Concurrency and Data Types: a Specification Method
An Example with LOTOS

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
[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 present a specification method that takes into account both the specification of concurrent activity and the specification of the data types involved. It is applied here to LOTOS specification, but it may be used for other formalisms. Our method is both constraint oriented (for the processes decomposition into parallel subprocesses) and state oriented (for the design of the sequential components). This latter aspect is based on (i) the design of an automaton from the external behaviour description, (ii) the generation of a LOTOS specification associated with this automaton. We illustrate our method through a simple example, a hospital.

Keywords: specification, method, LOTOS, constraint oriented, state oriented, automaton

1 Introduction

While the importance of using formal specifications in software development is widely accepted, there is still a need for methods to help applying them. In this respect, we would like to quote [3]: "a formalism does not provide a method by fiat". It seems to be even more necessary to provide a method for specifications that involve both the specification of concurrent activity and the specification of the data types involved in this activity. While there is a need for describing both the dynamic (process control) and the static aspects (data type), the specification work is likely to be more complex. These two aspects are available in the LOTOS language [17, 5, 20] that was developed for the specification of distributed systems. The dynamic part is based on process algebra, like CCS [21] or CSP [15], and the static part is based on abstract data types, as in the algebraic specification language ACT ONE [7].

* This work was achieved while this author was at IRIN.

Among the few methods related to LOTOS development, the main tendencies are the constraint oriented approach and the state oriented approach. In the constraint oriented approach the overall system is split into concurrent subprocesses. This approach is well suited to earlier phases of the specification process where the emphasis is put on the requirement constraints. In the state oriented approach, entities of the system are modelled as processes with internal states. The transitions between internal states express internal actions or communications between processes. This approach is well suited when links between the dynamic and static parts need to be expressed.

The method we present in this paper is applied to LOTOS specification and improves existing ones in three ways. First it is a general method that mixes the two main approaches, and produces a modular description with a dynamic behaviour and its associated data type. Secondly, the dynamic behaviour extraction is based on a guarded finite state machine. This state machine is progressively and rigorously built from the requirements. Type information and operation preconditions are used to retrieve states and transitions. The dynamic behaviour specification is computed from the automaton using some patterns. The last improvement is the assisted computation of the functional part. Our method reuses a technique which allows one to obtain an abstract data type from an automaton [1]. This technique extracts the signature and generators from the finite state machine. Furthermore, the automaton is used to generate parts of the axioms.

The paper is structured as follows. We briefly present the LOTOS language in Section 2. Section 3 discusses and compares existing methods for LOTOS specifications. Section 4 presents the general process of our method, for the dynamic part and the static part of the specification. In the conclusion we summarize the main points of our method and we mention our work on applying it to other formalisms.

2 A Short Presentation of LOTOS

LOTOS [17] (Language of Temporal Ordering Specification) is a Formal Description Technique developed within ISO (International Organization for Standardization). It is a language devoted to the specification of distributed and open systems, especially to protocols and telecommunications.

The most important characteristics of LOTOS are: complementary formalisms for data and control, the semantics based on process algebra and abstract data types, executability and modularity.

The language is usually presented in two views: basic LOTOS and full LOTOS.

2.1 Basic LOTOS

Basic LOTOS is the subset of the language where process synchronization is achieved. Following CCS [21] and CSP [15], a behaviour expression describes the

process control. Synchronization and communication are possible *via rendez-vous* without sharing memory.

A system is described as a set of processes which are entities able to perform actions and to interact with other processes. The environment of a process consists of other processes. It is possible to build a hierarchy of processes.

A behaviour expression describes the behaviour of a process, it specifies which actions are possible. Actions are atomic units, and are supported by gates. A predefined set of operators is used to combine actions and behaviour expressions to form new behaviour expressions. The following operators are described to make the paper self-contained.

Notation: a lowercase name is a gate, an uppercase name represents a behaviour expression.

The prefix operator: `a;A` denotes a rendez-vous on gate `a` followed by the behaviour `A`. It expresses sequential composition of actions.

The choice operator: `A [] B` denotes a choice between two behaviours. The selective parallel operator: `|[ ]|` is used to specify synchronization on a list of gates between two behaviours.
`A |[`g_1`,...,`g_n`]| B` denotes that `A` and `B` synchronize on common gates g_i. To the contrary, actions non equal to `a` or `b` are interleaved (executed asynchronously) in `A` and `B`. This operator is one of the most important. It has two variants:

- || full synchronization on all gates of `A` and `B`
- ||| interleaving, no synchronization at all (but for termination).

The `hide` operator hides a gate in a behaviour to disallow environment synchronization on that gate. For instance, `hide a in (A |[a,b]| B)` may be synchronized on `b` but not on `a`.

A special operator `stop` represents inaction. It cannot offer synchronization and it does nothing, it is a blocked process.

2.2 Full LOTOS

Full LOTOS copes with data specifications, data transfer during synchronizations and process parameterization over specified data domains. Data structures and values are described algebraically as in ACT ONE [7]. Variables are typed (`X:NAT, Y:BOOL, ...`), and the language is typed. Types are predefined and imported from a library or defined by a specification:

```
type ... endtype.
```

In full LOTOS, communications on gates are enriched by value passing. For example, in the following expression: `exchange !Put ?Get:NAT`, the `Put` value is sent during the rendez-vous on the `exchange` gate while a natural value is received in the `Get` variable. A process may accept arguments:

```
process STACK[push,pop] (S : Stack) : noexit :=
...
endproc
```

This process may be called or instantiated as in `STACK[push,pop](aStack)`. LOTOS allows the instantiation of processes in behaviours (e.g. in recursive processes). The `noexit` qualifier specifies that the process is non terminating (respectively `exit` denotes a process which may terminate). A behaviour expression can be preceded by a guard, this allows one to write some kind of conditional expression:

```
  [X >= 0] -> get; stop
[]
  [X < 0] -> put; stop
```

In the following specification example, the processes P, Q and R are composed in parallel:

```
specification FOO[...] : noexit
  behavior
        (P[a,b] ||| Q[b,c]) || R[c]
  where
        process P[...]
        ...
        endproc
  ...
endspec
```

A LOTOS specification contains processes and data type definitions and a behaviour expression.

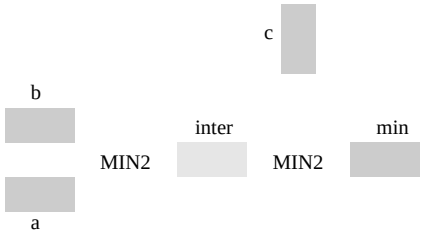
3 Existing Methods for LOTOS Specification

LOTOS is expressive enough to allow a wide range of specification styles, and it is possible to specify a given problem in various ways. Guidelines for specifications are therefore needed, but there are few papers that refer to methods for LOTOS specification. Among the mentioned methods, there are two main approaches, the constraint oriented approach and the state oriented one, and in both the emphasis is put on the dynamic part of the specification.

3.1 Dynamic Part

The *constraint oriented approach* is a "divide and conquer" type method. In general, the specifier uses the following steps: (i) structure the specification into several subprocesses (possibly recursive), (ii) specify each subprocess, and (iii) apply parallel composition of the subprocesses. The resulting process satisfies a constraint set that is equal to the union of the subprocesses constraint sets plus some synchronization constraints between them [5, 20]. Figure 1 shows how this approach is applied to the `MIN3` example where two `MIN2` subprocesses are composed in parallel (with a hidden communication on gate `inter`).

The *state oriented approach* is a hybrid approach where objects are processes managing data types [19]. For each state, different sets of communications may




```
process MIN3[a,b,c,min] : noexit :=
  hide inter in
     ( MIN2[a,b,inter]
  |[inter]|
     MIN2[c,inter,min] )
endproc
```

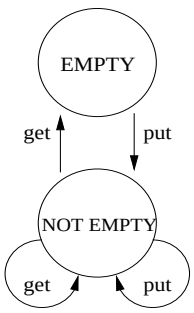
```
process MIN2[in1,in2,out] : noexit :=
  in1 ?x1:NAT; in2 ?x2:NAT; out !min(x1,x2); stop
[]
  in2 ?x2:NAT; in1 ?x1:NAT; out !min(x1,x2); stop
endproc
```

Fig. 1. An example using the constraint oriented approach

happen depending on the process internal state. The transitions represent either communications between processes or internal transitions, and generally have an effect on their data types. See Figure 2 for an example of this approach applied to an unbounded buffer.

A related method dealing with embedded systems is presented in [6]. It is based on communicating concurrent objects, every object being a process with a parameter that stands for its internal (private) state. These processes use cycles: their behaviours achieve a given service followed by a recursive call.

These approaches are compared in Section 3.4.




```
process BUFFER[put,get] (b:Buffer) : noexit :=
   EMPTY[put,get] (b)
endproc
process EMPTY[put,get] (b:Buffer) : noexit :=
   put ?x:T; NOT-EMPTY[put,get] (cons(b,x))
endproc
process NOT-EMPTY[put,get] (b:Buffer) : noexit :=
   [nb(b)=1] -> get !car(b); EMPTY[put,get] (cdr(b))
[] [nb(b)>1] -> get !car(b); NOT-EMPTY[put,get] (cdr(b))
[] put ?x:T; NOT-EMPTY[put,get] (cons(b,x))
endproc
```

Fig. 2. An example using the state oriented approach

3.2 Static Part

Concerning the static part, [26, 11] propose to use a *constructive approach* and [11] explains it is better to deal with the data part after the dynamic part; but there is no method really for the creation of the static part (algebraic data types), and in any case it is not related to the dynamic part.

3.3 A Mixed Method

We studied the approach followed by [11] to specify a matrix switch problem. In the following, we describe our understanding of their approach. This will be a starting point for our method that adds and modifies some points of this approach.

Informal description. The system that is to be specified is described in natural language.

Parallel architecture. Let us first remark that we prefer "concurrent activity" since parallel and architecture are, usually, hardware referring terms. Here the components executed in parallel are described. Each one is modelized by a LOTOS process. The interactions between components are modeled by formal gates. The communications between the components are then determined and the processes are linked together using LOTOS parallel composition operators. If needed, the operations that are not relevant at this level of abstraction are hidden (using the `hide` operator). This step can be recursively applied to the subsystems in order to obtain purely sequential components.

Sequential components. Each sequential component can be described using a finite state automaton. This automaton is then translated into a LOTOS behaviour expression.

Data types. The sorts and operations that appeared in earlier phases are put together. The corresponding axioms are then obtained using a constructive approach. In [11], it is mentioned that this can lead to over-specification and loss of generality.

Let us note that [11] does not suggest methods for the automaton construction, and for the data type specification (apart from the recommended constructive approach). In Section 4 we extend this approach to provide these features.

3.4 Approaches Comparison

The constraint oriented approaches are well suited to service specification and make the decomposition of processes into sub-components easier.

The authors of [11] think the state oriented approach can lead to unstructured specifications. We think it allows to easily understand the internal behaviour of

the processes and agree with [26] that it helps producing more efficient implementations (in purely sequential languages).

As noted by [5, 9, 10], there are many specification styles in LOTOS and not yet a way to give a quality measure for LOTOS specifications.

It seems to be roughly accepted [26, 27] that mixed methods are of great interest and are likely to yield distributed systems specifications that enjoy the three important properties stressed by [27]: *orthogonality* (independent architectural requirements should be specified by independent definitions), *generality* (generic, parameterized definitions should be preferred over collections of special purpose definitions), and *open-endedness* (designs should be maintenable, i.e. easy to extend and modify).

4 A New Method for LOTOS Specification

4.1 Introduction

In this section we describe our method (Figure 3) for LOTOS specification and we illustrate it with the example of a hospital. As explained in the previous section it is based on our study of the approach followed by [11]. Our method brings some novel features such as the semi-automatic retrieval of the sequential components automatons and the use of the Ω-derivation principles [4] to obtain axioms from the automaton (a complete description may be found in [23]).

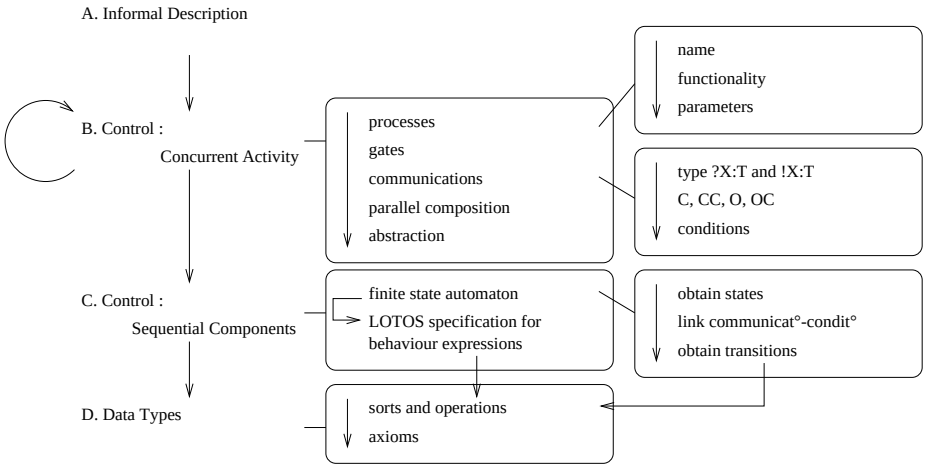


Fig. 3. Our method for LOTOS specification

4.2 Dynamic Part

This part of our method refers to the dynamic part of LOTOS specifications.

A. Informal Description. The system is described in natural language with a clear statement on its components and their interactions.
The case study is a hospital informally described as follows. There are usual admissions (patients) in a limited number and emergency admissions in an unlimited number. The admissions are processed in a first-in first-out way but the emergencies have priority. Externally, the following services (functionalities) are available: PRESENT (is a patient present in the hospital ?), ADMIT (non emergency admission), EMERGENCY (emergency admission) and EXIT (exit of the latter healed patient). Let us note that in this case study, the communication conditions will depend on the processes data.

B. Concurrent Activity. This phase consists of several activities (Figure 3) which are described below: (1) identify the different *processes* corresponding to parallel components, and provide for each process its name, functionality and parameters, (2) determine the communication *gates* between these processes, (3) study the *communications* (typing, conditions, etc.), (4) express *parallel composition* of the processes, and (5) possibly use *abstraction* to hide some communications.

B.1. Processes. Processes are used to model components that run in parallel. The processes "functionalities" (e.g. `noexit` when recursive) are defined. Each process has its own internal private data (static part) together with a behaviour (dynamic part). A data type is created for each process type and an object of that type is used as the process formal parameter. This decomposition is constraint-oriented.

B.2. Gates. Interactions between components (from the informal description) are modeled with formal gates representing the services (interface) of the component.

At this point, we obtain a single nonterminating process, the hospital, parameterized by an object of type Hospital, with the services PRESENT, ADMIT, EMERGENCY and EXIT:

```
process HOSPITAL[PRESENT,ADMIT,EMERGENCY,EXIT] (h:Hospital) : noexit :=
...
endproc
```

B.3. Communications. In this phase, the communications (on the gates) between the components are typed. Note that we require an explicit typing also for outputs (this symmetry is useful at the data type specification level). All data types used here have an algebraic specification. Generic types can be used but have to be instantiated later. For each process, four sets of (typed) gates are determined: C (constructors without conditions), CC (constructors with conditions), O (observers without conditions) and OC (observers with conditions). Constructors refer to operations that modify the internal data of the process

and observers to operations that do not modify it. Conditions are preconditions (stating when an operation is possible or not) or "behaviour conditions" (i.e. the resulting state depends on their value). This partition helps creating the automatons in a semi-automatic way. $\sigma = C \cup CC \cup O \cup OC$ is the set of all the gates of the process. This partition is not global on the gates names but rather local to each process.

Considering the hospital process, we have:

- OC is empty
- O: PRESENT : ?p:Patient, !x:Bool
- C: EMERGENCY : ?p:Patient
- CC:
  - EXIT : !p:Patient, with conditions `c-notEmpty` (a precondition used to state that EXIT is only possible when the hospital is not empty) and `c-emergency` (a behaviour condition needed to express that the result of the EXIT action depends on whether or not there is an emergency patient)
  - ADMIT : ?p:Patient, with condition `c-notFull` (a precondition used to state that ADMIT is only possible when the hospital is not full).

We can obtain signatures from this typing ([23] for details). Note that all data types introduced in this step should be later specified.

B.4. Parallel composition. The components are composed using LOTOS parallel operators. We propose to use composition patterns when possible. See [19, 10, 18, 14] for examples. The pattern instantiation has to be simulated (see [25] for example) since LOTOS has no way to parameterize behaviours (apart from the gate names and data parameters).

B.5. Abstraction. When necessary, communications between components can be hidden using the `hide` operator.

C. Sequential Components. As noted by [26], the use of automata is appropriate since representing whole behaviours by means of trees is practical only when the node number is limited (which is not the case for recursive processes). In this phase the LOTOS behaviour expression is obtained from a finite state automaton.

C.1. Finite State Automaton. A process has a finite number of abstract states in which it can react to finite sets of communications. The composition of the communication conditions obtained above is used (Table 1) to determine the different states of the automaton. Relationships (expressed by formulas) between the conditions eliminate the incoherent states (here: `c-emergency` $\Rightarrow$ `c-notEmpty`, and: `c-notEmpty` $\vee$ `c-notFull`). Boolean tuples characterize the states.

[1] An "intermediate" state corresponds to a hospital that is neither empty nor full.

Table 1. Conditions composition

c-notEmpty	c-emergency	c-notFull	interpretation	reference
true	true	true	intermediate with emergency	InterEmer[1]
true	true	false	full with emergency	FullEmer
true	false	true	intermediate without emergency	InterNoEmer[1]
true	false	false	full without emergency	FullNoEmer
false	true	true	impossible	
false	true	false	impossible	
false	false	true	empty hospital	Empty
false	false	false	impossible	

The transitions of the automaton correspond to communications leading from an abstract state to another one (or possibly the same). To obtain the transitions, we will use their preconditions and postconditions (Table 2 where ∀ states that the condition value is not relevant). Expressing the postconditions may require the use of new conditions[2] (e.g. `c-onlyOnePatient` below). These new conditions will be used to guard the transitions. Relationships between these conditions ("old" and new ones) will limit the number of transitions and prevent creating transitions leading to incoherent states.

Table 2. Preconditions and postconditions

	preconditions		
gate	c-notEmpty	c-emergency	c-notFull
EMERGENCY	∀	∀	∀
ADMIT	∀	∀	true
EXIT	true	∀	∀

	postconditions		
gate	c-notEmpty'	c-emergency'	c-notFull'
EMERGENCY	true	true	c-notFull
ADMIT	true	c-emergency	¬ c-nextFull
EXIT	¬ c-onlyOnePatient	¬ c-onlyOneEmergency	¬ c-emergency ∨ c-notFull

O operations (observers without conditions) are always enabled and do not modify the process data, so they are represented by transitions from a state to itself (transition labelled by o in Figure 4). OC operations are only possible in some states and are represented by transitions from the states satisfying the conditions and leading to the same state.

For the constructors (C and CC operations) we use the following scheme: for each abstract state e identified by a tuple t_e, if t_e satisfies the preconditions

[2] See [23] for a discussion on the new conditions appearing in the postconditions.

of the constructor, we compute the tuple t_f corresponding to the postconditions and there is a transition from e to f.

For example, let us consider the emergency admission of a patient, EMERGENCY, and the state `Empty`. Its corresponding tuple is `<false,false,true>`. It matches the EMERGENCY precondition ($< \forall, \forall, \forall >$) with {`c-notEmpty`=false, `c-emergency`=false, `c-notFull`=true}. Applying this substitution to the EMERGENCY postcondition we obtain `<true,true,true>`, that characterizes the `InterEmer` state; therefore there is a transition labelled by EMERGENCY from `Empty` to `InterEmer`.

We now need to determine the initial state (which should be unique for data coherence sake). It is obtained from the conjunction of the preconditions of the operations that should be possible or not in this state. Here we want ADMIT and EMERGENCY to be possible, and EXIT not to be possible. Therefore the initial state should satisfy $\neg$ `c-empty` $\wedge$ `c-notFull` which corresponds to `<false,`$\forall$`,true>`, that is the (`Empty`) state.

The complete automaton is given in Figure 4 and the conditions associated to the numbers are given in Table 3. Let us note that our method leads us automatically to take into consideration "special cases" as one-place hospitals (see the guarded transitions [1*] and [7*] in Figure 4).

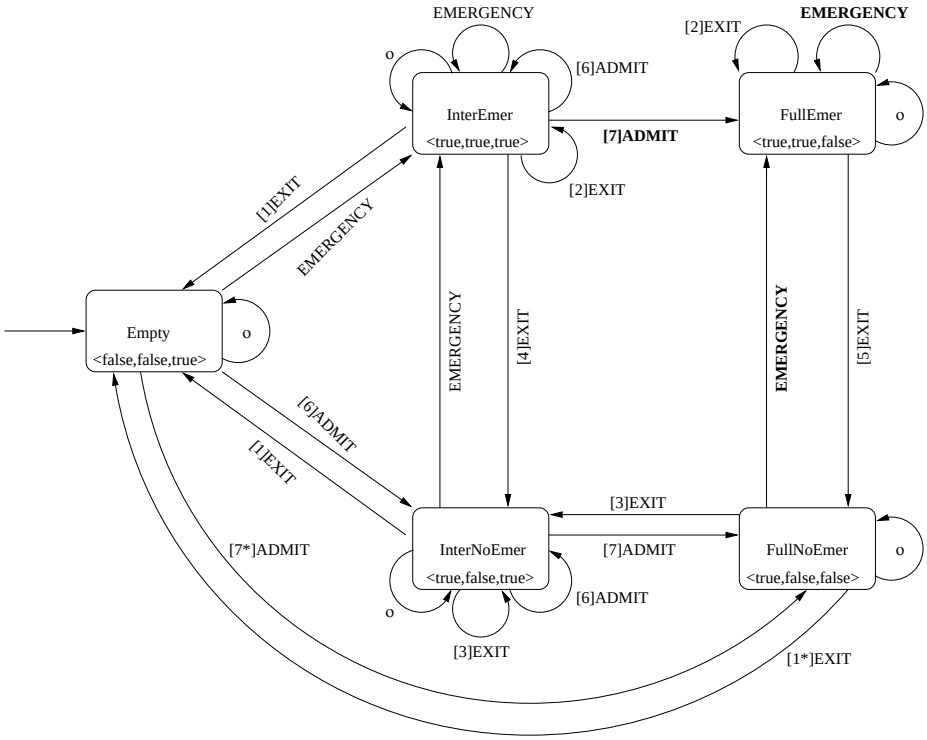


Fig. 4. Hospital automaton

Table 3. Transitions conditions

number	condition
1, 1*	c-onlyOnePatient
2	¬ c-onlyOneEmergency
3	¬ c-onlyOnePatient
4	¬ c-onlyOnePatient ∧ c-onlyOneEmergency
5	c-onlyOneEmergency
6	¬ c-nextFull
7, 7*	c-nextFull

C.2. LOTOS Behaviour Expressions. We herein summarize the retrieval of the LOTOS behaviour expressions from the automatons. This can be done using two different patterns: a single process is associated with each automaton, and (1) for each state a conditional branch is created and some simplifications are achieved to get readable behaviour expressions, or (2) a subprocess is created for each state of an automaton (this last approach requires/allows much less simplifications). Depending on the goal to be achieved, the user may then choose between both approaches. This issue is further discussed in [23].

We herein chose the first approach (Fig. 5). Process 1 is obtained after grouping all conditional branches corresponding to the same operation (see next page). Then Process 2 is obtained after simplification of the guards (which may be achieved using some proof tool).

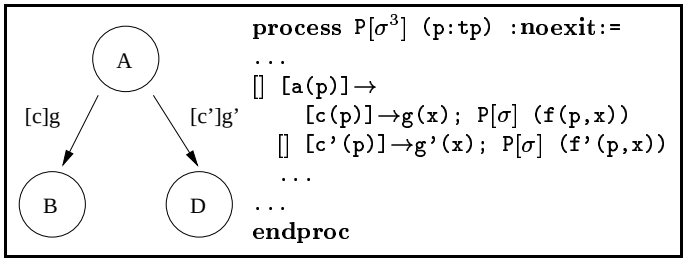


Fig. 5. Transformation pattern (1)

4.3 Static Part: D. Data Types

The last step is to build the specification of the data types associated with each process. For each data type, the work done in the preceding steps provides most of the signature (cf. [23] for the automatic processing description). Indeed, the operation names and profile are retrieved from the communications typing, and

[3] σ denotes the process gates.

Process 1

```
process HOSPITAL[PRESENT,EMERGENCY,EXIT,ADMIT] (h: Hospital) : noexit :=
   PRESENT ?p:Patient !(present(h,p)); HOSPITAL[σ⁴] (h)
[] EMERGENCY ?p:Patient; HOSPITAL[σ] (emergency(h,p))
[] [(empty(h) ∧ ([6] ∨ [7*]))
   ∨ (interEmer(h) ∧ ([6] ∨ [7]))
   ∨ (interNoEmer(h) ∧ ([6] ∨ [7]))]→
      ADMIT ?p:Patient; HOSPITAL[σ] (admit(h,p))
[] [(interEmer(h) ∧ ([1] ∨ [2] ∨ [4]))
   ∨ (interNoEmer(h) ∧ ([1] ∨ [3]))]
   ∨ (fullEmer(h) ∧ ([2] ∨ [5]))]
   ∨ (fullNoEmer(h) ∧ ([1*] ∨ [3]))]→
      EXIT !first(h); HOSPITAL[σ] (cure(h))
endproc
```

Process 2

```
process HOSPITAL[PRESENT,EMERGENCY,EXIT,ADMIT] (h: Hospital) : noexit :=
   PRESENT ?p:Patient !(present(h,p)); HOSPITAL[σ] (h)
[] EMERGENCY ?p:Patient; HOSPITAL[σ] (emergency(h,p))
[] [c-notFull(h)]→ADMIT ?p:Patient; HOSPITAL[σ] (admit(h,p))
[] [c-notEmpty(h)]→EXIT !first(h); HOSPITAL[σ] (cure(h))
endproc
```

from the different conditions found while building the automaton. A few additional operations may be needed to express some axioms. Most of the axioms[5] are written in a "constructive style" which requires to identify the generators. [1] develops a method for retrieving a data type associated to an automaton and to compute a minimal set of operations needed to reach all its states. In our example this set is {`init, admit, emergency`}. The GAT method developed in [1] uses the Ω-derivation [4] to write the axioms (conditional equations). To extract the axioms describing the properties of an operation `op(h:Hospital)`, the states where `op` is possible are searched together with the generators reaching these states which leads to the left hand sides of the equations. The premises of the axioms are built from the features of the generators associated edges: (i) the edges source state characteristic function, and (ii) the edges condition. Then, an axiom is generated for each transition associated to the operation `op` in a given state. An additional premise may be generated whenever there is a guard on this transition. The specifier has to provide the right hand side terms when the premises are proved to be satisfiable.

We show here a part of the extraction process for the axioms of the `cure` operation that is associated with EXIT. This operation is possible in the states:

[4] σ denotes the process gates, here: PRESENT,EMERGENCY,EXIT,ADMIT.
[5] except the formulas expressing the relationships between the conditions

`InterEmer, FullEmer, InterNoEmer, FullNoEmer`. Let us consider the state `FullEmer`. There are two EXIT edges in this state with respectively guards `[2]` and `[5]` and three generator edges (**ADMIT** and **EMERGENCY** in Fig. 4). We should get 3×2 axioms but some premises are not satisfiable, hence we get the following 4 axioms:

```
% AXIOM 1:  "InterEmer" + [7] admit + [2]
(interEmer(h) /\ cNextFull(h) /\ ~cOnlyOneEmergency(admit(h, p))) =>
            cure(admit(h, p)) = admit(cure(h), p);
% AXIOM 2: "InterEmer" + [7] admit + [5]
(interEmer(h) /\ cNextFull(h) /\ cOnlyOneEmergency(admit(h, p))) =>
            cure(admit(h, p)) = admit(cure(h), p);
% AXIOM 3: "FullEmer" + loop emergency + [2]
(fullEmer(h) /\ ~cOnlyOneEmergency(emergency(h, p))) =>
            cure(emergency(h, p)) = emergency(cure(h), p);
% NO AXIOM: "FullEmer" + loop emergency + [5] : not possible
% (fullEmer(h) /\ cOnlyOneEmergency(emergency(h, p)))
% NO AXIOM: "FullNoEmer" + vertical emergency + [2] : not possible
% (fullNoEmer(h) /\ ~cOnlyOneEmergency(emergency(h, p)))
% AXIOM 4: "FullNoEmer" + vertical emergency + [5]
(fullNoEmer(h) /\ cOnlyOneEmergency(emergency(h, p))) =>
            cure(emergency(h, p)) = h;
```

Once the algebraic specification is written, it may be used to prove properties that are useful to validate and verify the specification (e.g. we processed our example using LP [12] which could compute a canonical rewriting system from our axioms).

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". In this paper, we address a specification method for systems where both concurrency and data types issues have to be addressed. Our method takes advantage of both the constraint and state oriented approaches that are used for LOTOS specification. The first step is the informal description of the problem to be solved with a focus on the expected services. The system is then described in terms of parallel components with well defined external interfaces (the gates and communications typing). The behaviour of the components is described by sequential processes that are associated with data types. The study of the communications and their effect on the associated data type allows one to build, in a semi-automatic way, an automaton describing the process internal behaviour. This approach is particularly appropriate for systems where communication conditions depend on the processes data. The automaton is then translated into the specification language LOTOS, for both the dynamic and the static part. So our method permits one to split processes in an easy way and to have data types that are consistent with the sequential processes at once. Note this allows the specifier to use both tools

to check properties on the automaton (model checking or bisimulations e.g. [2]) and theorem provers available for algebraic specifications (e.g. LP [12]).

As in the "agendas" described by [13], our method clearly establishes for each step what is to be achieved, and what are the pieces of information to be provided and then how they can be automatically used to save part of the work the specifier has to do.

While we present our method in detail when applied to LOTOS specification, it is clear that it may be applied to other formalisms. Work on SDL [8], Estelle [16] and Object-Z [24] has been done [22] or is under study.

Our method as been used on small size examples (as a transit node [22]) but still has to be tested on industrial size case-studies.

Acknowledgements: We would like to thank our referees for their careful reading and interesting remarks.

References

1. 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.
2. André Arnold. *Systèmes de transitions finis et sémantique des processus communicants.* Etudes et recherches en informatique. Masson, 1992.
3. E. Astesiano and G. Reggio. Formalism and method. In M. Bidoit and M. Dauchet, editors, *TAPSOFT'97*, volume 1214 of *Lecture Notes in Computer Science*, pages 93–114. Springer-Verlag, 1997.
4. Michel Bidoit. Types abstraits algébriques : spécifications structurées et présentations gracieuses. In *Colloque AFCET, Les mathématiques de l'informatique*, pages 347–357, Mars 1982.
5. Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–29, January 1988.
6. R.G. Clark. Using LOTOS in the Object-Based Development of Embedded Systems. In C.M.I. Rattray and R.G. Clark, editors, *The Unified Computation Laboratory*, pages 307–319, Department of Computing Science and Mathematics, University of Stirling, Scotland, 1992. Oxford University Press.
7. H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification*, volume 1. Springer-Verlag, Berlin, 1985.
8. Jan Ellsberger, Dieter Hogrefe, and Amardeo Sarma. *SDL : Formal Object-oriented Language for Communicating Systems.* Prentice-Hall, 1997.
9. Hubert Garavel. *Compilation et vérification de programmes LOTOS.* Thèse de doctorat (PhD Thesis), Université Joseph Fourier, Grenoble, Novembre 1989.
10. Hubert Garavel. Introduction au langage LOTOS. Technical report, VERILOG, Centre d'Etudes Rhône-Alpes, Forum du Pré Milliet, Montbonnot, 38330 Saint-Ismier, 1990.
11. Hubert Garavel and Carlos Rodriguez. An example of LOTOS Specification : The Matrix Switch Problem. Rapport SPECTRE C22, Laboratoire de Génie Informatique — Institut IMAG, Grenoble, June 1990. Available at http://www.inrialpes.fr/vasy/Publications/Garavel-Rodriguez-90.html.

12. 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.
13. 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.
14. 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.
15. C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.
16. ISO/IEC. ESTELLE: A Formal Description Technique based on an Extended State Transition Model. ISO/IEC 9074, International Organization for Standardization, 1989.
17. ISO/IEC. LOTOS: A Formal Description Technique based on the Temporal Ordering of Observational Behaviour. ISO/IEC 8807, International Organization for Standardization, 1989.
18. 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.
19. G. J. Leduc. LOTOS, un outil utile ou un autre langage académique ? In *Actes des Neuvièmes Journées Francophones sur l'Informatique — Les réseaux de communication — Nouveaux outils et tendances actuelles (Liège)*, Janvier 1987.
20. L. Logrippo, M. Faci, and M. Haj-Hussein. An Introduction to LOTOS: Learning by Examples. *Computer Networks and ISDN Systems*, 23:325–342, 1992. improved version available by ftp at lotos.csi.uottawa.ca.
21. Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1980.
22. 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, November 1998. /papers/rr180.ps.gz in Poizat's web page.
23. 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, February 1998. /papers/rr170.ps.gz in Poizat's web page.
24. Graeme Smith. A Fully-Abstract Semantics of Classes for Object-Z. *Formal Aspects of Computing*, 7(E):30–65, 1995.
25. K. Turner. Relating architecture and specification. *Computer Networks and ISDN Systems*, 29(4):437–456, 1997.
26. Kenneth J. Turner, editor. *Using Formal Description Techniques, An introduction to Estelle, LOTOS and SDL*. Wiley, 1993.
27. C. A. Vissers, G. Scollo, M. Van Sinderen, and E. Brinksma. Specification Styles in Distributed Systems Design and Verification. *Theoretical Computer Science*, 89(1):179–206, 1991.