Argonaute: Graphical Description, Semantics and Verification of Reactive Systems by Using a Process Algebra

Florence Maraninchi
LGI, IMAG-CAMPUS, BP 53X
38041 GRENOBLE cedex
FRANCE
e-mail: maraninx@imag.imag.fr

Abstract

The ARGONAUTE system is specifically designed to describe, specify and verify reactive systems such as communication protocols, real-time applications, man-machine interfaces, ... It is based upon the ARGOS graphical language, whose syntax relies on the *Higraphs* formalism by D. Harel [HAR88], and whose semantics is given by using a process algebra. Automata form the basic notion of the language, and hierarchical or parallel decompositions are given by using operators of the algebra. The complete formalization of the language inherits notions from both classical process algebras such as CCS [MIL80], and existing programming languages used in the same field such as ESTEREL [BG88] or the STATECHARTS formalism [HAR87]. Concerning complex system description, ARGOS allows to describe intrinsic states directly — with the basic automaton notion — and only them: connections between components need no extra-state. The ARGONAUTE system allows to describe reactive systems graphically, to specify properties by means of temporal logic formulas, to produce a model on which logic formulas can be evaluated and to simulate an execution of the system described, by using the external graphical form to show evolutions. We present the global structure and functionalities of the ARGONAUTE system, and the theoretical basis of the ARGOS language.

1 Introduction

We are interested in the general field of the validation of *reactive systems*, so called by D. Harel and A. Pnueli [HP85]. Reactive systems include real-time applications, communication protocols, or man-machine interfaces. Dealing with reactive system validation includes describing them with some appropriate description language, simulating executions, and specifying the expected behaviour. These various aims are considered as more or less important when building a tool for the validation of reactive systems.

1.1 Validation of reactive systems

The languages for the description of reactive systems follow different approaches.

Some languages, like ESTEREL [BG88], are aimed at *programming* reactive kernels of complex systems. ESTEREL offers high level design concepts and modular development. Validation includes simulation and verification by graph reduction methods.

Other languages are aimed at describing complex systems in the most readable way. The *state-transition* approach and the graphical one fall into this part. In describing communication protocols, for instance, one often starts drawing automata, which have to be translated into authorized constructions of a language. A language in which direct automata descriptions are authorized constructions is therefore interesting. However, human design of complex and large automata is impractical. For instance, the CCITT SDL [SDL] for communication protocols deals with one level concurrency between machines. There is no way to introduce hierarchic design, and even simple protocols have complex graphical descriptions. An interesting language following the same approach is the STATECHARTS formalism [HAR87]. Here, the state-transition formalism is extended to allow hierarchic representation with OR or AND decompositions of states. Since automata can be represented in a concise manner, complex systems with numerous states can be described. However, the design is not really modular, and the complete figure is needed to understand the behaviour of a system. This is a great limitation to the approach. The STATECHARTS formalism is the fundation of the STATEMATE visual working environment for the design of complex systems [STMA]. The STATEMATE environment allows interactive building of systems, and provides a good tool for the simulation of executions, or various evaluations. Moreover, the language is given a formal semantics, and global properties such as the absence of deadlock can be verified for the system described.

Finally, concerning the validation, an interesting approach is the specification by means of temporal logic formulas which describe the expected behaviour of a system. A finite state graph model is built from the description of the system with some appropriate description language, and the logical formulas are evaluated on this model. The description language must have a formal semantics to allow automatic generation of the state graph model. Moreover, when a specification formula is false, it is interesting to provide the user with a *diagnostic* relating this result with the external form of the description language. XESAR [XES] and CAESAR[GAR89] follow this approach. The description languages of CAESAR and XESAR are respectively LOTOS [LOT] and ESTELLE/R, a variant of ESTELLE [EST86]. As these languages were not designed while taking into account the model checking approach, they must be adapted in order to allow the user to relate the system descriptions with the specification formulas. In XESAR, when a formula is false, a specific tool called CLEO [RAS88] can give a diagnostic of the form: "the formula is false because the system can reach a state that do not verify it, by executing the following sequence of actions". The states involved in a diagnostic are states of the model, which can be hard to relate with the description of the system in ESTELLE/R. On the other hand, the two languages lack high level design concepts that could allow to describe a system by successive refinement stages.

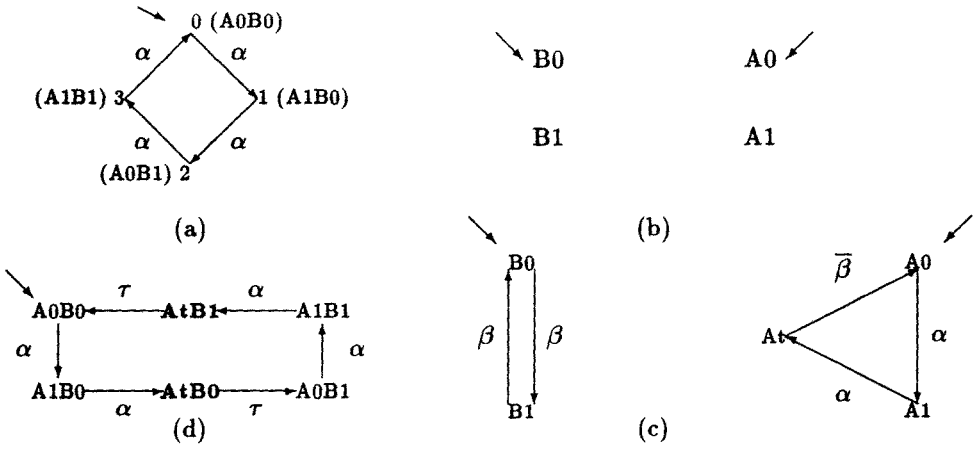

Figure 1: Describing intrinsic states only

1.2 Main objectives for a description language

We are mostly interested in the specification of reactive systems with model checking based methods. We want to design a description language by taking into account the characteristics of such languages required in a specification environment like XESAR, as stated above. Moreover, the language must have good properties for the designer. This includes readability, as intented by the automata based graphical languages, and high level design concepts such as hierarchical decomposition.

In the model checking approach, the first objective is to limitate, if possible, the size of the model generated. Indeed, finite models have been proved interesting, especially for protocol validation, but complex systems can lead to very large models, therefore limiting the interest of the model checking approach. This phenomenon has several reasons, among which the intrinsic complexity of the problem described. But with classical description languages, states can appear in the description, which do not correspond to *intrinsic- states* of the system. We call them *extra-states.*

We are interested in the state-transition approach, with graphical syntax. An automata based language allows to describe intrinsic states of basic behaviours directly. The basic automaton notion is used together with powerful constructs such as hierarchical or parallel composition, which are given a graphical syntax too, and a compositional semantics. This is the only way to provide modular development and splitted graphical representation. But, when trying to introduce those high level concepts, one must keep in mind that we want to describe intrinsic states of the systems only. Constructs like parallel composition must not need adding "extra"-states in order to express links between components.

Let us observe a modulo 4 α counter. The intrinsic states of this system correspond to the four possibles values of the counter: 0, 1, 2 and 3. When describing such a system in a state-transition frame, two approaches can be followed. We can represent the four intrinsic states directly, and build a single automaton with the four expected transitions.

(see figure 1, a). We stated above that this solution cannot be used when the system has numerous states. We can also decompose the set of states. Figure 1(b) shows the decomposition of the counter into two bits. With these two bits, we want to build two automata, and to compose them with an appropriate operator in order to obtain the global behaviour described in (a).

Here, the state decomposition correspond to the usual parallel concept. But if the parallel operator semantics is not well chosen, we cannot build (a) with two automata containing only the states of (b). For instance, when describing the counter with the asynchronous model of CCS, one is led to introduce an extra-state AT, and an internal event β in order to express the communication between the two automata. (see c). The counter is described by (bitB | bitA)$\backslash\beta$ where bitA $= \alpha\alpha\overline{\beta}$ bitA and bitB $= \beta\beta$ bitB. The corresponding state graph is given by (d): it contains *two* extra-states ATB0 et ATB1. This situation is neither particular to our example, nor due to the way it is modelled, and appears currently with formalisms like CCS.

The above remarks show a way to design a description language that takes into account the main requirements of specification environments. The basic notion is automata, with graphical syntax. Operations provide a way to structure complex systems descriptions. They must be carefully defined to meet four aims: they can be used without leading the designer to introduce extra-states in the basic behaviours, in order to express their links; their relation with informal design concepts is clear; their semantics is well defined; they possess a readable graphical syntax.

1.3 The ARGOS language and the ARGONAUTE system

We propose the ARGOS language and the ARGONAUTE system, for the description and the validation of reactive systems. ARGOS has a graphical syntax derived from the *Higraphs* formalism proposed by D. Harel [HAR88] and used in the STATECHARTS formalism. However, ARGOS is not an attempt to give another formal semantics of the STATECHARTS, as it can be found in [HGR88]. The formal definition of this new language relies on a process algebra, whose basic terms describe automata and whose operators capture the high level design concepts of parallel or hierarchical composition. The latter cannot be taken from classical process algebras, since they lack hierarchical concepts. They must be adapted from notions of other languages, such as ESTEREL. Each operator allows to express communication between components which contain intrinsic states only. Moreover, the underlying process algebra gives good properties to the graphical language: it allows to describe complex systems by several figures, since the design can be really modular.

Concerning the system functionalities, ARGONAUTE is similar to XESAR [XES] or CAESAR [GAR89]. The formal semantics leads to an automatic model generation tool that builds only the *intrinsic* states of the system described. Moreover, diagnostics produced by the evaluation of logical formulas can be related easily to the external graphical form of the system.

In the following sections, we first present the ARGOS language informally. Then we give some details about its formal semantics, by comparing it with existing algebras and languages. We end with an overall description of the ARGONAUTE system functionalities.

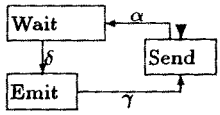

Figure 2: Automaton description

2 Informal description of ARGOS

We describe the basic automaton notion first; then we describe the constructs of ARGOS, which give composed objects called *processes.*

2.1 Automata

As above mentioned, automata constitute the basic notion of the language. One can describe them by giving explicit *states* and *labeled transitions.* The graphical syntax of automata is given by figure 2. The *initial* state of the automaton is marked by a little arrow without label.

Labels describe *events* which determine the evolution of the system modeled by the automaton. Events, like events in process algebras, are abstract notion, and one can give them various meanings, such as: "acknowledgement is passed from the medium to the emitter" or "alarm beep is activated" or "connection attempt ends correctly", and so on. In particular, events may represent input or output of the system as well. A transition leading from state A to state B with label α means: *by taking into account event α the systems evolves from state A to state B.*

We show in section 2.2 that label can be structured in two fields, the *toggle event* field, and the *generated events* field, in order to control interactions between automata.

2.2 Parallel composition

Parallel composition is a powerful design concept which appears in classical process algebras as the parallel operator. In ARGOS, it has a particular semantics chosen by taking into account the objectives of section 1.2.

Figure 3 shows a modelization of the mutual exclusion. Two users, represented by two automata *User1* and *User2*, compete for the use of a resource, represented by the automaton *resource*. *User1* can request the use of the resource with *request1* and liberate it by *free1*. The behaviour of *User2* is symmetrical. The *resource* can leave state FREE to enter state USED by taking into account *request1* or *request2*. Then it can reenter the FREE state by *free1* or *free2*. The global system is a *process* built with the parallel operator from the three automata: the parallel *components*. A state of the global process is a configuration of states of the components, for instance R1/FREE/R2.

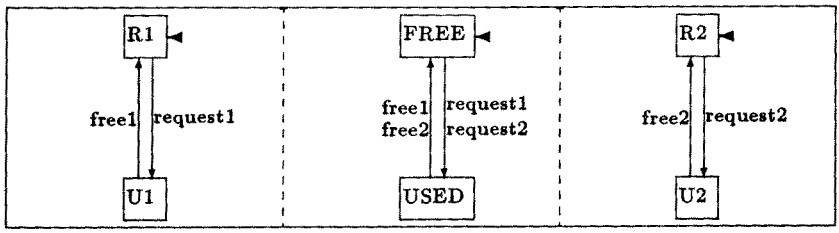

Figure 3: The mutual exclusion example

The behaviour of the process is as follows:
The *initial* state is given by the initial states of the components: R1/FREE/R2.

For each event α, all components where α appears must evolve together by α or not. These evolutions are independent from those of the components where α does not appear. The result is the evolution of the global system by α. In our example, starting from the initial state, one request, say *request1*, can be taken into account. The system then enters global state U1/USED/R2. Then, a request from the second user cannot be taken into account, since the resource cannot execute *request2*. The only possible action is *free1*, which leads to the initial state.

This describes n-ary *rendez-vous*. The parallel operator has good properties such as commutativity or associativity, which are fundamental when dealing with a graphical language, where unassociative operators needing explicit parenthesis are uneasy to draw.

However, our definition has two major drawbacks. First, the structure of synchronization is *rigid*, because *each* occurrence of an event α must synchronize *always* or *never* with *each* occurrence of the same event in another component. There is no way to specify that some occurrences must give way to synchronization, and some other must not.

On the other hand, the definition is *symmetrical*. The situation where two components are blocked on an α transition — as in the mutual exclusion example — is a particular one. In other situations, it is very difficult to express that one component is blocked, while the other is not, by using our symmetrical definition of parallel composition.

This can lead to introduce extra-states. However, to force synchronization is never easy if it is not a built-in feature. Rigidity of our definition is therefore sometimes the good tool. We can keep this definition if another feature allows to describe situations where the synchronization is less rigid.

The *generated events* field of transition labels mentioned in section 2.1 provides this feature. It suppresses the drawbacks of parallel composition defined with *rendez-vous* only. It allows to build the transitions of a composed system from those of its components without being bound to introduce extra-states in those components.

We describe informally the global behaviour of the system shown by figure 4. The initial state is A/C/E. When event α occurs, the first component reaches state B and generates event β. This means that *this* occurrence of α must synchronize, *if possible*, with a β-labeled transition somewhere in the remainder of the system. Here the remainder is in state C/E, where event β can be taken into account. Therefore, the global evolution

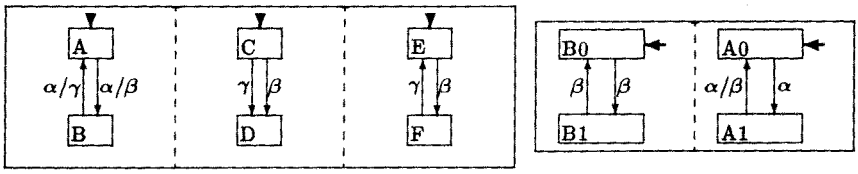

Figure 4: A simple example for the use of generated events, and the 2-bit α counter

is from state A/C/E to state B/D/F by α : all components who are able to react do. This is the definition of the *broadcast* of β. In state B/D/F, when α occurs, the first component reaches state A and generates γ. (This occurrence of α has not the same effect as the other). The second component is in state D and cannot react to γ. Hence the third component cannot react neither, because of the parallel operator definition. The global evolution is from B/D/F to A/D/F by α.

Figure 4 shows the description of the two-bit α counter with generated events. The global behaviour is as expected, and contains only the four intrinsic states. To our opinion, the generated event feature allows to use parallel composition freely in order to express global states of a system in a concise manner. All behaviours of the global system (i.e. sets of transitions between global states) can be obtained without introducing extra-states in the basic components.

The generated events feature is similar to the emitted signals of ESTEREL or the generated events of the STATECHARTS formalism. Our semantics is based upon the *synchrony hypothesis* adopted by ESTEREL, although we show that the problem is not the same in this language and in ARGOS. In the STATECHARTS formalism, semantics of generated events is computed non-deterministically, which considerably limits the cases in which it could help, see [HPSS86] and [HGR88]. Section 3 give details on semantics of generated events, relations between this feature and determinism, differences between ARGOS and ESTEREL.

2.3 An operator for the hierarchical design concept

This feature is inspired originally by the refinement operation of the STATECHARTS formalism, which is related with high level concepts as top-down or bottom-up problem analysis. It has no equivalent in classical process algebras.

When trying to build an algebra operator to capture the intuitive semantics of the refinement operation, one must solve several problems, since the refinement of the STATECHARTS is much more an economy in drawing edges of a global automaton than a conceptual decomposition into processes. These problems happen to be crucial for the development of the ARGOS language. The main idea of our definition is to involve two *processes*, a *controller* and a *controlled* one, in an *asymmetrical control* operation. The graphical syntax shows the representation of the controlled process in the box which corresponds to the state of the controller being "refined". That is, the state which defines

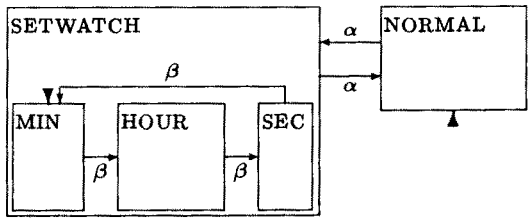

Figure 5: Example of the asymmetrical control operation: a simple wristwatch

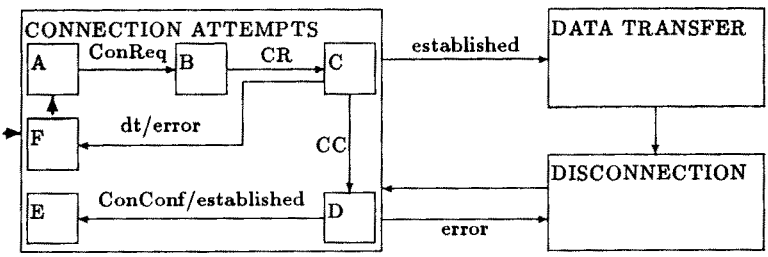

Figure 6: Example of asymmetrical control: a communication protocol

how the controller performs its control.

Let us observe figure 5. The controller maintains the states of a wristwatch interface: NORMAL mode, SETWATCH mode... The controlled process refines the SETWATCH mode, to show HOUR setting, MINUTE setting and SECONDE setting.

Each transition of the controller with target SETWATCH *starts* the controlled process in its initial state, that is MINUTE setting. On our example, this is done by taking into account event α, related to one button of the wristwatch. The three states of the controlled process form a cycle which can be run over with another button, related to event β. Each event β is taken into account by the controlled process to let the user run over the cycle, while the controller remains in SETWATCH state. When event α occurs, no matter what the internal state of SETWATCH mode is, the wristwatch leaves this state to reenter NORMAL one: the controller executes a transition which leaves the control state, and therefore *kills* the controlled process: it can no longer take events into account, and the information about its state is lost.

Now, let us observe another example, see figure 6. It shows a simple communication protocol, with connection and disconnection. The protocol data units are *CR* (connection request), *CC* (connection confirmation) and *dt* (data transfer). The service data units are *ConReq* (connection request) and *ConConf* (connection confirmation). The main process, called *Transaction*, maintains the states of the protocol with respect to the connection being established or not: CONNECTION ATTEMPTS, DATA TRANSFER, ERROR AND DISCONNECTION. The CONNECTION ATTEMPTS state controls a sequential process called *ConnAtt* in charge to establish the connections. Once it has been started, this

process evolves by *ConReq*, *CR*, *CC*, *ConConf* and *dt* while the controller remains in the control state, since these events cannot make it leave this state. When the controlled process *ConnAtt* reaches the state E where the connection is established, it generates the event *established*. This transition must synchronize, if possible, with a transition labeled by *established* of another process. This is possible, because the controller and the controlled process can evolve together. The transitions labeled by *ConConf/established* and *established* are executed simultaneously, and give a transition labeled by *ConConf* from CONNECTION ATTEMPTS/E to DATA TRANSFER. The state of the controlled process is then irrelevant, since it has been killed by the controller leaving the control state. The global system behaviour is similar when *dt* occurs in state C : *ConnAtt* reaches state F, while generating event *error*. *Transaction*, by taking into account event *error*, reaches state DISCONNECTION.

The asymmetrical control operator allows to decompose the set of the global intrinsic states of a system with a notion of top-down analysis, as in the STATECHARTS philosophy of system description. The semantics of the operator, and the generated events feature, then allow to express all kinds of global behaviours by relating two *processes*. In the first example (figure 5), the controlled process has a infinite behaviour; in the second one (figure 6), it has a terminating behaviour. This captures the STATECHARTS inter-level transition too. The generated events feature gives a way to relate a particular termination of the controlled process with a particular transition of the controller. See figure 6, with the two termination causes : established and *error*. Moreover, the compositionnal semantics of the new operator allows to represent the two levels separately in a readable way.

As above defined, the asymmetrical control introduces the UPTO construction of ESTEREL [BG88] as an algebra operator. When an event labeling a transition occurs, the source state of the transition is exited, no matter what the present state of the controlled process is. Notice that the controlled process *reacts* at the instant when this event occurs. Indeed, we rely on this choice to realize inter-level transitions by means of generated events. However, it is possible to give the asymmetrical control operator a semantics where the controlled process does *not* react when an event which exits the control state occurs. We must distinguish two kinds of automata transitions. For the first kind, the semantics of the operator is as previously defined; conversely, when a transition of the second kind is taken, the process controlled by its source state does not react. With these two kinds of transitions, we are provided with a way to express local priority between events: the event which causes a control state to be exited has priority upon all possible events from the present state of the controlled process.

2.4 Internal events

When using generated events, we say: "execute generated event if this is possible somewhere in the system". This states implicitly that the generated event is broadcasted into the whole system, and must be taken into account everywhere in it. It is sometimes desirable to limit the broadcast to some part of a system. For this purpose, we introduce a new operator, which allows to declare an event to be *internal* to some part of the system. We give this operator a graphical syntax by using a rectangular box. See figure 7. The new operator has two effects.

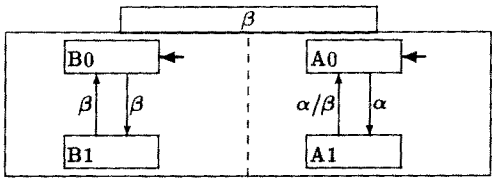

Figure 7: The two bits α counter, declaring event β to be internal

When β is generated within the scope of the operator which declares it as internal, we must try to execute it in this scope only. If it isn't possible, β is simply lost. The transition which generates it cannot be synchronized with a β-labeled transition out of the scope. This limits the broadcast.

In the scope where β is declared as internal, it cannot be executed if not generated by a local transition whose toggle event is not β. Since the name β disappears when it is made internal, it is no more visible out of the scope.

The internal event operator provides as an algebra operator the ESTEREL notion of local signals.

3 Determinism and causality

When dealing with generated events, causality problems arise. The non-deterministic solution adopted in the STATECHARTS for all causality problems cannot be applied here. Indeed, we relie on the determinism of our language constructs to met the aim of describing only intrinsic states of the system. We present first the main ideas of the model generation, as detailed in [MAR89]. Then we give the causality problems, and show how they are solved.

3.1 Generation of the model

The formal operational semantics defines how the state graph model is obtained from a set of automata descriptions, and a term of the algebra which describes a structure involving those automata. The model of an automaton is given by the automaton itself, considered as a state graph. The states of the model are state configurations built from the automata which compose the system described. The transitions of the model are labeled by Γ/Δ, where Γ and Δ are some sets of events defined below. The complete state graph model is defined by a set of rewrite rules à la Plotkin.

The first idea is to build *multi-event* transitions, from the basic automata transitions. A label α/β means that we must try to execute α and β simultaneously. Therefore, we must define evolutions of a system by $\{\alpha,\beta\}$. Since automata labels only provide $\{\alpha\}$ transitions, the only way to produce a transition labeled by $\{\alpha,\beta\}$ is to take an α-labeled transition in one component and a β-labeled one in another component. These

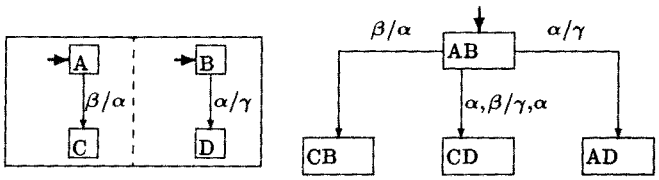

Figure 8: A parallel composition with generated events, possible multi-event evolutions from initial state

components can be parallel components, or the controlled process and the controller involved in an asymmetrical control operation. Figure 8 shows the multi-event transitions built from the transitions of the two automata involved in the parallel composition of figure 8.

The second idea concerns the last operator: when declaring an event α to be internal, we can take into account the effects of generating α within the scope of its declaration. No information about further components is needed, since α is made local.

The system in which we declare α as internal has global transitions Γ/Δ of three kinds:

- $\alpha \in \Delta, \alpha \notin \Gamma$ (see figure 8, AB to CB by β/α).
  We must try to execute α and β simultaneously from state AB. This is possible if and only if there exists a transition $\{\alpha,\beta\}/\Lambda$ from AB (it has been produced when computing the parallel composition). There exists AB to CD by $\{\alpha,\beta\}/\{\alpha,\gamma\}$. Therefore we produce a transition AB to CD by β/γ for the system where α is internal.

- $\alpha \in \Gamma$ (see transitions AB to AD by α/γ and AB to CD by $\{\alpha,\beta\}/\{\alpha,\gamma\}$).
  These transitions cannot be taken into account, since α cannot be executed if not generated. The only use of such transitions is by the preceding law.

- $\alpha \notin \Gamma \cup \Delta$.
  These transitions are not concerned by α. They remain unchanged in the composed system.

The resulting behaviour of the global system, from its initial state, is reduced to one transition, from AB to CD by β/γ.

3.2 Relations between events and model generation

When an α-labeled transition and a β-labeled one have the same source state, we state that events α and β cannot occur simultaneously, since we could not give a deterministic semantics for the automata described. In the mutual exclusion example of figure 3, we state that *request1* and *request2* cannot occur simultaneously. Each automaton gives

a set of *relations* $\alpha\#\beta$ meaning "α and β cannot occur simultaneously". This kind of relation can be found in the event structures of [WIN80]. The relations are global to the system which involves the automata. Moreover, the user can give explicit relations between events, which are not induced by the automata descriptions.

When building the transitions of a composed system from those of its components, one must apply two laws. First, if $\alpha\#\beta$, we must not build transitions labeled by Γ/Δ where $\alpha \in \Gamma$ and $\beta \in \Gamma$. This reduces the size of the model generated. On the second hand, when we are able to build a transition labeled by Γ/Δ, we must check the set $\Pi = \Gamma \cup \Delta$ with respect to the global relations of the system. If there exists some relation $\alpha\#\beta$ such that $\alpha \in \Pi$ and $\beta \in \Pi$, an error occurs. Indeed, the above relation states that events α and β never occur simultaneously, while the transition we are trying to build requires all the events of Π to be executed simultaneously. Detecting such an error refuses the composed system as non well-formed.

3.3 Causality problems

A causality problem is encountered when the composition of several transitions labeled Γ_i/Δ_i leads to a global label $\bigcup_i \Gamma_i / \bigcup_i \Delta_i$ where $\bigcup_i \Gamma_i \cup \bigcup_i \Delta_i$ contains two events α, β involved in a relation. We stated above that such a situation leads to the system being refused. Let us observe the above example (see figure 8). If events β and γ are defined informally by: "the signal a is not present and the signal b is" and "the signal a is present", then we have the classic causality problem where by supposing that signal a is not present, we are led to generate it, see [BG88]. In ESTEREL, the user can label its transitions by a and $\neg a \wedge b$ explicitly. The relation between a and $\neg a$ is known by the language and used to refuse situations which lead to causality problems. In ARGOS, the language knows β and γ only. The user must then state explicitly that $\beta\#\gamma$. By giving this relation, he allows the language to detect the causality problem, and to reject the system.

Our solution to the causality problem is based upon the ESTEREL notion of *relation* between events, with a slight difference. As events are an abstract notion, unstructured, internal causality problems occur with relations given by automata descriptions only. If the user does not give more information about the informal meaning of events, by using relations, the system composed is causally correct.

4 The ARGONAUTE system functionalities

Figure 9 shows general organization of the system. The graphical interactive environment of ARGONAUTE allows to manipulate the graphical description of a system on the screen. It can be used to build systems and formulas as well, or to show evolutions during a simulation.

The system description editor provides interactive building and step by step control of the contextual syntax of the language. One starts building automata by assembling states and transitions, naming the ones and labeling the others. No control is performed during this phase. When an automaton description is finished, the user can request the system

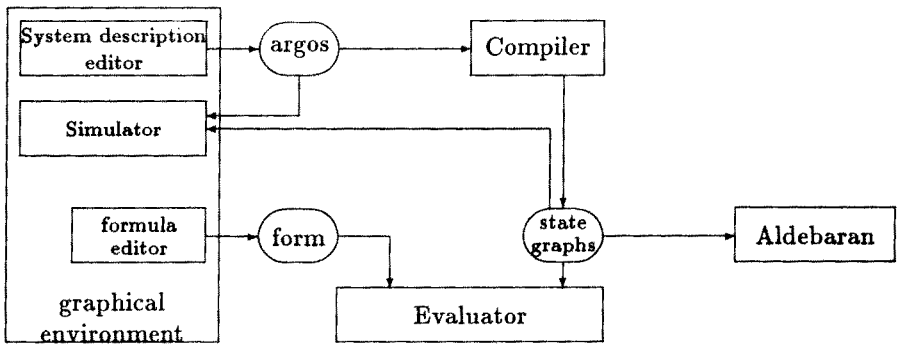

Figure 9: Overall description of ARGONAUTE

to control it, with respect to the contextual syntax defined formally for automata. The system answers by giving a diagnostic if the automaton is not correct, or by marking it as usable for further constructs, if it is correct. The user can build complex structures, by using operators of the algebra. Each operation can require a control of the contextual syntax, depending on the nature of the operator employed. Each process or automaton of the session is presented in a window, which can be manipulated by the user according to classical window-manager functionalities (moving, resizing, exposing...). The system uses a global menu to provide file operations, or operations concerning all the processes of the session, and an icon menu which allows to build automata starting from boxes and arrows. A system description is saved by using the textual format *argos*.

Temporal logic formulas are built with temporal operators, like POT or INEV, classical logic operators and *basic predicates*. *Basic predicates* express properties of the current state of a system description, as variable values, or situation with respect to the execution of transitions. They must be expressed by using the external form of the description language, i.e. exact names of variables (including paths if there is a notion of scope), or transition names. In ESTELLE/R, an variant of ESTELLE has to be made to allow the user to designate and name certain transitions. On the other hand, editing basic predicates can be done by looking at the system description to build by hand the complete names of the objects to be referred to, but this is not an easy task. As the internal and the external forms of ARGOS are close, states and transitions are notions of the external form too. This provides an easy way to express basic predicates such as *at(state A)* or *enable(trans t)*. Moreover, the ARGONAUTE system provides a formula editor, in which basic predicates can be built by designating objects on the graphical description of a system. For instance, the basic predicate *at(state A)* is produced when the user chooses interactively the state A on the graphical description of his system. If needed, ARGONAUTE automatically produces the complete name of the object designated, or whatever information to be added to the internal form of the formulas in order to ensure unambiguity. The formulas are saved according to the *form* format, which is chosen to allow easy adaptation of the XESAR toolkit formula evaluator. When a formula is false, both states and transitions appearing in the basic predicates of the diagnostic are related easily to the external form of the language.

When the system is compiled, the resulting state graph model is save with the *state graph*

format. It is used by the formula evaluator and by the graphical simulator. It can also be used as an entry by the ALDEBARAN reduction tool [FER88], which allows reductions according to various equivalence and congruence relations.

5 Conclusion

The system proposed belongs to the CESAR family. But it differs from other tools: the description language has been especially designed while keeping in mind special constraints of the model checking approach, as the size of the model produced or the need of informative diagnostic when a formula is false. These two aims are taken into account when building a language whose internal and external forms are close. Moreover, concerning the first aim, we think that our language allows to describe a complex system (whose intrinsic states are yet numerous) without introducing "false" states due to inter-process communication.

On the other hand, the language and its semantics have been designed together. Formal semantics is well defined, and leads to a tool for the automatic generation of the state graph model.

As far as theoretical work is concerned, ARGOS can be viewed as an attempt to build a language with both formal concepts of process algebras and high level concepts of existing programming languages. This work seems fructuous as regards to the language designed, and also with respect to the formal definition of high level, but informal, concepts like bottom-up analysis.

Further theoretical work

Further theoretical work has two major areas: extending the langage kernel, and providing the user with higher level constructs which can be built easily with the present kernel.

To extend the language kernel, we can think about introducing event values, such as signal values of ESTEREL, or shared variables which can be tested and set by components. On the other hand, we mentionned in section 2.3 the introduction of a basic mechanism to allow local priority between events.

As far as high level constructs are concerned, it is already possible to define state temporal constraints proposed by D. Harel [HAR84] and modeled explicitly in [MAR87], by using the generated event feature. However, one must keep in mind that several advantages due to the proximity of the internal and the external forms of the language, may disappear if the user is provided with high level constructs whose semantics is given via a translation into lower level operators. More accurately, the external form of an internal state could be hard to build. Nevertheless, if the new constructs still express intrinsic states, and the internal communication generating no state is not called in question again, the major advantage of ARGOS does not disappear: states of the model really correspond to intrinsic states.

System improvement

The present version of the ARGONAUTE system is a prototype. We wanted to get experience about programming graphically, in order to define what tools must be available in such an environment. This work has made obvious that complete graphical representations of complex systems is impractical. However, our language has good properties w.r.t. compositional design, and should therefore be usable to describe systems by giving separately the graphical descriptions of automata, and the structure in which they are involved. Graphical description and the notion of area are worth looking for pure automata or asymmetrical control operations, but they could be avoided for parallel composition. In all cases, our definition of the kernel constructs allows one to understand the global behaviour of a system without being bound to get the whole graphical representation.

References

[BG88] G. BERRY, G. GONTHIER, *The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation*, ENSMP-INRIA, Sophia-Antipolis, 06565 Valbonne - France (1988).

[EST86] *Estelle: A Formal Description Technique Based on an Extended State Transition Model*, ISO/TC97/SC21 (1986).

[FER88] J.C. FERNANDEZ, *Aldebaran : un système de vérification par réduction de processus communicants*, thèse, Université Joseph Fourier Grenoble (1988).

[GAR89] H. GARAVEL, *Compilation et vérification du langage Lotos*, thèse, Université Joseph Fourier Grenoble, to appear in 1989.

[HAR84] D. HAREL, *Statecharts: A Visual Approach to Complex Systems*, First version, Dept. of Applied Math., Weizmann Institute of Science, Rehovot, Israel (1984).

[HAR87] D. HAREL, *StateCharts : A visual Approach to Complex Systems*, Science of Computer Programming, Vol. 8-3, pp. 231-275 (1987).

[HAR88] D. HAREL, *On Visual Formalisms*, CACM vol. 31, no 5 (1988).

[HGR88] C. HUIZING, R. GERTH, W.P. DE ROEVER, *Modelling Statecharts Behaviour in a Fully Abstract Way*, 13^{th} CAAP, LNCS 299, Springer Verlag, (1988).

[HP85] D. HAREL, A. PNUELI, *On the Development of Reactive Systems*, Logic and Models of Concurrent Systems, Proc. NATO Advanced Study Institute on Logics and Models for Verification and Specification of Concurrent Systems, NATO ASI Series F, vol. 13, Springer-Verlag (1985).

[HPSS86] D. HAREL, A. PNUELI, J.P. SCHMIDT, R. SHERMAN., *On the Formal Semantics of Statecharts*, Proc. Symposium on Logic in Computer Science (LICS) pp 54-64 (1986).

[LOT] *LOTOS: A Formal Description Technique*, ISO/TC97/WG16-1 (1984).

[MAR87] F. MARANINCHI, *Statecharts: sémantique et application à la spécification de systèmes*, DEA, INP Grenoble (1987).

[MAR89] F. MARANINCHI, *Sémantique du langage* ARGOS, unpublished (1989).

[MIL80] R. MILNER, *A Calculus of Communicating Systems*, Springer-Verlag, LNCS 92 (1980).

[RAS88] A. RASSE, *CLEO, Interprétation de la non-correction de programmes sur un modèle*, RT C10, Spectre project, LGI-IMAG Grenoble (1988).

[SDL] *CCITT SDL: overview*, Computer Networks and ISDN Systems, vol. 13, Number 2 (1986).

[STMA] *The STATEMATE Working Environment for System Development*, AD CAD Ltd., Rehovot, Israel (1987).

[WIN80] G. WINSKEL, *Events in Computations*, PhD Thesis, University of Edinburgh (1980).

[XES] J.L. RICHIER AND C. RODRIGUEZ AND J. SIFAKIS AND J. VOIRON, *XESAR: A Tool for Protocol Validation. User's Guide*, LGI-Imag (1987).