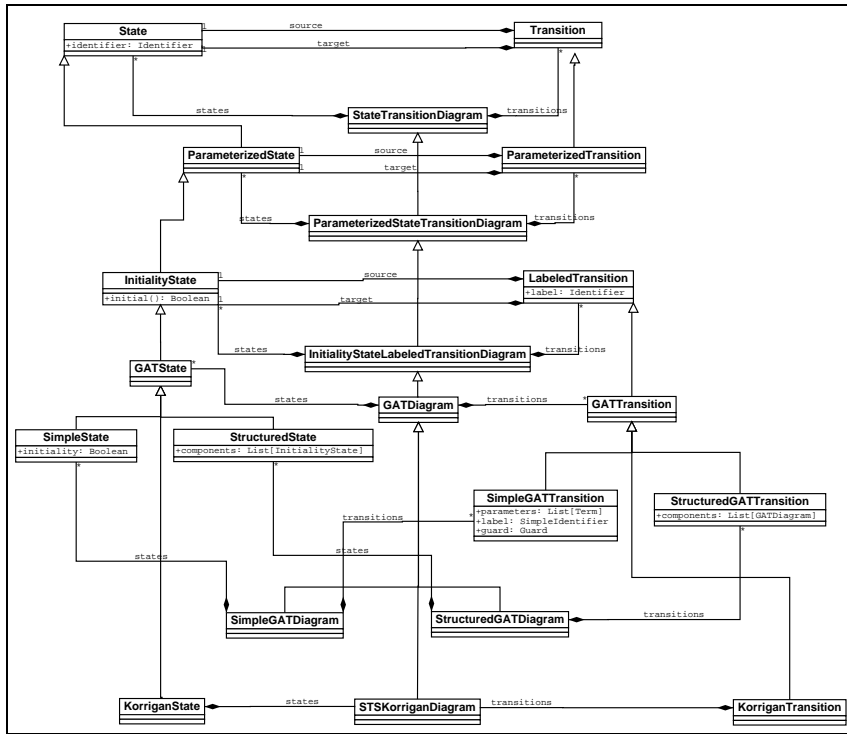


Figure 10: the CLAP hierarchy (part of)

diagrams, choosing the list of synchronization, the synchronization mode (and the resulting type). For example we can simulate either LOTOS or CCS synchronization rules with the same operation (but with different parameters of course).

### 5.3 LOTOS Generation

We have defined translation mechanisms to generate LOTOS or SDL specification from a KORRIGAN specification [24, 8]. The principle is depicted in [Fig. 13] and follows the schema described in [Fig. 8]. These mechanisms use patterns to generate the dynamic behaviours. The translation of the static part is straightforward. We start from a file containing a KORRIGAN view description. The parsing mechanism produces an instance of the view class with a static part and a dynamic part. The static part is transformed into an instance of the ACT ONE class. The dynamic part is transformed into a Basic LOTOS instance. Both transformations are done according to the patterns defined

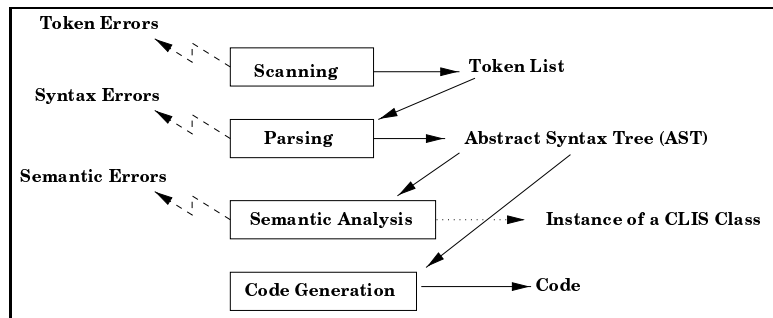
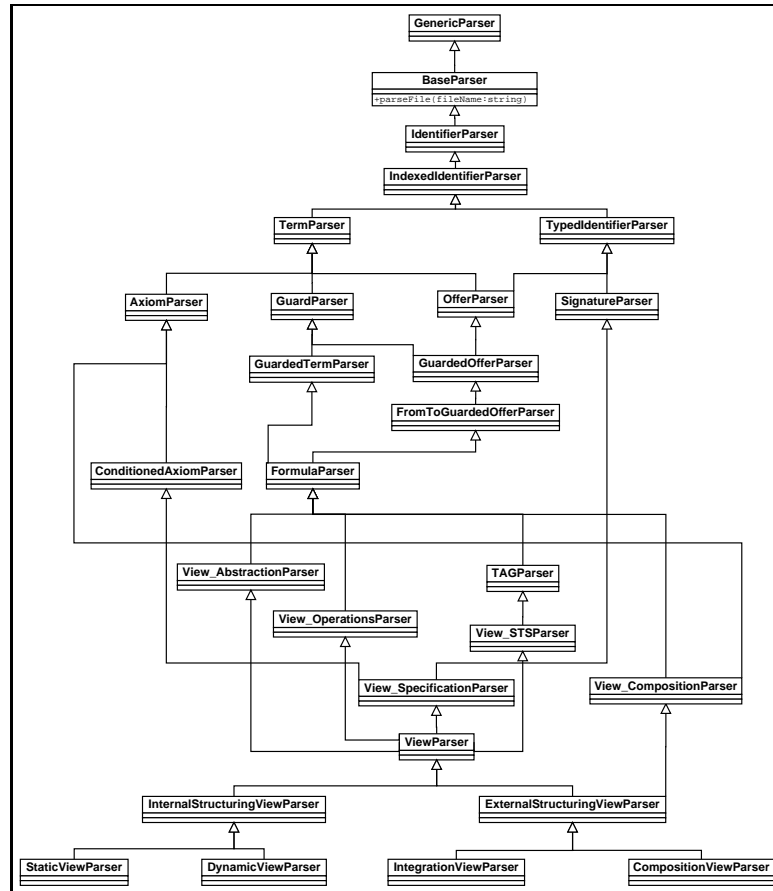


Figure 11: SPARK parsing principles

Figure 12: *the parsing hierarchy (part of)*

in [8]. These two instances permit to build a full LOTOS class instance. This class has a print method which is then used to write LOTOS code. This code generation was extended to SDL [8] with the support of a specification method.

## 5.4 Object-Oriented Code Generation

The object-oriented code generation, see [Fig. 14], is achieved with the following steps (see [8] for more details). From a KORRIGAN specification, as with LOTOS, we consider two main transformations, one for the static part and one for the dynamic part. The dynamic part, a symbolic transition system with communications and structuration, is implemented by controller structures. These structures are then translated into a concurrent object-oriented language (Active JAVA [19]). When the algebraic part is not executable it is refined into an executable one (through interactions with the user). Algebraic specifications are translated into an intermediate object-oriented code based on Formal Classes [1], and implemented into pure JAVA. This intermediate level is a formal and object-oriented model which allows us to simplify and to abstract the object-oriented generation of the static part. Finally the JAVA class implementing the static part is encapsulated in an Active JAVA implementing the dynamic part. This was experimented without using the [Fig. 8] principles. To be able to follow them, we need to reify object-oriented languages by classes in the CLIS hierarchy.

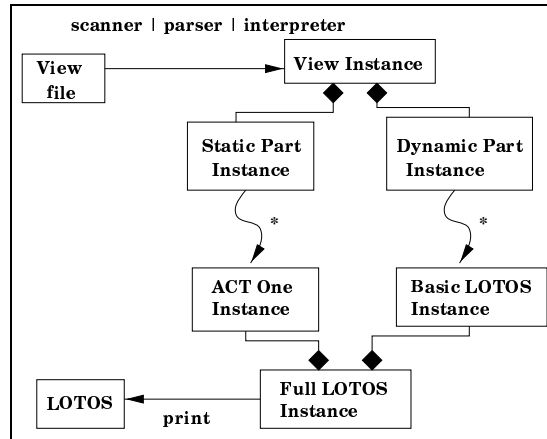


Figure 13: *the view-to-LOTOS translation principle*

## 5.5 G-Derivation

Another tool will be associated with the CLAP Library: the G-derivation tool. Usually one writes algebraic axioms in a constructive way over the constructors of the data type. G-derivation adapts this idea with a constructor term generation based on a STS [24, 7]. This is useful with views because the method may extract an algebraic specification compatible with the STS. The main points of the method are:

- the signature is automatically extracted from the STS,
- a graph traversal helps to choose the constructors of the data type,
- an algorithm, based on the STS graph traversal, builds a great part of the axioms,
- the specifier has only to give the right-hand side conclusion terms.

For the moment, this tool is implemented within the Smalltalk environment. This generation will be extended to structured STSs.

## 5.6 Verification

It is possible to generate inputs to verification tools using the [Fig. 8] translation mechanism. For example we can use the CADP [11] environment to verify LOTOS specifications resulting from the [Section 5.3] translation. Another idea is to translate the KORRIGAN specification into a specific formalism where both static and dynamic parts are expressed using the algebraic framework (*e.g.* CASL-LTL [26]), and then verify this resulting specification using a theorem prover. Moreover we plan to define and implement specific verification tools. Mixed specifications require specific symbolic verification means [25, 15]. Current symbolic model-checkers does not seem well-suited to our KORRIGAN STSs. Thus, in the future we will propose some proof mechanisms more adequate to our STSs.

## 6 Conclusion

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, Xfig, ... We also have a generic tool to describe state-transition diagrams, to build their (a)synchronous composition, and to compute their graphical representation. Object-oriented source code may be generated from KORRIGAN specifications.

The KORRIGAN environment is based on a classification of specifications with a general parsing mechanism. New formalisms may be integrated, and translation mechanisms for them may be defined. It was not too difficult

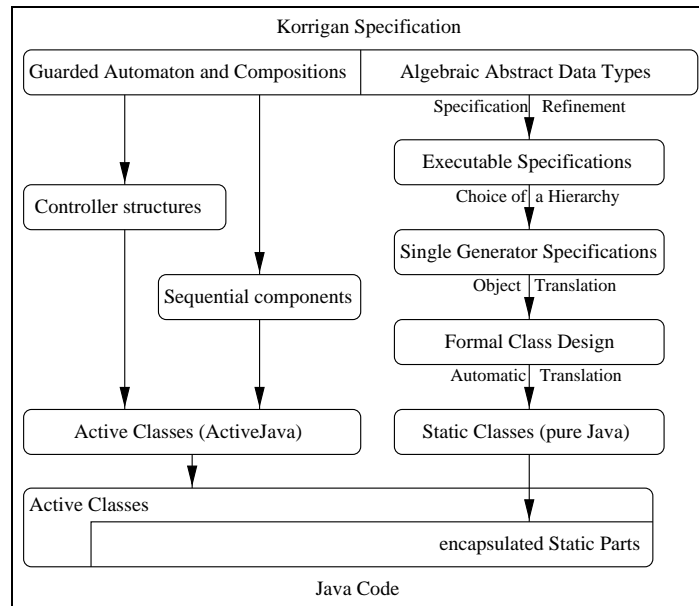


Figure 14: the view-to-JAVA code generation

to prototype such an application with PYTHON. We have already a general hierarchy for LP, LOTOS, KORRIGAN views and symbolic transition systems. We have also a general process to extend our environment. Tools have been implemented: parsers, Xfig documentation and the synchronous product of STSs. Other tools have been experimented in Smalltalk or in CLOS, they are being integrated in the current environment.

We do not have yet a graphical user interface, we will begin its design soon. It will integrate a method for mixed system we have developed [8] and the different tools we have experimented. Another issue deals with model-checking and verifications. We have done some small experimentations, but until now we have nothing really useful on complex systems. As a first proposal, we interface our environment with existing formalisms and verification tools. Then we will propose proof mechanisms more adequate to our model.

## References

- [1] Pascal ANDRÉ, Dan CHIOREAN, et Jean-Claude ROYER. The Formal Class Model. In *Joint Modular Languages Conference, Modula, Oberon & friends*, ISBN 3-89559-220-X, pages 59–78, Ulm, Germany, 1994.
- [2] John AYCOCK. Compiling Little Languages in Python. In *7th International Python Conference*, Houston, Texas, November 1998. <http://www.foretec.com/python/workshops/1998-11/proceedings.html>.
- [3] David M. BEAZLEY. SWIG : An Easy to Use Tool For Integrating Scripting Languages with C and C++. In *4th Annual USENIX Tcl/Tk Workshop*, pages 287–294, Monterey, California, 1996.
- [4] Ferenc BELINA, Dieter HOGREFE, et Amardeo SARMA. *SDL with Applications from Protocol Specification*. The BCS Practitioner. Prentice Hall, 1991.
- [5] Tommaso BOLOGNESI et Ed. BRINKSMA. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–29, 1988.
- [6] P. BOROVANSKY, C. KIRCHNER, H. KIRCHNER, P.E. MOREAU, et C. RINGEISSEN. An Overview of ELAN. In C. et H. KIRCHNER, éditeurs, *2nd International Workshop on*

- Rewriting Logic and its Applications, WRLA'98*, volume 15. Elsevier Science B. V., 1998. <http://www.elsevier.nl/locate/entcs/volume15.html>.
- [7] Christine CHOPPY, Pascal POIZAT, et Jean-Claude ROYER. Control and Data Types using a Multiview Formalism. In *14th Workshop on Algebraic Development Techniques*, Bonas, France, September 1999.
- [8] Christine CHOPPY, Pascal POIZAT, et Jean-Claude ROYER. From Informal Requirements to COOP: a Concurrent Automata Approach. In J.M. WING AND J. WOODCOCK AND J. DAVIES, éditeur, *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.
- [9] Christine CHOPPY, Pascal POIZAT, et Jean-Claude ROYER. A Global Semantics for Views. In T. RUS, éditeur, *International Conference on Algebraic Methodology And Software Technology, AMAST'2000*, volume 1816 of *Lecture Notes in Computer Science*, pages 165–180. Springer Verlag, 2000.
- [10] UML CONSORTIUM. The OMG Unified Modeling Language Specification, Version 1.3. Rapport technique, June 1999. <ftp://ftp.omg.org/pub/docs/ad/99-06-08.pdf>.
- [11] Jean-Claude FERNANDEZ, Hubert GARAVEL, Alain KERBRAT, Radu MATEESCU, Laurent MOUNIER, et Mihaela SIGHIREANU. CADP: A Protocol Validation and Verification Toolbox. In *8th Conference on Computer-Aided Verification*, pages 437–440, 1996.
- [12] Stephan GARLAND et John GUTTAG. An overview of LP, the Larch Prover. In *Proc. of the 3rd International Conference on Rewriting Techniques and Applications*, volume 355 of *Lecture Notes in Computer Science*. Springer-Verlag, 1989.
- [13] GOSLING, JOY, et STEELE. *The Java Language Specification*. Addison Wesley, 1996.
- [14] M. HENNESSY et H. LIN. Symbolic Bisimulations. *Theoretical Computer Science*, 138(2):353–389, 1995.
- [15] Carron KIRKWOOD et Muffy THOMAS. Towards a Symbolic Modal Logic for LOTOS. In *Northern Formal Methods Workshop NFM'96*, eWIC, 1997.
- [16] Mark LUTZ. *Programming Python*. O'Reilly & Associates, 1996.
- [17] Till MOSSAKOWSKI, KOLYANG, et Bernd KRIEG-BRÜCKNER. Static semantic analysis and theorem proving for Casl. In *12th Workshop on Algebraic Development Techniques*, volume 1376 of *Lecture Notes in Computer Science*, pages 333–348. Springer Verlag, 1997.
- [18] P. D. MOSSES. CoFI: The Common Framework Initiative for Algebraic Specification and Development. In M. BIDOIT et M. DAUCHET, éditeurs, *Proc. TAPSOFT'97*, number 1214 in *Lecture Notes in Computer Science*, Berlin, 1997. Springer Verlag.
- [19] Juan M. MURILLO, Juan HERNÁNDEZ, Fernando SÁNCHEZ, et Luis A. ÁLVAREZ. Coordinated Roles: Promoting Re-usability of Coordinated Active Objects Using Event Notification Protocols. In Paolo CIANCARINI et Alexander L. WOLF, éditeurs, *3rd International Conference, COORDINATION'99*, volume 1594 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.
- [20] Gaël NEDELEC et Gwen SALAÜN. CLAP: a Class Library for Automata in Python. Master's thesis, Université de Nantes, 1999. <http://www.sciences.univ-nantes.fr/info/recherche/mgl/FRANCAIS/ThemesetProjets/SHES/CLAP/>.
- [21] The CoFI Task Group on LANGUAGE DESIGN. CASL The Common Algebraic Specification Language Summary. Version 1.0. Rapport technique, 1999. <http://www.bricks.dk/Projects/CoFI/Documents/CASL/Summary/>.

- 
- [22] S. OWRE, S. RAJAN, J.M. RUSHBY, N. SHANKAR, et M.K. SRIVAS. PVS: Combining specification, proof checking, and model checking. In Rajeev ALUR et Thomas A. HENZINGER, éditeurs, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414. Springer-Verlag, 1996.
- [23] Pascal POIZAT. *Korrigan : un formalisme et une méthode pour la spécification formelle et structurée de systèmes mixtes*. Thèse de Doctorat, Université de Nantes, November 2000.
- [24] Pascal POIZAT, Christine CHOPPY, et Jean-Claude ROYER. Concurrency and Data Types: a Specification Method. An Example with LOTOS. In J. FIADERO, éditeur, *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.
- [25] J. RATHKE et M. HENNESSY. Local Model Checking for Value-Passing Processes (Extended Abstract). In Martín ABADI et Takayasu ITO, éditeurs, *3rd International Symposium on Theoretical Aspects of Computer Software TACS'97*, volume 1281 of *Lecture Notes in Computer Science*, pages 250–266. Springer Verlag, 1997.
- [26] G. REGGIO, E. ASTESIANO, et C. CHOPPY. CASL-LTL: A CASL Extension for Dynamic Reactive Systems – Summary. Rapport technique DISI-TR-99-34, DISI – Università di Genova, Italy, 1999. <ftp://ftp.disi.unige.it/person/ReggioG/ReggioEtAl199a.ps>.
- [27] Wolfgang RIEF. The KIV-approach to Software Verification. In M. BROU et S. JÄHNICHEN, éditeurs, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software - Final Report*, volume 1009 of *Lecture Notes in Computer Science*. Springer Verlag, 1995.
- [28] Bjarne STROUSTRUP. *The C++ Programming Language*. Addison-Wesley, 1987.





# The KORRIGAN Environment

Christine Choppy, Pascal Poizat, Jean-Claude Royer

## Abstract

This paper presents an environment to support the use of specification for mixed systems, *i.e.* systems with both dynamic (behaviour) and static (data type) aspects. We provide an open and extensible environment based on the KORRIGAN specification model. This model uses a hierarchy of view concepts to specify data types, behaviours and compositions in a uniform way. The key notion behind a view is the symbolic transition system. A good environment supporting such a model needs to interface with existing languages and tools like JAVA, LOTOS, LP or PVS. The core of our environment is the CLIS library which is devoted to the representation of our view concepts and existing specification languages. This environment is implemented with the object-oriented language PYTHON. Actually, it provides an integration process for new tools, a specification library, a parser library, LOTOS generation and object-oriented code generation for KORRIGAN specification.

Categories and Subject Descriptors: D.2 [**Software Engineering**]: Design Tools and Technique

General Terms: Environment, Specification, Tools

Additional Key Words and Phrases: Formal Specification, Software Environment, Specification Library, Mixed System, View, Symbolic Transition System, Dynamic Behaviour, Data Type.