From Informal Requirements to COOP: A Concurrent Automata Approach

Pascal Poizat[1], Christine Choppy[2], and Jean-Claude Royer[1]

[1] IRIN, Université de Nantes & Ecole Centrale
2 rue de la Houssinière, B.P. 92208, F-44322 Nantes cedex 3, France
{Poizat, Royer}@irin.univ-nantes.fr
http://www.sciences.univ-nantes.fr/info/perso/permanents/poizat/
phone: +33 2 51 12 58 22 — fax: +33 2 51 12 58 12
[2] LIPN, Institut Galilée - Université Paris XIII,
Avenue Jean-Baptiste Clément, F-93430 Villetaneuse, France
Christine.Choppy@lipn.univ-paris13.fr

Abstract. Methods are needed to help using formal specifications in a practical way. We herein present a method for the development of mixed systems, i.e. systems with both a static and a dynamic part. Our method helps the specifier providing means to structure the system in terms of communicating subcomponents and to give the sequential components using a semi-automatic concurrent automata generation with associated algebraic data types. These components and the whole system may be verified using common set of tools for transition systems or algebraic specifications. Furthermore, our method is equipped with object oriented code generation in Java, to be used for prototyping concerns. In this paper, we present our method on a small example: a transit node component in a communication network.
Keywords: Concurrent systems, specification method, automata, object oriented (Java) code generation.

Stream: Foundations and Methodology
Mini-Track: FoSS (Foundations of Software Specifications)

1 Introduction

The use of formal specifications is now widely accepted in software development. Formal specifications are mainly useful to provide an abstract, rigorous and complete description of a system. They are also essential to prove properties, to prototype the system and to generate tests. The need for a method that helps and guides the specifier is another well-known fact. A last point is the need for mixed specifications: *i.e.* specifications able to describe both the dynamic (process control) and the static aspects (data types). We think that mixed specifications also enable, at a specification level, to have a clear separation of concerns between these two aspects of systems that should be orthogonal as advocated (at the implementation level) by recent Concurrent Object Oriented Programming (COOP) research.

We herein present a method based on LOTOS [7, 20] and SDL[1] [11] experiences [25, 24]. Our method was first presented in [26] and is here elaborated in terms of *agenda* and extended to Java code generation. We chose to describe our method in terms of the *agenda* concept [17, 16] because it describes a list of activities for solving a task in software engineering, and is developed to provide guidance and support for the application of formal specification techniques. Our method mixes constraint-oriented and state oriented specification styles [33] and produces a modular description with a dynamic behaviour and its associated data type.

The dynamic behaviour extraction is based on a guarded automaton that is progressively and rigorously built from requirements. Type information and operation preconditions are used to define states and transitions. The dynamic behaviour is computed from the automaton using some standard patterns. The last improvement is the assisted computation of the functional part. Our method reuses a technique [3] which allows one to get an abstract data type from an automaton. This technique extracts a signature and generators from the automaton. Furthermore, the automaton drives the axiom writing so that the specifier has only to provide the axioms right hand sides.

Our method is extended here to code generation. Code generation is a really useful tool from a practical point of view. It allows to generate from a specification a prototype which may be used as the basis for the future system, to validate client requirements or to test the system. We use Java [15] as a target language for the static part and we focus on the dialect ActiveJava [1] for the dynamic part.

The paper is structured as follows. We briefly present the case study: a transit node case in a telecommunications network [5]. In Section 3, the general process of our method is given. Section 4 is devoted to code generation, namely it consists in two subsections: the static generation part in Java and the dynamic generation part in ActiveJava. The conclusion summarizes the main points of our method.

2 Case-Study Presentation

This case study was adapted within the VTT project [5] from one defined in the RACE project 2039 (SPECS : Specification Environment for Communication Software). It consists of a simple transit node where messages arrive, are routed, and leave the node. The informal specification reads as follows:

clause 1 The system to be specified consists of a transit node with: one *Control Port-In,* one *Control Port-Out,* N *Data Ports-In,* N *Data Ports-Out,* M *Routes Through.* The limits of *N* and *M* are not specified.

[1] Both are used for the specification of distributed systems and are mixed specification languages.

clause 2 **(a)** Each port is serialized. **(b)** All ports are concurrent to all others. The ports should be specified as separate, concurrent entities. **(c)** Messages arrive from the environment only when a *Port-In* is able to treat them.

clause 3 The node is "fair". All messages are equally likely to be treated, when a selection must be made,

clause 4 and all data messages will eventually transit the node, or become faulty.

clause 5 *Initial State :* one *Control Port-In,* one *Control Port-Out.*

clause 6 The *Control Port-In* accepts and treats the following three messages:

(a) *Add-Data-Port-In-&-Out(n)* gives the node knowledge of a new *Port-In(n)* and a new *Port-Out(n).* The node commences to accept and treat messages sent to the *Port-In*, as indicated below on *Data Port-In.*
(b) *Add-Route(m,n_i)* , associates route *m* with *Data-Port-Out(n_i).*
(c) *Send-Faults* routes some messages in the faulty collection, if any, to *Control Port-Out.* The order in which the faulty messages are transmitted is not specified.

clause 7 A *Data Port-In* accepts only messages of the type : *Route(m).Data.*

(a) The *Port-In* routes the message, unchanged, to any one (non determinate) of the open *Data Ports-Out* associated with route *m*. If no such port exists, the message is put in the faulty collection.
(b) (Note that a *Data Port-Out* is serialized – the message has to be buffered until the *Data Port-Out* can process it).
(c) The message becomes a faulty message if its transit time through the node (from initial receipt by a *Data Port-In* to transmission by a *Data Port-Out*) is greater than a constant time *T*.

clause 8 *Data Ports-Out* and *Control Port-Out* accept messages of any type and will transmit the message out of the node. Messages may leave the node in any order.

clause 9 All faulty messages are eventually placed in the faulty collection where they stay until a *Send-Faults* command message causes them to be routed to *Control Port-Out.*

clause 10 Faulty messages are **(a)** messages on the *Control Port-In* that are not one of the three commands listed, **(b)** messages on a *Data Port-In* that indicate an unknown route, or **(c)** messages whose transit time through the node is greater than *T*.

clause 11 **(a)** Messages that exceed the transit time of T become faulty as soon as the time T is exceeded.
(b) It is permissible for a faulty message not to be routed to *Control Port-Out* by a *Send-Faults* command (because, for example, it has just become faulty, but has not yet been placed in a faulty message collection),
(c) but all faulty messages must eventually be sent to *Control Port-Out* with a succession of *Send-Faults* commands.

clause 12 It may be assumed that a source of time (time-of-day or a signal each time interval) is available in the environment and need not be modeled within the specification.

3 A New Specification Method

Overall Presentation

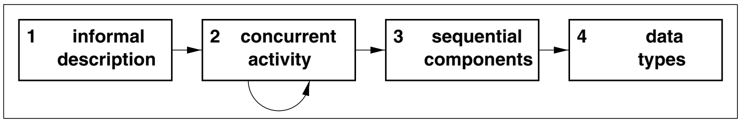


Fig. 1. The step dependencies at the overall level

Our method is composed of four steps for obtaining the specification (cf. Fig. 1): the **informal description** of the system to be specified, the **concurrent activity** description, the **sequential component** descriptions by an automaton, the **data type** specifications. Two validation steps may also be associated to the specification steps but they are not detailed here. Each step is described below with a proper agenda and is briefly described. A more complete presentation of our method may be found in [25] and the whole application to the case study in [24].

Step 1: Informal description (Fig. 2)

step	expression / schema	validation conditions
1.1: system functionalities	`F`$_i$`: text`	◦ no redundancy
1.2: system constraints	`C`$_i$`: text`	◦ consistency
1.3: system data	`D`$_i$`: sort`	

Fig. 2. Informal description agenda

The aim of this first step is to sketch out the system characteristics.

Step 1.1: Description of the system functionalities. In this substep all possible operations are inventoried, given a name (F_i in Fig. 2) and described. For instance, in our example, some operations are given by clauses 6 and 7:

– at the Control Port In: the reception of a command message (`in_cmde`)
– at the Data Ports In: the reception of a data message (`in_data`).

Step 1.2: System constraints description. The system constraints relative to orders, operations ordering, size limits, ... are expressed here and should be consistent, *i.e.* without contradiction.

Step 1.3: System data description. The point of view is very abstract here. The system data are given a name (and a sort name).

The transit node data are a list of open ports numbers (clause 6a), a list of routes (clauses 6b and 7a) and a list of faulty messages (clauses 4, 6c, 7a and 9).

Step 2: Concurrent activity

In this step the components that are executed in parallel are identified. Each one is modeled by a process. The process decomposition into subprocesses is inspired by the constraint-oriented and the resource-oriented specification styles [32, 33]. For each process there is a control part (dynamic part). Moreover, a data type (static part) may be associated with sequential processes and a variable of this type used as a formal parameter for the process. This unifies with the object-oriented encapsulation concept used later on. This step is decomposed in the following way: 2.1 communications, 2.2 decomposition and distribution, 2.3 parallel composition. As shown in Fig. 3, the substeps 2.2 and 2.3 may be iterated.

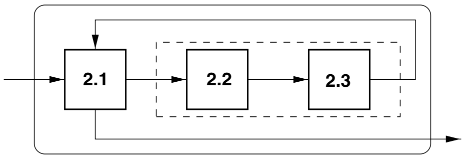


Fig. 3. Step dependencies for the concurrent activity

Step 2.1: Communications.

step 2.1.1: communication ports and data. The components interactions are modeled by communications on formal ports that represent the component services. Both communication ports and data are given by the informal description (step 1) or a previous decomposition (step 2).

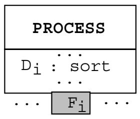
step	expression / schema	validation conditions
2.1.1: communication ports and data		∘ no omission: the $\mathtt{F}_i$ and $\mathtt{D}_i$ from 1.1 and 1.3 are all taken into account
2.1.2: communications typing	$\mathtt{F}_i$: $?\mathtt{x}_j:\mathtt{s}_j$ $!\mathtt{x}_k:\mathtt{s}_k$	∘ no omission: the $\mathtt{F}_i$ from 1.1 are all taken into account ∘ emission sorts are *available*

Fig. 4. Communications agenda

step 2.1.2: communications typing. The data that are either emitted (denoted by !x:T) or received (denoted by ?x:T) on the ports are typed. The communication typing is used to describe the processes interface, and also to specify the static part.

A validation criteria is to make sure that the emission sorts of a process are "available". A sort `s` is *available* for a process when:

– either it is directly available, that is predefined and imported, defined within the process, or received by the process
– or there exists an imported operation `f:` $d^* \to s$ such that all sorts in d^* are available. Since the data specification is achieved at step 4 this criteria validation may not be completed at this level.

We use a type `Msg` as an abstraction to represent the different messages in the transit node:

– reception of a command message: `in_cmde : ?m:Msg`
– reception of a data message: `in_data : !id:PortNumber ?m:Msg ?r: RouteNumber`
– emission of a list of faulty messages: `out_cmde : !l:List[Msg]`
– emission of a data message: `out_data : !m:Msg`

Step 2.2: Decomposition and distribution. The process decomposition into subprocesses is done using pieces of information (constraints) from the informal description (or a previous decomposition), using the data and/or functionalities. The decomposition may be done in such a way that already specified components may be reused.

Clause 1 leads to consider four components in the transit node: the Control Port In (CPI), the Data Ports In (DPI), the Control Port Out (CPO) and the Data Ports Out (DPO). The CPI manages the declared routes and the open ports (clause 6). The DPI needs access to information about which ports are related to which routes (clause 2a). Given the pieces of information collected from the preceding steps, the system may be represented as in Fig. 6 (see also Fig. 8).

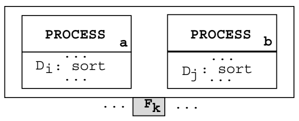
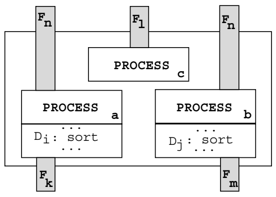
step	expression / schema	validation conditions
2.2.1: data distribution		◦ all data should be distributed in the subprocesses (cf. 2.1.1)
2.2.2: functionalities distribution	 Fn Fl Fn PROCESS c PROCESS a PROCESS b Di : sort Dj : sort Fk Fm	◦ functionalities and related data ◦ all functionalities should be distributed in the subprocesses (cf. 2.1.1)

Fig. 5. Decomposition and distribution agenda

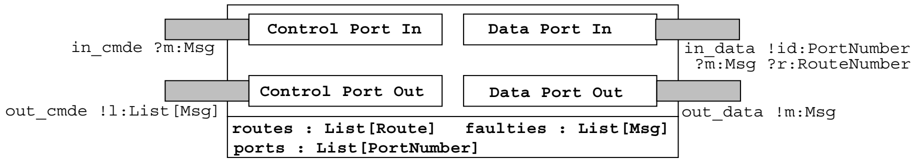


Fig. 6. Transit node external view (from Step 2.2)

Step 2.3: Parallel composition (Fig. 7). Processes composition often follows already known patterns, such as the synchronization schemata or the architectural styles given in [21, 18, 31]. Therefore we suggest to use a library of "composition schemata" that may be extended as needed. The specification of the subprocesses parallel composition may be derived from the set of the composition schemata using language specific features such as LOTOS operators [25] or SDL block structure and channels [24].

The process composition may lead to create new (internal) communications and possibly new data that have to be specified. Let us note that the process parallel composition is a way to express some constraints between the processes. Thus, clause 2b leads to a constraint on the parallel composition between the different ports. In order to take this into account, the DPI (resp. DPO) should be rather composed by interleaving than by synchronization.

Faulty messages. They are saved in the CPO collection (clause 9). They are (clause 10) either incorrect command messages (`wrong_cmde`, received by the CPI), or data messages with an unknown route (`wrong_route`, received by a DPI), or obsolete messages (`timeout`, from a DPO).

Information on routes. The DPI needs information on the transit node routes (whether a route is declared, and what are its associated ports). These pieces of information are held by the CPI, and will be transmitted to the DPI through

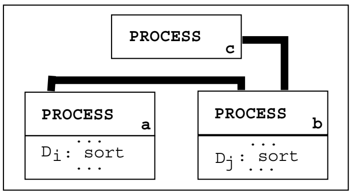
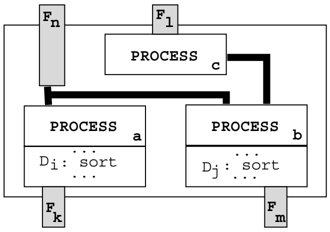
step	expression / schema	validation conditions
2.3.1: composition schema choice	 PROCESS c PROCESS a Di : sort PROCESS b Dj : sort	
2.3.2: schema application (cf. steps 2.2 and 2.3.1)	 Fn Fl PROCESS c PROCESS a Di : sort PROCESS b Dj : sort Fk Fm	○ relations between the process constraints and the constraints obtained through the subprocesses composition

Fig. 7. Parallel composition agenda

question/answer communications between the DPI and the CPI (`ask_route` and `reply_route`).

Message routing. When the data message route is correct, the message is routed (`correct` communication) by the DPI to one of the corresponding DPOs (clause 7a).

New ports. When the CPI receives the *Add-Data-Port-In-&-Out* command, it creates the corresponding ports (clause 6). In our modelization, this is taken into account by the fact that the Data Ports are enabled (`enable` communication) by the CPI on reception of this command.

New data. New data may arise from decomposition (or recomposition). Here, the DPOs are serialized (clause 7b) and have a buffer for messages. The Data Ports have an identifier used in enabling and routing communications.

Step 2 was iterated until obtaining the Fig. 8 schema. In the sequel, we shall focus on the DPI communication typing which is the following:

```
correct : !ident:PortNumber !m:Msg       ask_route : !r:RouteNumber
wrong_route : !m:Msg                     enable : !ident:PortNumber
reply_route : !r:RouteNumber ?l:List[PortNumber]
in_data : !id:PortNumber ?m:Msg ?r:RouteNumber
```

Step 3: Sequential components (Fig. 9)

Each sequential component is described by a guarded finite state automaton.

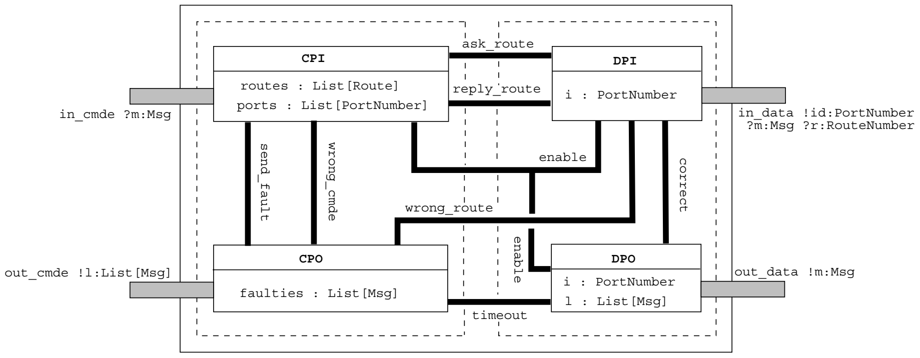


Fig. 8. Transit node internal view (from Step 2.3)

Steps 3.1 to 3.4: Conditions. The ports are put in four disjoint sets depending on whether they modify (C and CC) or not (O and OC) the state of the process and whether they are conditioned (CC and OC) or not (C and O). The names stand for *Constructor* (C), *Conditioned Constructor* (CC), *Observer* (O) and *Conditioned Observer* (OC).

The term "condition" refers to preconditions required for a communication to take place, and also to conditions that affect the behaviour when a communication takes place.

It should be checked that all conditions mentioned in step 1.2 are taken into account. However, some of them will be taken into account when dealing with the parallel composition of processes (step 2.3).

Applying the steps 3.1 and 3.2 to the DPI leads to identify the following conditions: `enabled` (the port is enabled), `received` (a message is received and not yet routed), `requested` (routing information was requested), `answered` (the answer on routing information is received), and `routeErr`[2] (routing error).

For instance, the `wrong_route` operation in CC has the following conditions: `enabled` $\wedge$ `received` $\wedge$ `requested` $\wedge$ `answered` $\wedge$ $\neg$`routeErr`.

Relationships between conditions are expressed by formulas ($\vdash \phi_i(\mathrm{C}_j)$). The relationship formula have to be consistent and may lead to simplify some conditions (possibly eliminating some).

In the DPI example, we have: `answered` $\Rightarrow$ `requested`, `requested` $\Rightarrow$ `received`, `received` $\Rightarrow$ `enabled`. This is consistent and leads to: `answered` $\wedge$ $\neg$`routeErr` when applied to the condition on `wrong_route`.

Steps 3.5 to 3.7: States retrieval. Whether a process may perform a service (through a communication on a port) depends on which abstract state the process is in. The states are thus retrieved by composition of the communications conditions (these conditions were identified in steps 3.2 to 3.4, and a truth table is constructed in 3.5). The formulas ($\phi_i(\mathrm{C}_j)$) expressing relationships between

[2] The `routeErr` condition bears on the value of the variable l of `reply_route` after the communication took place (see the communication typing at the end of step 2).

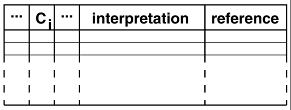
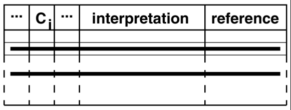
step	**expression / schema**	**validation conditions**
3.1: obtaining ports of O, OC, C, CC	`O, OC, C, CC`	∘ disjoint sets
3.2: conditions on ports of OC or CC category: precondition or behaviour	$\mathtt{F}_i$: $\mathtt{C}_j\mathtt{(category)}$	∘ 1.2 (cf also 2.3)
3.3: relationships between conditions	$\vdash \phi_i(\mathtt{C}_j)$	$\vdash_\lambda$ consistency: $\vdash \wedge_i \phi_i(\mathrm{C}_j)$
3.4: simplification of conditions		$\vdash_\lambda$ simplifications
3.5: creating the conditions table	 ... C_i ... interpretation reference	
3.6: elimination of impossible cases	 ... C_i ... interpretation reference	$\vdash_\lambda \phi_i(\mathrm{C}_j)$ (3.3)
3.7: states	$\mathcal{E}_i =< ..., v(\mathtt{C}_j), ... >$ $v(\mathtt{C}_j) \in \{T, F\}$	
3.8: operations preconditions	$\mathcal{P}_k =< ..., v(\mathtt{C}_j), ... >$ $v(\mathtt{C}_j) \in \{T, F, \forall\}$	$\vdash_\lambda$ consistency of preconditions w.r.t. $\phi_i(\mathrm{C}_j)$ $\vdash$ correction w.r.t. 3.2
3.9: operations postconditions	$\mathcal{Q}_k =< ..., \Phi_i(\mathtt{C'}_j), ... >$	C' : C + new conditions
3.10: relationships between conditions	$\vdash \phi_i(\mathtt{C'}_j)$	$\vdash_\lambda$ consistency: $\vdash \wedge_i \phi_i(\mathrm{C'}_j)$ $\vdash_\lambda$ consistency of postconditions w.r.t. $\phi_i(\mathrm{C'}_j)$
3.11: computing the transitions	$\mathcal{T} = f(\mathcal{E}, \mathcal{P}, \mathcal{Q})$	
3.12: choice of an initial state from possible (O_i) and impossible ($\overline{O_j}$) operations	$\mathcal{E}_{init}$	$\vdash_\lambda$ consistency of $\wedge_i \mathcal{P}_{O_i} \wedge_j \neg \mathcal{P}_{\overline{O_j}}$ $\vdash_\lambda$ only one initial state
3.13: automaton simplifications		$\vdash$ equivalences
3.14: translating the automaton to the target language		$\vdash$ automaton / specification
3.15: simplifying the specification		$\vdash$ correct simplifications

Fig. 9. Sequential components agenda

these conditions are used to eliminate incoherent states (3.6). Table 1 gives the DPI coherent states.

Table 1. State conditions table for the Data Port In

enabled	received	requested	answered	routeERR	state
T	T	T	T	T	IR (Incorrect Route)
T	T	T	T	F	CR (Correct Route)
T	T	T	F	F	WA (Waiting for Answer)
T	T	F	F	F	RfR (Ready for Request)
T	F	F	F	F	RfI (Ready for Input)
F	F	F	F	F	NA (Not Authorized)

Steps 3.8 to 3.11: Transitions retrieval. To retrieve the transitions, we shall define each operation in terms of possible source states (preconditions) and corresponding target states (postconditions). Therefore, preconditions ($\mathcal{P}$, 3.8) and postconditions ($\mathcal{Q}$, 3.9) are expressed in terms of the conditions values, respectively before and after the communications take place. The case where the condition value is not relevant is denoted by $\forall$, and $=$ denotes the case where the value is not modified after the communication. Verifications and simplifications may be achieved on both preconditions and postconditions [25].

Examples of preconditions and postconditions for some DPI operations are given below with the following notation: en for `enabled`, rec for `received`, req for `requested`, rep for `answered`, and `rerr` for routeErr.

`ask_route`	en	rec	req	rep	rerr
$\mathcal{P}$	T	T	$\forall$	F	$\forall$
$\mathcal{Q}$	=	=	T	=	=

`reply_route`	en	rec	req	rep	rerr
$\mathcal{P}$	T	T	T	F	$\forall$
$\mathcal{Q}$	=	=	=	T	l=[]

There are generally *critical cases* [25] and some postconditions may not be expressed using only state conditions. It is thus necessary to use new conditions to check whether the process data type is in a critical state. Informally, critical state conditions act as transition guards to avoid infinite state automata.

Operationally to retrieve the transitions, and for each operation:
- start from a given state e (a condition boolean tuple)
- if this tuple yields the operation precondition, find the tuple for the corresponding postcondition and the associated state f
- there is therefore a transition from e to f
- start over with another state.

Some improvements of this operational method are given in [25]. This automatic method leads to deal with cases that might not have been detected otherwise, as the critical cases.

Step 3.12: Initial state. In order to determine the *initial state*, it is necessary to identify the services (associated to some ports) the process should give or not in that state (constraints). It is a requirement based choice. The potential initial states are found from the ports preconditions and the state table. If no initial state is found, this means that the specifier gave inconsistent constraints for it. In order to be able to generate code, a single initial state is needed. When several potential initial states are found, it possible to choose one of them arbitrarily or by adding some constraint on the services. The DPI automaton is given in Fig. 10.

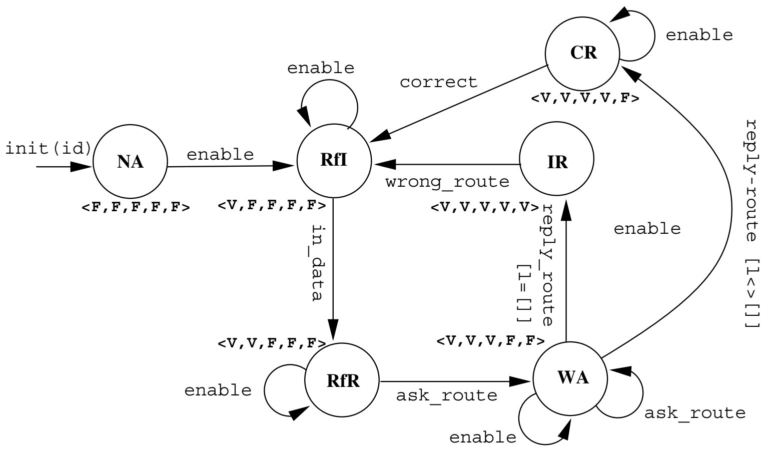


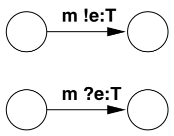
Fig. 10. Data Port In automaton

Steps 3.13 to 3.15: Simplifications and translation. It is possible to translate the automaton to various specification languages by applying translation schemata. This technique was applied to LOTOS [25] and to SDL [24]. Whenever the translation is not optimal, some simplifications may possibly be applied [25, 24]. The automaton may also be simplified before the translation, for instance by hierarchical grouping of states achieved using the conditions [24].

Step 4: Data types

The last step is the specification of the abstract data types associated to each process, and of the data types used within the processes. As regards the data types associated to each process, the work achieved in the preceding steps yields most of the signature (cf. [25] from a description of the automatic processing). Thus, the operations names and profiles are retrieved automatically from the communication typing, and from the conditions identified upon building the automata. Let us note that with each communication m, one or several algebraic operations may be associated (Fig. 11).

Some additional operations may be needed to express the axioms. Most of the axioms are written in a "constructive" style which requires to identify the generators. [3] describes a method to retrieve the data type associated to an automaton and to compute a minimal set of operations necessary to reach all states.



for emissions:
  m-c : DPI → DPI
  m-o : DPI → T
for receptions:
  m-c : DPI × T → DPI
  m-o : DPI → T (optional)

Fig. 11. Correspondence between communications and algebraic operations

In our example, this set is `init, enable, in_data, ask_route, reply_route`. [3] uses Ω-derivation [6] to write the axioms (conditional equations). In order to extract the axioms describing the properties of the operation `op(dpi:DPI)`, the states where this operation is allowed should be identified together with the generators to reach these states, thus yielding the premises and the axioms left hand sides.

Part of this automatic processing is shown here for the axioms of the `correct_c` operation, the internal operation associated with the `correct` transition. The premises express conditions on the source states and on the `l` variable.

```
% correct-c : DPI -> DPI
CR(dpi) => correct-c(enable-c(dpi)) = correct-c(dpi)
WA(dpi) /\ not(l=[]) => correct-c(reply-route-c(ask-route-c(dpi),l)) =
    correct-c(reply-route-c(dpi,l))
WA(dpi) /\ not(l=[]) => correct-c(reply-route-c(enable-c(dpi),l)) =
    correct-c(reply-route-c(dpi,l))
RfR(dpi) /\ not(l=[]) =>
    correct-c(reply-route-c(ask-route-c(enable-c(dpi)),l)) =
        correct-c(reply-route-c(ask-route-c(dpi),l))
RfI(dpi) /\ not(l=[]) =>
    correct-c(reply-route-c(ask-route-c(in-data-c(dpi,m,r)),l)) = dpi
```

The algebraic specification may then be used for proving properties needed for the specification verification and validation.

4 Code Generation

Once we get a formal specification it is not necessarily executable. Often the dynamic part is executable because it is based on operational models (state transition diagrams). This is not always true for the static part (algebraic abstract data type).

We will illustrate assisted code generation in Java, however the method is suitable for other object-oriented languages.

The general method is depicted on Fig. 12 and is split in two parts: the static part (on the right of Fig. 12) and the dynamic part (on the left of Fig. 12).

4.1 Static Part Generation

Java classes are generated for each abstract data type in the specification, and this process is achieved by four intermediate steps (cf. Fig. 12). The translations

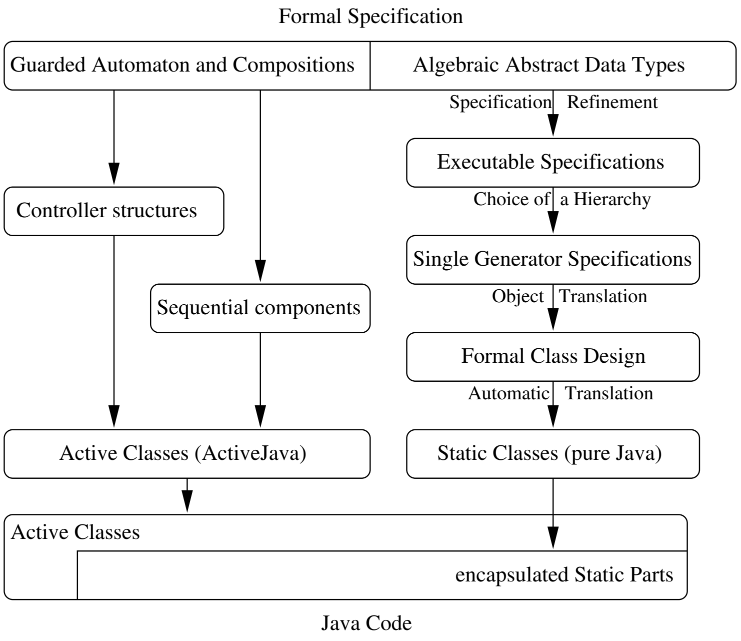

Formal Specification
Guarded Automaton and Compositions
Algebraic Abstract Data Types
Specification Refinement
Executable Specifications
Choice of a Hierarchy
Controller structures
Single Generator Specifications
Sequential components
Object Translation
Formal Class Design
Automatic Translation
Active Classes (ActiveJava)
Static Classes (pure Java)
Active Classes
encapsulated Static Parts
Java Code

Fig. 12. Object code generation scheme

are partly automatic, for instance to get a simpler or a more efficient result may require some specifier (or programmer) interaction. The first step is to obtain an executable specification (possibly through refinements). Then, code generation is decomposed into (i) the choice of a hierarchy for representing the specification generators, (ii) the translation into formal classes (i.e. abstractions of classes in object-oriented languages), from which (iii) a generation in a given language (e.g. Java) may be done.

Executable Specification. The "constructive" style adopted for the specifications associated with the automatons is likely to yield executable specifications (e.g. through rewriting, where tools, e.g. [14], may be used to check the convergence). However, other specification modules may be introduced (e.g. for the data managed by the processes) with other specification styles (e.g. observational style). A refinement process (abstract implementation [12]) is then needed to add elements for executability such as algorithmic choices, etc.

Single Generator Specifications. In object-oriented languages, classes have a single generation operation called for instance "new" (or the class name), while algebraic specifications allow several generators. The problem addressed here is how to represent these generators within classes, or more precisely how to transform (e.g. by abstract implementation) the original algebraic specifications into single generator specifications from which classes may be derived. We propose several solutions to this issue. A first solution is to associate to the several generators of the algebraic specification a single generator with a "switch" (to each original generator), we refer to this solution as the "flat organization". Another solution is to use the associated class as an interface to subclasses, where each subclass is associated to one generator of the original specification, this will be denoted as the "two level hierarchy organization". Then, of course, it is possible to mix these two solutions as appropriate.

Several frameworks are available for abstract implementation [12], a first sketch is to follow the general framework of Hoare's representation [19]. It consists into defining an abstraction function, to prove it is an onto function and to prove the implementation of operations. These aspects are not detailed here.

In the following, we present the alternative organizations for single generator specifications. When the abstract data type has only one generator we directly apply the simple representation described below to get a class.

Flat Organization. In this organization, a specification module with several generators is transformed into a single generator specification module with a "switch" to each original generator. For example, in the `DPI` specification module, the generators are `init, enable, in_data, ask_route, reply_route`. We define `SwitchDPI = {init, enable, in_data, ask_route, reply_route}` and the single generator `newSDPI` (SDPI stands for Switch DPI) with the profile `newSDPI : Switch PortNumber Msg RouteNumber List SDPI -> SDPI` (note that this profile may be easily computed from the DPI generators profiles). The abstraction function `Abs` is defined as usual, e.g.:
`Abs(newSDPI(reply_route, Bport, Bmsg, Broute, Z, T)) == reply_route_c(T, Z)`...
Terms beginning by `B` are don't care values. We also introduce selectors associated to relevant arguments occurring in the single generator, e.g.:
`switch(newSDPI(S, X, Y, R, Z, T)) = S`
`(S = reply_route` $\wedge$ `WA(T)) => selRoutes(newSDPI(S, X, Y, R, Z, T)) = Z`...
The axioms are then transformed within this framework to complete the specification.

Two Level Hierarchy Organization. In this approach, several specification modules are associated with the original specification module: one module that is just an interface to modules that introduce (each) one of the original generators together with the appropriate subsort. Clearly, this approach may yield semantics issues (depending on the framework adopted), and may not be as practical and straightforward as the previous one. However, in some cases the specification style may be more legible.

Mixed Organization. Of course between these two previous extrema there are many other ways to transform the type depending of the chosen hierarchy. We studied in [2] how to get a better hierarchy and we presented a general process for it. However some important problems remain: metrics to define a best hierarchy and problems linked with inheritance of properties.

In case of abstract data types with less than five generators, the flat organization is acceptable but with more complex ones this will not be the case.

Another way to solve this problem is to introduce a kind of inheritance or subsort (OBJ subsort) in the method. This problem is known to be difficult in itself and rather complex with concurrent systems.

Formal Class Design: The Model. This model [4] defines the notion of *formal class* as an abstraction of a concrete class in languages like C++, Eiffel, Java or Smalltalk. A formal class is an algebraic specification (as abstract data type) with an object-orientation. This general model is functional and unifies the major concepts of object-oriented programming. It can be used both to build formal specifications and to design a system. An abstract operational semantics [4] was given to this model using conditional term rewriting [10].

Figure 13 shows a formal class example associated to the SDPI specification module obtained with the flat organization.

```
                                FCDPI

field selectors
    switch    : FCDPI ⟶ Switch
    ident     : FCDPI ⟶ PortNumber
         requires: switch(Self) = init
    selRoutes : FCDPI ⟶ List
         requires: switch(Self) = reply_route ∧ WA(previous(Self))
         ...

methods
correct_c : FCDPI  ⟶  FCDPI
;; correct_c : internal operation associated to a correct route
        (switch(Self) = enable ∧ CR(previous(Self))) =>
        correct_c(Self) = correct_c(previous(Self))
        (switch(Self) = reply_route ∧ switch(previous(Self)) = ask_route
         ∧ WA(previous(Self)) ∧ is_empty(list(Self))) =>
         correct_c(Self) = correct_c(new(reply_route, Bport, Bmsg, Broute,
                                 selRoutes(Self), previous(previous(Self))))
        ...
```

Fig. 13. Formal Class FCDPI

The translation into (purely functional) Java code is straightforward. A formal class is translated to an interface (corresponding to the signature) and an implementation class. We use abstract methods and classes when needed (depending on the chosen organization). The structure is represented by private instance variables. Fields selectors are coded by a public accessor to the corresponding instance variable with a condition corresponding to the precondition.

A tool to generate Java classes is not available yet, but experimental tools have been done for Eiffel and Smalltalk.

Formal Class Design: A Simple Representation. The simple representation allows one to translate a single generator type into a formal class, denoted by `FCADT`. This generator will be the `newFCADT` instantiation function of the object model. We must identify selectors, *i.e.* operations $\texttt{sel}_i$ such that $\texttt{sel}_i\texttt{(new(X}_1\texttt{, ..., X}_n\texttt{)) = X}_i$. These field selectors yield the instance variables of the class. We assume that the specification (axioms and preconditions) has no variable named `Self`. A term is said to be in a receiver position if it appears at first position in an operation different from the generator. If a variable appears in a receiver position in the left conclusion term then it will be replaced by `Self`. In our model this variable denotes the object receiver. An important translation rule is to replace $\texttt{newSADT(e}_1\texttt{, ..., e}_n\texttt{)}$ by `V` with `V : FCADT`. This leads to a set of equations: $\texttt{sel}_i\texttt{(V) = e}_i$.

1. This rule is applied on every `newSADT` occurrence in a receiver position in the left conclusion term, where `V` is named `Self`. If $\texttt{e}_i$ is a variable then it is replaced by $\texttt{sel}_i\texttt{(Self)}$ in axioms. If $\texttt{e}_i$ is neither a variable nor a don't care term, the equation $\texttt{sel}_i\texttt{(Self) = e}_i$ is added to the axiom condition.
2. This rule is applied on all other occurrences in the left conclusion term with any variable other than `Self`.

This representation was processed over the single generator specification SDPI and the result is the above formal class (Fig. 13).

4.2 Dynamic Part Generation

This part deals with the code generation for the dynamic part in an object oriented language. The language we aim at is ActiveJava [1], a Java dialect (pre-processed into pure Java) based on ATOM [23].

The ActiveJava model defines abstract states, state predicates, methods activation conditions and state notification. Its main advantages are that: (i) its syntax is defined as a Java extension, (ii) it permits to model both inter and intra-object concurrency, and (iii) it supports both asynchronous and synchronous message passing. ActiveJava presents a good adequation between reusability of components (through separation of concerns into a dynamic and a static part) and expressivity.

The dynamic part generation is build upon (i) coding subsystems structuration and LOTOS-like communication semantics, (ii) coding sequential automata, and (iii) integrating dynamic and static parts.

Structuration and Communication Semantics. LOTOS communication semantics is more strict than the ActiveJava (object oriented) one. For this purpose and for better structuration matters, we choose to model each subsystem structuration with a *controller*. This approach is close to Coordinated Roles [22] and aims at the same properties: a better structuration and reusability of composition patterns.

The system specification is taken as a tree with sequential components at the leaves and controllers at the nodes where the LOTOS structuring mechanisms are encoded. The subtrees under a given controller will be called its *sons*. Structuration for the transit node is given in Fig. 14 where coordinators are drawn as square boxes and sequential components as round boxes. This is another representation of Fig. 8.

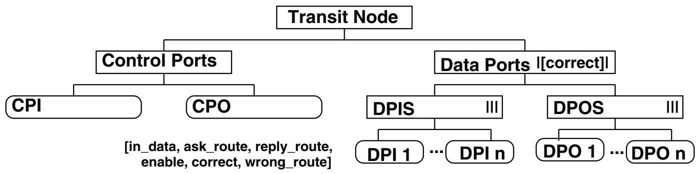


Fig. 14. Sketch of structuration and coordinators of the transit node

We have three main communication mechanisms for structuration: interleaving, synchronization and hidden synchronization.

Common mechanisms. The communication is achieved in three phases as shown in Fig. 15. In the *run phase*, controllers dispatch calls to a *run method* to their non waiting sons. Thus, in Fig. 15-1, C sends a `run` method to P but not to E. When these calls reach non controller components (i.e. the leaves, as P in Fig. 15-1) or controller with all sons blocked, then the second phase, *return phase* begins.

In this return phase, sons return the communications they are ready to get involved with in a *runnable list*: a list of tuples containing the communication name and its arguments, with values for emission arguments and a special indicator (_) for reception arguments. P returns `[(‘‘m’’,_,4),(‘‘n’’,3)]` to assess it is ready for a m or n communication (Fig. 15-2). The controller then computes a *common intersection* of the runnable lists and sends it up. Here, `n` from P does not match with anything from E whereas two `m` elements match to make the intersection that C sends upwards. Since some E and P runnable lists elements are in the common intersection, E and P are called *participants*. Elements with the same communication name have to match in the same way LOTOS offers match. Matching cases are given in Table. 2. All other cases mismatch.

The second phase ends when there is no intersection (this yields a blocking status) or at the root where a final intersection in computed. The controller where the second phase ends is called *temporary root*.

In the third phase (Fig. 15-3), the temporary root sends down the message corresponding to the final intersection it has previously computed. This message has to be unique, and non determinism (whether a received value has not been bound or there is communication non determinism) is solved by the temporary

Table 2. Matching elements and intersections

element 1	element 2	common intersection
("m",_:T)	("m",value:T)	("m",value:T)
("m",value:T)	("m",_:T)	("m",value:T)
("m",value:T)	("m",value:T)	("m",value:T) – same values
("m",_:T)	("m",_:T)	("m",_:T)

root controller [27]. Controllers send the message only to *participants* (both P and E for C) and then erase their table entry. Non participant sons are left waiting. To end, the temporary root controller relaunches the first phase by sending again the run method to its non waiting sons.

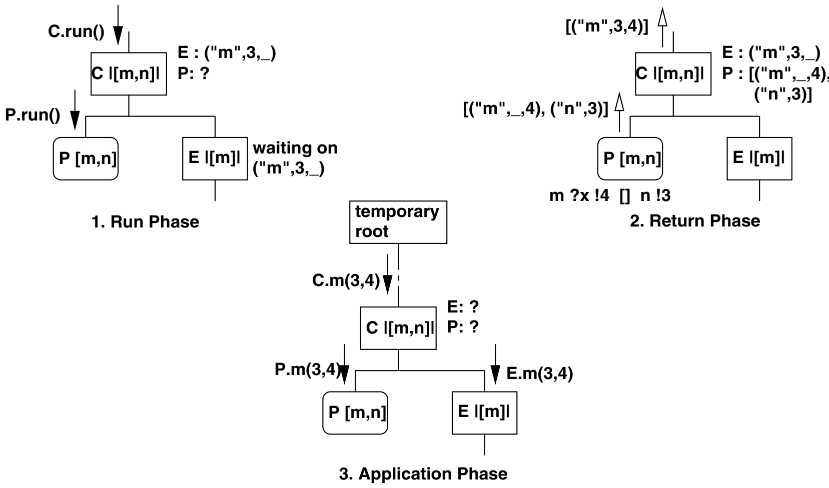


Fig. 15. A communication scheme example

Interleaving. As soon as a controller receives a non synchronized element in a runnable list, it transmits it up.

Synchronization. When two sons are composed in order to synchronize on a given method, their parent controller will transmit the corresponding element in runnable lists only if it has received this element in both sons runnable lists.

Hidden synchronization. In the return phase, when the runnable lists reach a node, elements referring to hidden messages are not returned upwards but are kept at this node level. When only hidden messages reach a controller which has to block them, this controller acts as the temporary root controller. If there are also non hidden messages, the controller chooses whether to transmit them upwards or to act as the temporary root (this choice simulates non determinism).

Coding the Automata in ActiveJava. The precondition and postcondition tables are used to code the automaton. But this has to be slightly modified to take into account `run` message receptions. The schema given in Fig. 16 is applied to each state.

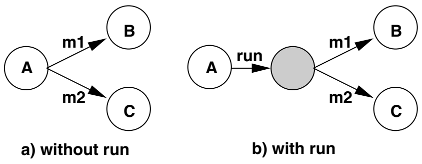


Fig. 16. Adding `run` in the automata

Operation preconditions are defined as “activation conditions” in ActiveJava. Optionally, condition variables may be set (postconditions) in the “post actions” methods of ActiveJava. In the class constructor, initial values for the conditions are set according to the automaton initial state. Fig. 17 illustrates this on a part of the DPI example.

```
active class DPI(CPI cpi, FC fc, DPO dpo) {
    boolean v_a, v_r, v_d, v_rep, v_rerr;
    PortNumberList received_l; ...

abstract state definitions {a as is_a(); ...}

activations conditions {
    reply_route(RouteNumber r, PortNumberList l)
        with reply_route_precondition(); ...}

synchronization actions {
    reply_route(RouteNumber r, PortNumberList l) with
        post_actions reply_route_postcondition(); ...}

implement synchronization {
    boolean is_a() {return v_a;}
    ...
    boolean reply_route_precondition() {
        return v_a==TRUE && v_r==TRUE && v_d==TRUE && v_rep==FALSE;}
    ...
    void reply_route_postcondition() {
        v_rep=TRUE; v_rerr=received_l.isEmpty();}
    ...
}}
```

Fig. 17. (Part of) Active Class for DPI

Integrating the Static and the Dynamic Parts The integration of the static and the dynamic part is done using encapsulation of a static class instance into the dynamic class with static methods called into dynamic methods bodies (Fig. 18). Observers are called in the run method to compute some of the run list elements arguments. Statics methods are also called in each corresponding dynamic method.

```
import StaticClass;

active class DynamicClass {
    StaticClass nested;
    < ActiveClass part >
    public methods {
        public DynamicClass( < arguments > ) {
            nested = new StaticClass( < arguments > );
            < dynamic initialization (initial state) >
        }
        public RunnableList run() {
        // uses nested.observers return values in runnable list
        }
        public void otherMethod( < arguments > ) {
            nested.otherMethod( < arguments > );
        }}}
```

Fig. 18. Integration of static and dynamic parts

5 Conclusion

While there are good motivations for the use of formal specifications in software development, the lack of methods may restrict it to "few experts". There are several points which may cause problems: the use of formal notation, the structure of the system, the proofs of properties and the code generation (or refinement for others) are some of the most important. In this paper, we address a specification method for systems where both concurrency and data types issues have to be taken into account. One important feature is the help provided to the user: help to build dynamic behaviours, help to decompose the system, help to extract the data types and help to generate code. Our method takes advantage of both the constraint and state oriented approaches that are used for LOTOS or SDL specifications. The system is described in terms of parallel components with well defined external interfaces (the gates and communication typing). The behaviour of the component is described by a sequential process associated with an internal data type. The study of the communications and their effect on this data type allows one to build, in a semi-automatic way, an automaton describing the process internal behaviour. The automaton is then translated into a specification language (LOTOS or SDL). The data type is extracted by a semi-automatic method from this automaton.

The components and the whole system may then be verified using common set of tools for transition systems [13] or algebraic specifications [14].

Our specification method is equipped with a prototype generation. Object-oriented languages are another major phenomenon in software engineering. One cannot ignore the qualities of such code, however writing such code may be a hard task. We choose to generate Java code but our method may be applied to other object oriented languages. This code generation is mainly automatic and modular. We plan to experiment code generation on other frameworks for concurrent object oriented programming such as [8].

One future research direction is the extension of this approach to other specification languages, like Raise [30] or Object-Z [28]. Other connected areas of research are about object-oriented analysis and design methods. We currently work on the use of UML [29] diagrams to improve system architecture and to validate the automaton behaviour (with communication diagrams for instance). Therefore, we plan to provide our specification model with inheritance, more complete communication (experimenting new controller semantics) and structuration mechanisms as in related models [9].

References

[1] Luis A. Álvarez, Juan M. Murillo, Fernando Sánchez, and Juan Hernández. ActiveJava, un modelo de programación concurrente orientado a objeto. In *III Jornadas de Ingenería del Software, Murcia, Spain*, 1998.

[2] Pascal André. *Méthodes formelles et à objets pour le développement du logiciel : Etudes et propositions*. PhD Thesis, Université de Rennes I (Institut de Recherche en Informatique de Nantes), Juillet 1995.

[3] Pascal André and Jean-Claude Royer. How To Easily Extract an Abstract Data Type From a Dynamic Description. Research Report 159, Institut de Recherche en Informatique de Nantes, September 1997.

[4] Pascal André, Dan Chiorean, and Jean-Claude Royer. The formal class model. In *Joint Modular Languages Conference*, pages 59–78, Ulm, Germany, 1994. GI, SIG and BCS.

[5] M. Bidoit, C. Choppy, and F. Voisin. Validation d'une spécification algébrique du "Transit-node" par prototypage et démonstration de théorèmes. Chapitre du Rapport final de l'Opération VTT, Validation et vérification de propriétés Temporelles et de Types de données (commune aux PRC PAOIA et C3), LaBRI, Bordeaux, 1994.

[6] Michel Bidoit. Types abstraits algébriques : spécifications structurées et présentations gracieuses. In *Colloque AFCET, Les mathématiques de l'informatique*, pages 347–357, Mars 1982.

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

[8] D. Caromel and J. Vayssière. A Java Framework for Seamless Sequential, Multi-threaded, and Distributed Programming. In *ACM Workshop "Java for High-*

Performance Network Computing", pages 141–150, Stanford University, Palo Alto, California, 1998.

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

[10] Nachum Dershowitz and Jean-Pierre Jouannaud. *Rewrite Systems*, volume B of *Handbook of Theoretical Computer Science*, chapter 6, pages 243–320. Elsevier, 1990. Jan Van Leeuwen, Editor.

[11] Jan Ellsberger, Dieter Hogrefe, and Amardeo Sarma. *SDL : Formal Object-oriented Language for Communicating Systems.* Prentice-Hall, 1997.

[12] M. Navarro F. Orejas and A. Sanchez. Implementation and behavioural equivalence: a survey. In M. Bidoit and C. Choppy (Eds.), editors, *Recent Trends in data Type Specification*, volume 655 of *Lecture Notes in Computer Science*, pages 93–125. Springer-Verlag, August 1993.

[13] Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Radu Mateescu, Laurent Mounier, and Mihaela Sighireanu. CADP: A Protocol Validation and Verification Toolbox. In *8th Conference on Computer-Aided Verification*, pages 437–440, New Brunswick, New Jersey, USA, 1996.

[14] S. Garland and J. Guttag. An overview of LP, the Larch Prover. In *Proc. of the Third International Conference on Rewriting Techniques and Applications*, volume 355 of *Lecture Notes in Computer Science*, pages 137–151. Springer-Verlag, 1989.

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

[16] 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.

[17] Maritta Heisel. Agendas – A Concept to Guide Software Development Activities. In R. N. Horspool, editor, *Proceedings Systems Implementation 2000*, pages 19–32. Chapman & Hall, 1998.

[18] Maritta Heisel and Nicole Lévy. Using LOTOS Patterns to Characterize Architectural Styles. In Michel Bidoit and Max Dauchet, editors, *TAPSOFT'97 (FASE'97)*, volume 1214 of *Lecture Notes in Computer Science*, pages 818–832, 1997.

[19] C.A.R. Hoare. Proof of Correctness of Data Representations. *Acta Informatica*, 1:271–281, 1972.

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

[21] Thomas Lambolais, Nicole Lévy, and Jeanine Souquières. Assistance au développement de spécifications de protocoles de communication. In *AFADL'97 Approches Formelles dans l'Assistance au Développement de Logiciel*, pages 73–84, 1997.

[22] 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, *Third International Conference, COORDINATION'99*, volume 1594 of *Lecture Notes in Computer Science*, Amsterdam, The Nederlands, April 1999. Springer-Verlag.

[23] M. Papathomas, J. Hernàndez, J. M. Murillo, and F. Sànchez. Inheritance and expressive power in concurrent object-oriented programming. In *LMO'97 Langages et Modèles à Objets*, pages 45–60, 1997.

[24] Pascal Poizat, Christine Choppy, and Jean-Claude Royer. Un support méthodologique pour la spécification de systèmes “mixtes”. Research Report 180, Institut de Recherche en Informatique de Nantes, Novembre 1998. /papers/rr180.ps.gz in Poizat's web page.

[25] Pascal Poizat, Christine Choppy, and Jean-Claude Royer. Une nouvelle méthode pour la spécification en LOTOS. Research Report 170, Institut de Recherche en Informatique de Nantes, Février 1998. /papers/rr170.ps.gz in Poizat's web page.

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

[27] Pascal Poizat, Christine Choppy, and Jean-Claude Royer. From Informal Requirements to Object Oriented Code using Structured Concurrent Sequential Communicating Automata. Research Report, Institut de Recherche en Informatique de Nantes, 1999.

[28] Graeme Smith. A Fully-Abstract Semantics of Classes for Object-Z. *Formal Aspects of Computing*, 7(E):30–65, 1995.

[29] Rational Software and al. Unified Modeling Language, Version 1.1. Technical report, Rational Software Corp, http://www.rational.com/uml, September 1997.

[30] The Raise Method Group. *The RAISE Development Method*. The Practitioner Series. Prentice-Hall, 1995.

[31] K. Turner. Relating architecture and specification. *Computer Networks and ISDN Systems*, 29(4):437–456, 1997.

[32] Kenneth J. Turner, editor. *Using Formal Description Techniques, An introduction to Estelle, Lotos and SDL*. Wiley, 1993.

[33] C.A. Vissers, G. Scollo, M. Van Sinderen, and E. Brinksma. Specification styles in distributed systems design and verification. *Theoretical Computer Science*, (89):179–206, 1991.