Journal of Universal Computer Science, vol. 7, no. 1 (2001), 19-36
submitted: 1/9/00, accepted: 13/10/00, appeared: 28/1/01 © Springer Pub. Co.

The Korrigan Environment

Christine Choppy
(LIPN, Université Paris XIII, France
Christine.Choppy@lipn.univ-paris13.fr)

Pascal Poizat
(IRIN, Université de Nantes, France
Pascal.Poizat@irin.univ-nantes.fr)

Jean-Claude Royer
(IRIN, Université de Nantes, France
Jean-Claude.Royer@irin.univ-nantes.fr)

Abstract: This paper presents an environment to support the use of specification for mixed systems, *i.e.* systems with both dynamic (behaviour, communication, concurrency) 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. At the core of our environment is the CLIS library which is devoted to the representation of our view concepts and existing specification languages. Our environment is implemented using the object-oriented language PYTHON. It provides an integration process for new tools, a specification library, a parser library, LOTOS generation and object-oriented code generation for KORRIGAN specifications.
Keywords: Computer-Aided Software Engineering, Software Libraries
Category: D.2 Software Engineering, D.2.2 Design Tools and Technique

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 both in academic (LOTOS, $\mu\mathcal{SZ}$) and industrial worlds (SDL, UML), there is still a need for open and extensible tools and environments to support them. "Open" meaning here that it should be possible to link these tools and environments with other existing ones. 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 formating 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 language (for instance the UML [Con99]), we have to describe functional aspects, dynamic behaviours and static parts. The main problem is to glue all of these descriptions and get a consistent and global semantics. The KORRIGAN model is an attempt to overcome this problem. Our model of mixed systems is based on our notion of views [CPR00a]. 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 should be supported by various tools: parsing, editing, and pretty printing. 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 long term task.

In this paper we start with a presentation of the KORRIGAN specification model and its notion of view. Section 3 is devoted to the description of our environment: goals, principles, and architecture. Section 4 describes the libraries: a set of components for the KORRIGAN specifications but also for other specification languages. Section 5 presents different tools which have been developed and integrated in our environment. Section 6 deals with related work, models and environments. Section 7 describes future work.

2 The Korrigan Specification Model

Our model focuses on the specification of systems with both static and dynamic aspects and 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.

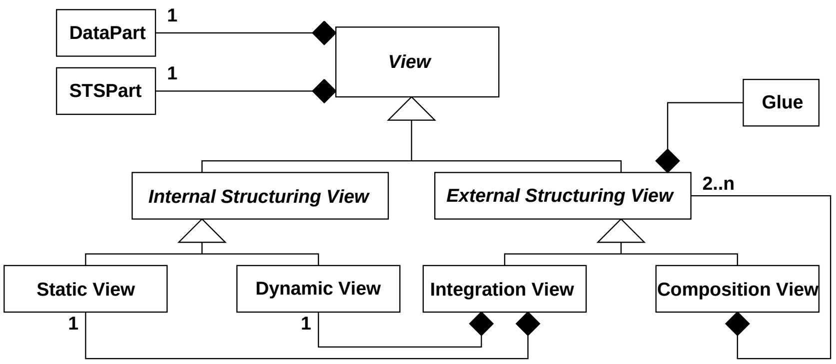

Figure 1: *the view model class diagram*

Our model is based on the notion of *view*, an interface to describe components. A view [Fig. 1] has a static part (`DataPart`) and a state-transition or behavioural part (`STSPart`). The key concept behind this notion is the *Symbolic Transition System* (STS) concept. STSs [HL95] are a general form of finite state-transition diagrams which provide an appropriate level of abstraction and avoid 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 is about verifications. The dynamic aspects of components are described in *dynamic views*. 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 [CPR00a].

STATIC VIEW StaticPasswordManager (SPM)

SPECIFICATION

imports Boolean, UserId, Password
ops
empty : $\rightarrow$ SPM
add : SPM, UserId, Password $\rightarrow$ SPM
modify : SPM, UserId, Password $\rightarrow$ SPM
declared : SPM, UserId $\rightarrow$ Boolean
correct : SPM, UserId, Password $\rightarrow$ Boolean
axioms
modify(empty,u,p) == add(empty,u,p);
(u = u2) $\Rightarrow$ modify(add(spm,u2,p2),u,p) == add(spm,u,p);
not(u = u2) $\Rightarrow$ modify(add(spm,u2,p2),u,p) == add(modify(spm,u,p),u2,p2);
...

STS

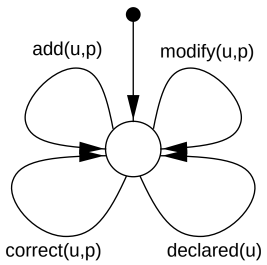

Figure 2: *the* `PasswordManager` *static view*

We illustrate the different views using a simple Unix-like password manager case study. As an example of a static view, see [Fig. 2], we have the data type which memorizes information about the users (it 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 the bottom of the [Fig. 2].

An example of the STS part of a KORRIGAN dynamic view is depicted in [Fig. 3].

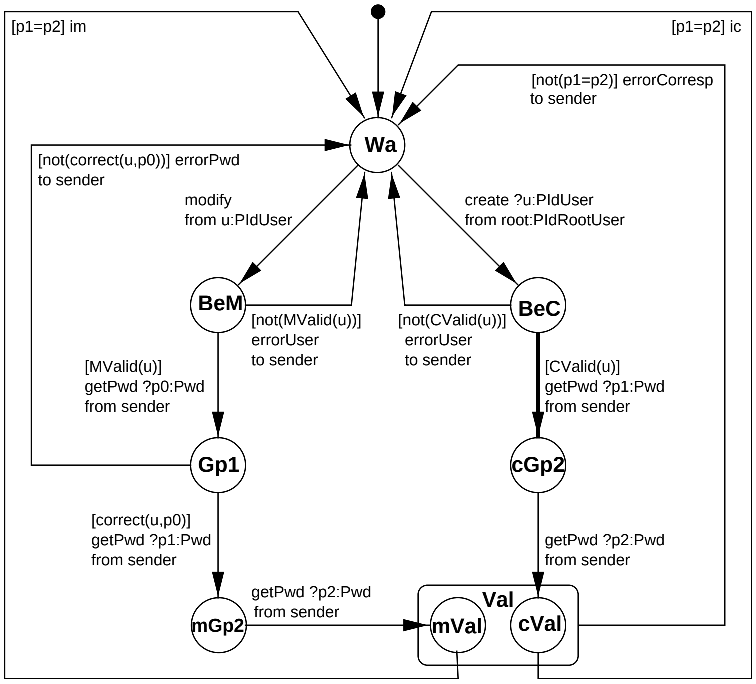

Figure 3: *the STS of the* `PasswordManager` *dynamic view*

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 which represent equivalent classes rather than simple and finite entities. 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 trigger this transition and get a password from the last process the component received a message from (the `root` process). The `CValid` guard is an abstract guard, checking that the user is declared. Such an abstract guard has to be mapped out of a particular data type. Therefore we must link it with an operation of the static view. 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. 4] where the static view (`STATIC`) and the dynamic (`DYNAMIC`) views are glued with four sets of formulas. The first set (`axioms`) expresses the correspondence of predicates between the static and dynamic views. For example

INTEGRATION VIEW Password Manager	
COMPOSITION ALONE	

is
STATIC s: SPM
DYNAMIC d: DPM
axioms
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)
	with true,
	{
	(d.([true] ic), s.(add(d.u,d.p1))),
	(d.([true] im), s.(modify(d.u,d.p1)))
	}
	initially true

Figure 4: *the `PasswordManager` integration view*

the `Cvalid` guard corresponds to the negation of the `declared` operation of the static view. The second and the third sets (the `with` clause has two arguments) 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 `ALONE` keyword means that any non explicitly synchronized transition has to happen alone. This is the LOTOS policy but we also provide two other concurrency modes: `KEEP` and `LOOSE`. The last set is a single formula which defines the initial state of the component as a restriction of the free composition of the two subcomponents initial states (here, `initially` true, means no restriction). The operational semantics [CPR00a] is based on the extraction of a global STS and a global data part specification from a composed view. From the [Fig. 4] textual description, we may build the [Fig. 5] global STS view.

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 password manager subcomponents (its static and dynamic parts). 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. 5] 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 in the "glue", and also from the product of STSs. This also illustrates the need for equivalent graphical and textual presentations of the same component. This fact is now widely accepted, for example in the SDL [BHS91] or UML [Con99] languages. Concurrent composition is achieved in a similar way, by gluing several views in the same way as in an integration view. Due to a lack of space, we do not describe here an example of a composition view (see [Poi00] for examples).

3 The Korrigan Environment

The current environment architecture is organized as in [Fig. 6], some parts are implemented and some other parts are currently still under development. CLIS (*Class LIbrary for Specification*) is a class library to support our concepts like views and STSs but it also provides interface formats, for example to target the Larch Prover tool [GG89], LOTOS [BB88] or Xfig. We developed our proper

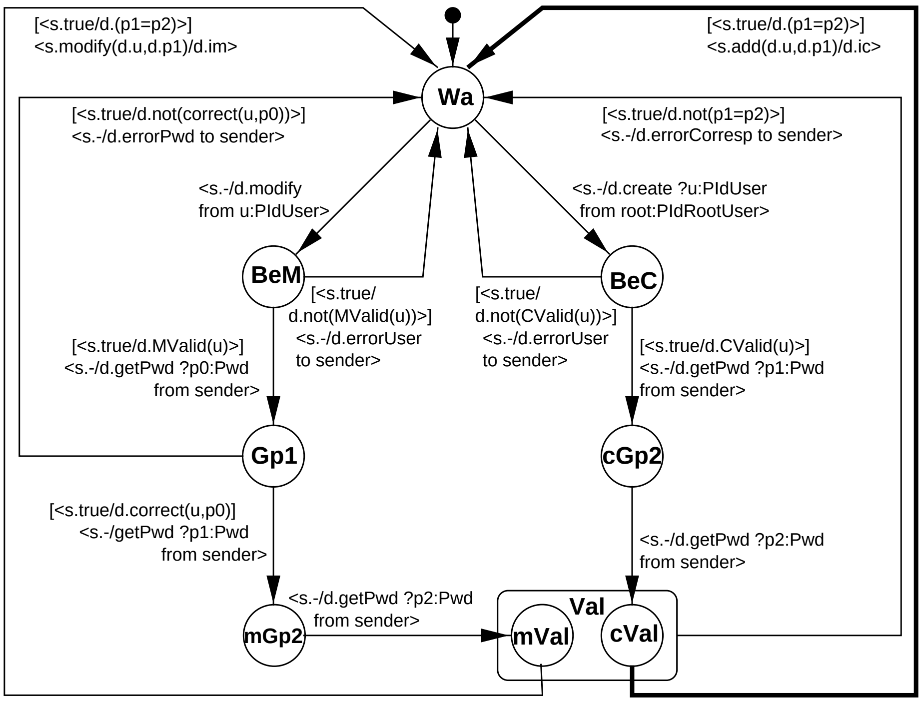

Figure 5: *the global STS corresponding to the integration view*

simple package for conditional rewriting. In the future, more efficient rewriting can be obtained by interfacing the KORRIGAN environment with an external rewriting system (ELAN [BKK+98] for example). We defined our proper classes to describe algebraic terms, offers, guards, 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 we defined the generation of Xfig files as a part of the CLAP Library (*Class Library for Automata in Python*). We also target verification tools (the Larch Prover [GG89], PVS [ORR+96]) and some object-oriented languages (Java [GJS96], C++ [Str87]). Starting from the problem description a method guides the KORRIGAN specification development [Poi00].

3.1 Design Principles

The design of our environment follows several principles. 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 [GJM+97]), theorem provers (*e.g.* the Larch Prover, KIV [Rei95], ELAN [BKK+98], PVS, HOL-CASL [MKB97]), 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

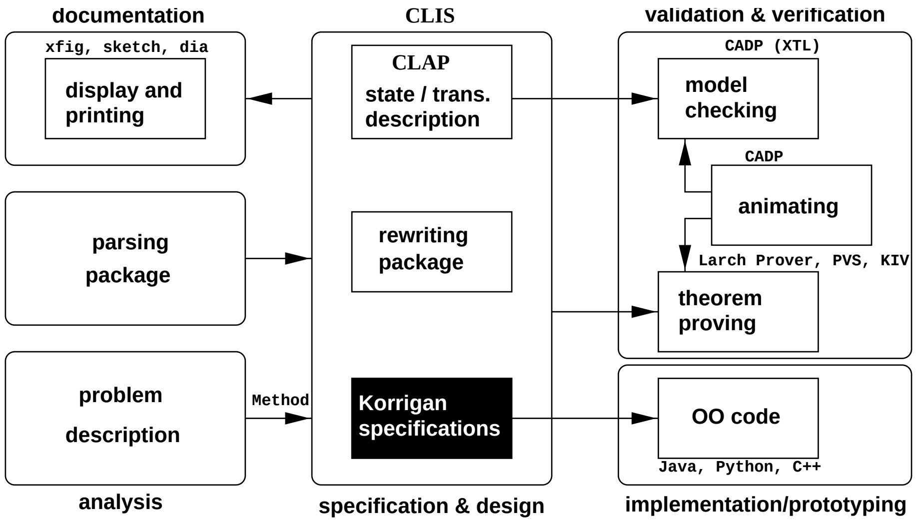

Figure 6: *the* KORRIGAN *architecture*

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 both in the design and in the implementation of our environment. We have chosen object-oriented programming because of its reusability and extensibility, and also because we had already a good experience of this programming model in a formal context [Roy01].

3.2 The Python Language

The implementation is done in PYTHON [Lut96]. 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 Smalltalk, 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).

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. 7]. 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,

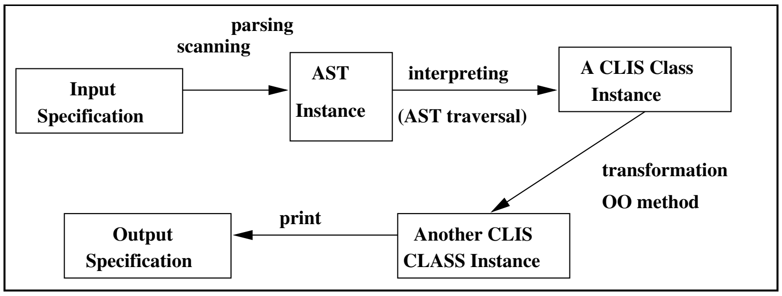

Figure 7: *the integration process*

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 a CLIS Library class. For example, one may want to transform the data part of a KORRIGAN static view into a CASL [CASL99] 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 define 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]) from KORRIGAN specifications. Let us note that while our concern here is to provide the tools, such transformations are semantically valid only for parts of the languages. Note also that this process can be implemented in other languages (C for example) and interface PYTHON with it.

4 The CLIS Library (*Class LIbrary for Specification*)

CLIS [Fig. 8] 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.

4.1 The CLAP Hierarchy (*Class Library for Automata in Python*)

State transitions diagrams are important in KORRIGAN, and more generally to represent models of the dynamic and concurrent systems [SNW93]. Thus, a special part of CLIS is devoted to them. CLAP enables one to define different kinds of state-transition diagrams by providing an extensible hierarchy of classes.

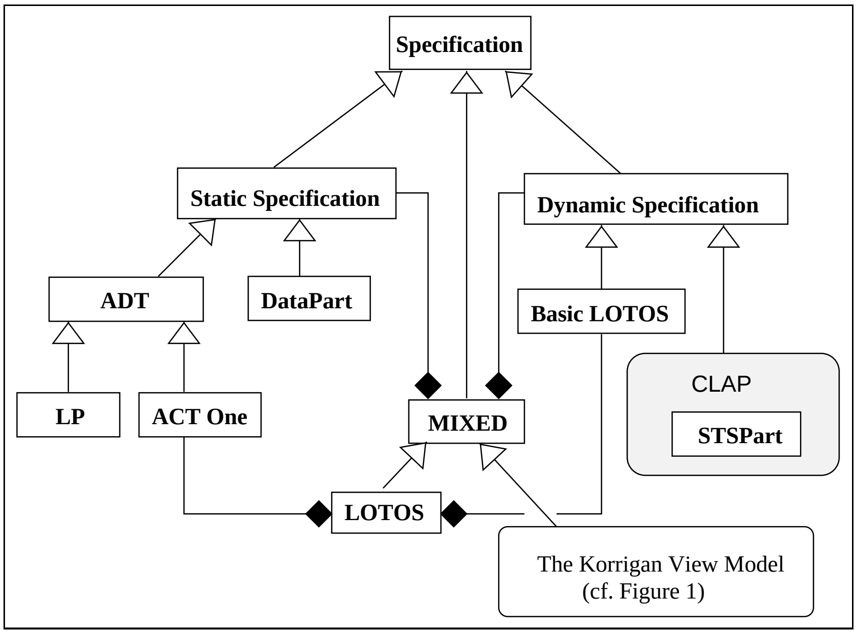

Figure 8: *the* CLIS *hierarchy (part of)*

For example, there are classes for automata with state or transition parameters (initial states, labels, emissions, receptions, colours), and Petri Nets. It is easy to add a new class corresponding to some new kind of state-transition diagram by subclassing. This has been done for KORRIGAN STSs. 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 into displaying and documentation formats following the [Fig. 7] schema.

The original part of CLAP concerns symbolic transition systems, since we take them into account which is not the case in other existing state and transition packages such as BCG in [GJM+97]. We do not have a very efficient implementation for STSs yet. STSs are abstract, hence they have generally a small size. Therefore, classical graph algorithms or model-checking usual tools are efficient enough.

5 Korrigan Environment Tools

On top of these class libraries, we have implemented tools in our environment. The focus is rather put on generality and rapid prototyping, efficiency is not our main goal at the moment. Note that all these mechanisms are implemented by classes and they can easily be extended.

5.1 Parsing

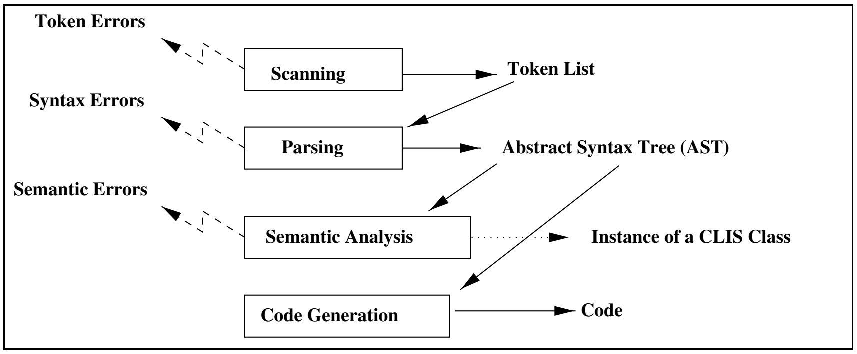

Figure 9: *SPARK parsing principles*

We choose to reuse a simple and general approach based on SPARK [Ayc98]. 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 or some other existing scanner. We may also redefine some of the methods which declare regular expressions and the associated actions. The main advantage of SPARK is to use object-oriented programming which provides extensibility and reusability of components. 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 (see [CPR00a]) 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 for the STS of the integration view of the password manager [Fig. 5]. This is the reason why the notion of structured STS is required in our hierarchies. This also requires to define structured identifiers, states or transitions. CLAP allows one to use 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 generic synchronous product of state and transition systems. It allows one to build the synchronous product of any number of state transition diagrams, choosing the list of synchronizations, 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 from a KORRIGAN specification [PCR99]. Note that our KORRIGAN model may be viewed as a strict superset of LOTOS. There are two main differences: the use of partial functions (while LOTOS uses total functions) and the glue for communications and concurrency which is more general than LOTOS synchronization. Restricting our model to total functions, `ALONE` concurrency and LOTOS compatible offers, a KORRIGAN specification has a direct LOTOS interpretation. If we do not consider such restrictions a more complex translation is possible but requires controllers as in [CPR99].

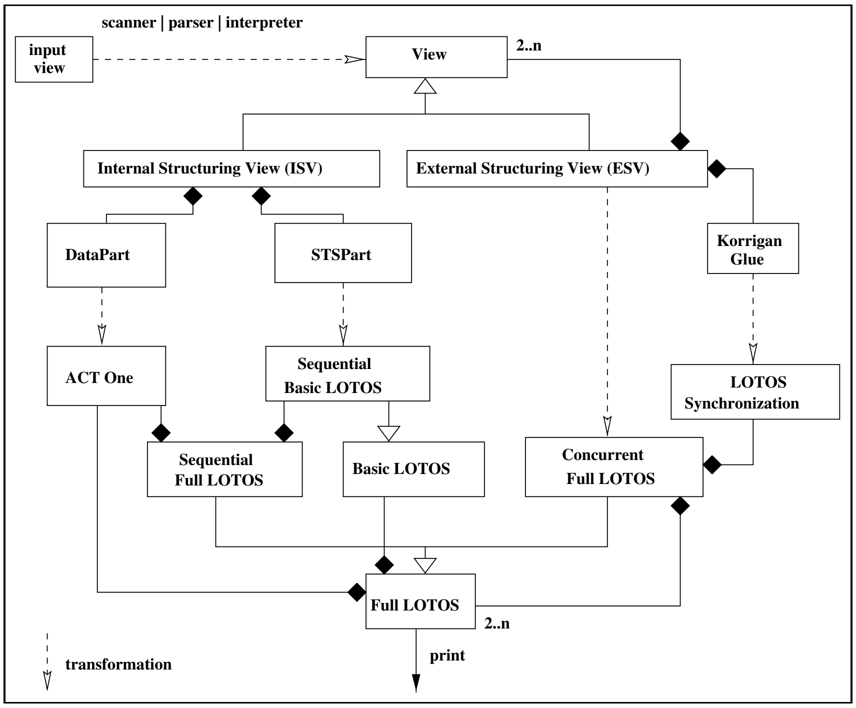

Figure 10: *the view-to-*LOTOS *translation principle*

Here, we restrict our translation to the case where KORRIGAN matches LOTOS. The principle is depicted in [Fig. 10] and follows the general schema we presented in [Fig. 7]. We start from a file containing a KORRIGAN view description. The parsing mechanism produces an instance of the view class. Depending on the class of this view instance (either ISV or ESV) a different transformation is done. An internal structuring view (static view or dynamic view) may be considered as a view with both a data part and a STS part. The translation of the static part proceeds as described in [PCR99] and gives a data type (an ACT ONE specification). We encapsulate this data type into the LOTOS process associated with the STS. The retrieval of the LOTOS behaviour from the STS can be done using different patterns and builds a sequential basic LOTOS instance. The way used here is to associate a process to each STS and for each state a conditional branch is created. Some simplifications are achieved (grouping branches for the same operation and simplifying guards) to get a more readable behaviour expression. Each component of an external structuring view is translated (recursively) into a full LOTOS instance and the glue part gives a synchronization expression. The LOTOS class has a print method which is then used to write LOTOS code.

5.4 Object-Oriented Code Generation

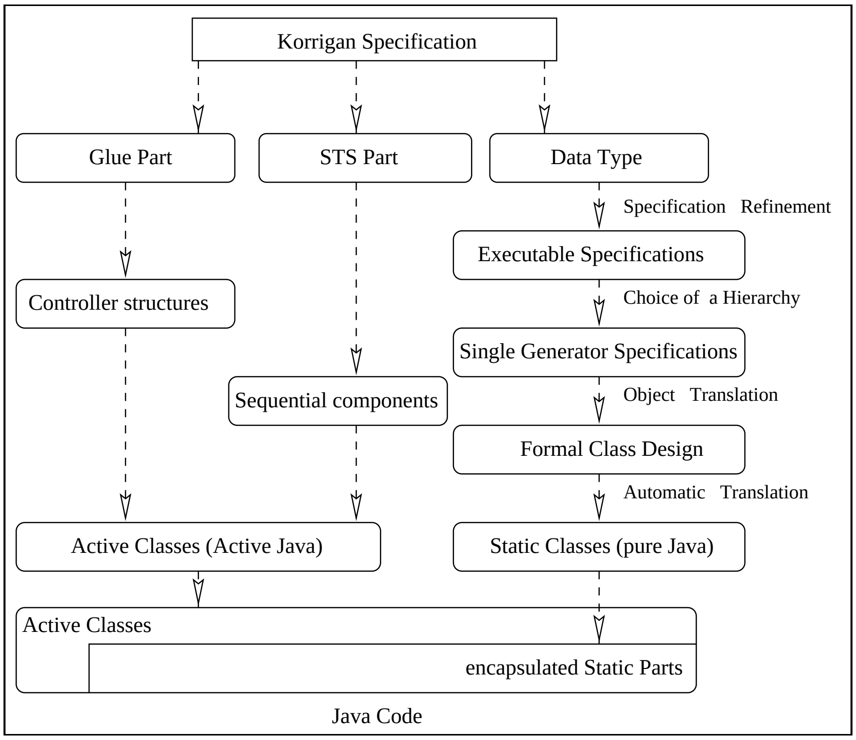

Figure 11: *the view-to-*JAVA *code generation principle*

The object-oriented code generation, see [Fig. 11], is achieved with the following steps (see [CPR99] for more details). Following an approach similar to LOTOS generation in [Section 5.3], two main transformations are performed on the KORRIGAN specification, one for internal structuring views and one for external structuring views. These transformations are based on more elementary transformations: one for data parts, one for STS parts and a last one for glue and behaviour structure. The glue and the behaviour structure, are implemented by controller structures. These structures are then translated into a concurrent object-oriented language (Active JAVA [MHSA99]). The STS part is implemented into an Active JAVA class. A class implements a set of methods which represent the transitions of the STS. The STS states and the guards define the operation preconditions which are implemented as "activation conditions" in Active JAVA. Condition variables may be set (postconditions) in the "post actions" methods of Active JAVA. In the class constructor, initial values for the conditions are set according to the automaton initial state. To be implemented the data part needs to be executable. Whenever 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 [Roy01], 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 into an Active JAVA class implementing the dynamic part.

6 Related Work

Our work is related to different specification languages but also to several software environments, see [Poi00] for a more exhaustive survey.

6.1 Model Comparisons

Our view model has some connections with the UML, see [CPR00b] for more details. Our model is supported by a UML-inspired graphical notation. One important and first difference is that we consider formal specifications which is not the case of the UML. Contrary to the UML, we consider that graphical notations are useful but not sufficient, therefore our method is applied to develop both textual and graphical specifications. We suggest, when possible, to reuse the UML notation, but also to present some proper extensions. Since our model is component-based, we have an approach which is rather different from the UML on communications and concurrent aspects. Thus we have specific notations to define dynamic interfaces of components, communication patterns and concurrency.

Clearly our model may also be related to LOTOS and SDL. LOTOS has a stricter policy for compatible offers (gate names must be equal). KORRIGAN expresses separately the glue between the data type and the dynamic behaviour. With LOTOS, the links to the data types are embedded in the description process. LOTOS allows pure interleaving, in addition KORRIGAN provides two other concurrency modes. Finally KORRIGAN does not restrict data types to total ones. There is the same limitation with data types in SDL as with LOTOS. A great difference between SDL and KORRIGAN is the communication mode which is asynchronous for SDL and synchronous for KORRIGAN. An implementation of asynchronous communications is possible in KORRIGAN *via* buffers. We can also note that the links to the data type are embedded in the dynamic behaviour. KORRIGAN and SDL have a good advantage over LOTOS, they provide an associated graphical notation.

Our model is also related to CASL-Chart [RR00], which is an extension of CASL to reactive systems. One great advantage of CASL-Chart is the possible use of existing tools like Statemate [HLN+90]. The main drawback is that the Statecharts [Har88] are a complex formalism which carry difficulties to modelize, to prove and to animate. Our STSs may be related to Statecharts ([Har88], or the UML ones) but for some differences. Our STSs are simpler (but less expressive) than Statecharts. They model sequential components, concurrency is done through external structuring and the computation of a structured STS from subcomponents STSs [CPR00a]. These STSs are built using conditions which enable one to semi-automatically derive them from requirements. Last our STSs may be seen as a graphical representation of an abstract interpretation for an algebraic data type [AR99].

$\mu\mathcal{SZ}$ [BGGK97] is also a formalism that uses Statecharts but with a combination of Z [Lig91] for data types and a temporal logic for safety properties. This is a modular formalism with features for communications, concurrency and time. It also allows one to animate specification using Statemate. A method has been defined in [GHD98].

6.2 Related Software Environments

The qualities of the graphic user interfaces and the number of tools of our environment are not competitive with industrial products like Objecteering[1], Rational Rose[2] or Telelogic Tau[3]. However, these environments focus on a specification language (except Telelogic Tau which supports both the UML and SDL). Our environment is rather devoted to interface with different environments and to combine several kinds of formal specification.

UML modellers, such as Rose or Objecteering, focus on graphical editing for the UML, object-oriented code generation and metrics. We are interested in verifications and also in object-oriented code generation. We also want to define tools reusable in other environments. UML Modellers are rather closed and do not interface easily with other formal tools.

Telelogic Tau (the SDL suite was previously named SDT) is a set of tools for the formal specification with SDL. Moreover these tools integrate: requirement analysis, system specification with the UML, message sequence charts and a formal design and verification with SDL. It may use SDL as the implementation language (to achieve tests) but also a SDL-to-C code generator. This environment includes converters (UML to SDL, SDL to Statecharts) and integrates the UML and SDL within a single tool. It provides a complete software development solution with up-to-date and industrial standards. Our environment is free and more open, and we have links with the UML but we consider that the current informal UML semantics does not allow a safe translation into a formal language.

CADP [GJM+97] is a set of tools devoted to the specification and the verification of protocols and distributed systems. It comprises specific tools for equivalence and bisimulation testing, model checking, test and animation (via code generation). It also enables one to apply different reductions on graphs. It

[1] Softeam, `http://www.softeam.fr`.
[2] Rational, `http://www.rational.com`.
[3] Telelogic, `http://www.telelogic.com`.

defines three different formats to integrate or link new tools: LOTOS specifications, nets of finite state machines and LTS (BCG format). These LTS may use complex data but they have to be reduced to finite sets of values/terms, they are not symbolic. The BCG format may be translated into a dozen of other state and transition formats, it can be edited and drawn. This environment also focuses on mixed systems but in a less expressive, less abstract and less readable manner. It provides nice graphic user interfaces integrating a complete set of tools. There is no real support for non finite state and transition diagrams and for complex data types. It is not defined to be used on other specifications (*e.g.* LP, OO languages, or documentation languages like Xfig).

7 Future Developments

Parts of the environment are still under development. Students are currently designing the graphic interfaces with the help of the Glade generator [Lut96]. Some tools like the two ones described below have been experimented in different contexts and will be integrated soon within our PYTHON software. In the future, we expect to integrate in the CLIS hierarchy some other specification languages (SDL [BHS91], CASL [Mos97] and CASL-LTL [RAC99]). The object-oriented code generation was experimented without using the [Fig. 7] principles. To be able to follow them, we need to reify the target object-oriented languages by classes in the CLIS hierarchy. The LOTOS code generation mechanism was extended to generate SDL in [CPR99] with the support of a specification method and it makes sense to enrich the current environment with these features.

7.1 G-Derivation

The G-derivation tool is a tool associated with the CLAP Library. 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 [AR99]. This is useful with views because the method may extract an algebraic specification compatible with the STS. The first point of the method is to automatically extract the signature from the STS. A graph traversal then helps the user to choose the constructors of the data type, and an algorithm, based on the STS graph traversal, builds a great part of the axioms, and finally the specifier has only to give the right-hand side conclusion terms. This tool was implemented in Smalltalk.

7.2 Verification

It is possible to generate inputs to verification tools using the [Fig. 7] translation mechanism. For example we can use the CADP [GJM+97] environment to verify LOTOS specifications resulting from the [Section 5.3] translation. Mixed specifications require specific symbolic verification means [RH97, KT97]. Current symbolic model-checkers does not seem well-suited to our KORRIGAN STSs. We plan to define and implement specific verification tools. An 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

[RAC99]), and then verify this resulting specification using a theorem prover. Thus, in the future we will propose some proof mechanisms more adequate to our STSs. An idea is developed in our research team, it is devoted to specifications involving both logical time and physical time with the use of PVS [All00].

8 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 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. The following tools were 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 are now designing the graphical user interface, it will integrate a method for mixed system we have developed [CPR99] and the different tools we have experimented. Another issue is model-checking and verification. One idea is to interface our environment with existing formalisms and verification tools (as done with LOTOS). Another idea is to implement specific proof mechanisms that are more specifically relevant for our model [All00].

References

[All00] Michel Allemand. Verification of properties involving logical and physical timing features. In *Génie Logiciel & Ingénierie de Systèmes & leurs Applications, ICSSEA'2000*, December 2000.

[AR99] Pascal André and Jean-Claude Royer. An Algebraic Approach to the Specification of Heterogeneous Software Systems. In *14th Workshop on Algebraic Development Techniques*, Bonas, France, September 1999.

[Ayc98] John Aycock. Compiling Little Languages in Python. In *7th International Python Conference*, Houston, Texas, November 1998. `www.foretec.com/python/workshops/1998-11/proceedings.html`.

[BB88] Tommaso Bolognesi and Ed. Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–29, 1988.

[BGGK97] Robert Büssow, Robert Geisler, Wolfgang Grieskamp, and Marcus Klar. The μSZ Notation Version 1.0. Technical Report 97–26, TU Berlin, FB 13, 1997.

[BHS91] Ferenc Belina, Dieter Hogrefe, and Amardeo Sarma. *SDL with Applications from Protocol Specification.* The BCS Practitioner. Prentice Hall, 1991.

[BKK+98] P. Borovansky, C. Kirchner, H. Kirchner, P.E. Moreau, and C. Ringeissen. An Overview of ELAN. In In C. and H. Kirchner, editors, *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`.

[CASL99] The CoFI Task Group on Language Design. CASL The Common Algebraic Specification Language Summary. Version 1.0. Technical report, 1999. `http://www.bricks.dk/Projects/CoFI/Documents/CASL/Summary/`.

[Con99] 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`.

[CPR99] 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, editor, *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.

[CPR00a] Christine Choppy, Pascal Poizat, and Jean-Claude Royer. A Global Semantics for Views. In T. Rus, editor, *International Conference on Algebraic Methodology And Software Technology, AMAST'2000*, volume 1816 of *Lecture Notes in Computer Science*, pages 165–180. Springer Verlag, 2000.

[CPR00b] Christine Choppy, Pascal Poizat, and Jean-Claude Royer. Specification of Mixed Systems in KORRIGAN with the Support of an UML-Inspired Graphical Notation. Technical report, IRIN, November 2000. submitted to FASE'2001.

[GG89] Stephan Garland and 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.

[GHD98] Wolfgang Grieskamp, Maritta Heisel, and Heiko Dörr. Specifying Embedded Systems with Statecharts and Z: An Agenda for Cyclic Software Components. In Egidio Astesiano, editor, *FASE'98*, volume 1382 of *Lecture Notes in Computer Science*, pages 88–106. Springer Verlag, 1998.

[GJM+97] 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.

[GJS96] Gosling, Joy, and Steele. *The Java Language Specification*. Addison Wesley, 1996.

[Har88] David Harel. On Visual Formalisms. *Communications of the ACM*, 31(5):514–530, 1988.

[HL95] M. Hennessy and H. Lin. Symbolic Bisimulations. *Theoretical Computer Science*, 138(2):353–389, 1995.

[HLN+90] David Harel, Hagi Lachover, Ammon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-Trauring, and Mark Trakhtenbrot. STATEMATE: A Working Environment for the Development of Complex Reactive Systems. *IEEE Transactions on Software Engineering*, 16(3):403–414, 1990.

[KT97] Carron Kirkwood and Muffy Thomas. Towards a Symbolic Modal Logic for LOTOS. In *Northern Formal Methods Workshop NFM'96*, eWIC, 1997.

[Lig91] D. Lightfoot. *Formal Specification using Z*. Macmillan, 1991.

[Lut96] Mark Lutz. *Programming Python*. O'Reilly & Associates, 1996.

[MHSA99] Juan M. Murillo, Juan Hernández, Fernando Sánchez, and Luis A. Álvarez. Coordinated Roles: Promoting Re-usability of Coordinated Active Objects Using Event Notification Protocols. In Paolo Ciancarini and Alexander L.

Wolf, editors, *3rd International Conference, COORDINATION'99*, volume 1594 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[MKB97] Till Mossakowski and 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.

[Mos97] P. D. Mosses. CoFI: The Common Framework Initiative for Algebraic Specifiation and Development. In M. Bidoit and M. Dauchet, editors, *Proc. TAPSOFT'97*, number 1214 in Lecture Notes in Computer Science, Berlin, 1997. Springer Verlag.

[ORR+96] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414. Springer-Verlag, 1996.

[PCR99] Pascal Poizat, Christine Choppy, and Jean-Claude Royer. Concurrency and Data Types: a Specification Method. An Example with LOTOS. In J. Fiadero, editor, *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.

[Poi00] Pascal Poizat. *Korrigan : un formalisme et une méthode pour la spécification formelle et structurée de systèmes mixtes*. PhD thesis, Université de Nantes, November 2000.

[RAC99] G. Reggio, E. Astesiano, and C. Choppy. CASL-LTL: A CASL Extension for Dynamic Reactive Systems – Summary. Technical Report DISI-TR-99-34, DISI – Università di Genova, Italy, 1999. `ftp://ftp.disi.unige.it/person/ReggioG/ReggioEtAll99a.ps`.

[Rei95] Wolfgang Reif. The KIV-approach to Software Verification. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software - Final Report*, volume 1009 of *Lecture Notes in Computer Science*. Springer Verlag, 1995.

[RH97] J. Rathke and M. Hennessy. Local Model Checking for Value-Passing Processes (Extended Abstract). In Martín Abadi and Takayasu Ito, editors, *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.

[Roy01] Jean-Claude Royer. An Operational Approach to the Semantics of Classes: Application to Type Checking. *Programming and Computer Software*, to appear 2001. ISSN 0361-7688.

[RR00] G. Reggio and L. Repetto. CASL-Chart : A Combination of Statecharts and of the Algebraic Specification Language CASL. In T. Rus, editor, *International Conference on Algebraic Methodology And Software Technology, AMAST'2000*, volume 1816 of *Lecture Notes in Computer Science*, pages 243–257. Springer Verlag, 2000.

[SNW93] Vladimiro Sassone, Mogens Nielsen, and Glynn Winskel. A Classification of Models for Concurrency. In *Proceedings of Fourth International Conference on Co ncurrency Theory, CONCUR'93*, volume 715 of *Lecture Notes in Computer Science*, pages 82–96. Springer Verlag, 1993.

[Str87] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1987.