Algebraic Specification at Work*

Egidio Astesiano Alessandro Giovini Gianna Reggio
Dipartimento di Matematica
Università di Genova – Italy

Franco Morando
Elsag Bailey S.p.A.
via Puccini 2, Genova – Italy

Abstract

Concurrent software systems are used in fields where safety and reliability are critical. In spite of that, the validation of concurrent software tends to rely more on empirical experiments rather than on rigorous proofs. This practice, which is already bad for sequential software, becomes awful for concurrent systems since, due to nondeterminism, the behaviour observed during testing is only the most likely to occur. We present a formal approach with associated tools for handling this situation. This method allows us to derive from the specification all possible behaviours of a concurrent program and hence to be sure that a tested program will always behave correctly on its test-bed. We outline the approach discussing its application to a typical situation in industrial practice.

Introduction

Concurrent systems are usually programmed using a standard sequential language mixed with calls to the underlying system primitives. If we model this mix as a concurrent language $\mathcal{L}$ (as it actually is), then the concurrent system becomes a program p in $\mathcal{L}$. We need methods for specifying rigorously both the concurrent system and its programming environment in a uniform framework.

During the specification phase a tool set can be used to check specification correctness and consistency. When the specification is ready programmers or hardware designers in charge of developing the system can use a language interpreter to run test programs. Prototyping is an informal approach to specification understanding of valuable interest in industrial environments: it enables companies to introduce formal specifications hiring a limited number of specialists in formal techniques, while implementors can use formal documents (that they probably cannot understand) through the filter of the tool set. When the specified system has been implemented, programs running on the system can be tested using a language interpreter. This application of formal techniques to concurrent software testing is interesting since the interpreter, if based on a formal specification, can fully exploit the nondeterminism introduced by concurrency.

Let us now briefly illustrate our formal approach. It consists of *conceptual tools*, *software tools* and *application development rules*.

Conceptual tools They are essentially a technique for modelling and specifying concurrent systems and languages. In practical applications we distinguish them in two parts: *Structural Concurrent Design and Specification(SCDS)* (a structural and hierarchical approach to the design and the formal specification of concurrent systems expressed as algebraic specifications), and *Formal Specification of Concurrent Languages (FSCL)* (given the syntax of a concurrent language, we specify its semantics by a translation into a behaviour language for describing processes; by applying the *SCDS* technique we build an abstract concurrent machine which gives the semantics of the behaviour language).

Software tools They consist essentially of *A Concurrent Rapid Prototyping System (CRP)* (given an abstract concurrent machine and an initial state *CRP* explores all finite execution paths) and of a *Concurrent Language Translator (CLT)* (given some language semantic clauses and a program, *CLT* translates it into a state of the abstract underlying concurrent machine).

Application development rules They are application dependent and consist of rigorous guidelines using the conceptual and software tools for the correct development of a class of applications. In this paper we illustrate the rules for a particular class, by means of a very simple example. We consider a concurrent schematic language $\mathcal{EL}^{[\,]}$ handling access to a shared resource parameterized over the mechanism for regulating the access; we instantiate then the language on two different mechanisms, a semaphore and shared bit; we try to implement the first with the second and use the tools to check the implementation.

After presenting conceptual and software tools in the first two sections, in the third section we show in practice the application development rules. The paper is organized so that the reader may start with section 3, going back to the previous sections only when in need of some more detailed explanations.

*This work has been partially supported by "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo" of CNR Italy and by the COMPASS ESPRIT-BRA W.G. n. 3264.

1 Conceptual Tools

The conceptual tools presented here are essentially an adaptation of the *SMoLCS* approach; the reader interested in a more complete presentation with mathematical foundations may refer to [AR1, AR2, AR3].

1.1 Structured Concurrent Design and Specification

The Structured Concurrent Design and Specification (*SCDS*) approach to the specification of concurrent systems is aimed at giving both *formal* and *behavioural guidelines* to those who are engaged in giving rigorous definitions of concurrent architectures, in the sense that:

- the foundations of the method rely on a rigorous mathematical theory which ensures the soundness of the overall schema and which does also guarantee that some fundamental properties of what is being specified do hold (for example, that the specifications admit models, that symbolic execution is possible, and so on);
- the guidelines given by the method correspond to requirements of *structurality* and *universality*, i.e., they provide a canonical way of structuring the specification (of a concurrent system) which is capable of specifying *any* concurrent system; also, the structure of the specification enlightens relevant concepts of the specified system or language and hence helps the designer in isolating the critical features.

We believe hence that these guidelines correspond to those provided by the method of structured programming for the development of programs in the sequential case. In section 1.1.1 we briefly present the principles of the method; while in section 1.1.2 we apply them to specify in a structured way concurrent systems.

1.1.1 Process Specification

The formal model of processes is that of *labelled transition systems.* A labelled transition system is a triple (S, L, $\longrightarrow$), where S is the set of the *states* of the system, L is the set of the *labels* and $\longrightarrow\ \subseteq \mathrm{S}\times\mathrm{L}\times\mathrm{S}$ is the *transition relation*, and as usual we write $s \xrightarrow{l} s'$ for $(s, l, s') \in\ \longrightarrow$. The usual interpretation of $s \xrightarrow{l} s'$ is that a process in the state s can evolve into the state s' by performing a transition labelled by l; the label l conveys the information related to the transition which is visible to the external world; as a convention, τ is used to label transitions which correspond to internal computstions.

In our approach, labelled transition systems are models of particular algebraic specifications: thus we talk of *algebraic transition systems.* The specification of an algebraic transition system TS is a specification with predicates (**preds**) of the form in figure 1, where S and L are the specifications of states and labels respectively, while SAx, DAx are sets of *static* and *dynamic* axioms respectively; they are sets of *positive conditional axioms*, i.e. first order formulas having form $\alpha_1 \wedge \ldots \wedge \alpha_n \supset \alpha_{n+1}$, where for $i = 1, \ldots, n+1$, α_i has form either $t = t'$ or $p(t_1, \ldots, t_k)$, with p predicate symbol.

spec TS = **enrich** $S + L$ **by**
  preds $- \xrightarrow{-} -: state \times label \times state$
  axioms SAx, DAx

Figure 1: Algebraic transition system schema

Static axioms SAx define the properties of the states and of labels, while dynamic axioms DAx define the transition relation $\longrightarrow$. Hence states, labels and transitions are specified axiomatically. Thus, while giving the axioms DAx, one can benefit from the identifications on states and labels which are induced by the axioms SAx. As a very simple example, once we include in a *CCS*-like language specification the static axiom $s' + s'' = s'' + s'$, we just need one dynamic axiom $s' \xrightarrow{l} s_1' \supset s' + s'' \xrightarrow{l} s_1'$ to specify the transitions related to the nondeterministic choice operator $+$, since the symmetrical one comes by application of the commutativity axiom.

By restricting the kind of axioms to the class of positive conditional formulas one is ensured that the specification admits the initial model, say I, characterized by: $I \models t = t'$ iff $TS \vdash t = t'$, and $I \models s \xrightarrow{l} s'$ iff $TS \vdash s \xrightarrow{l} s'$, where $\vdash$ denotes the usual Birkoff deductive system for conditional specifications (see, e.g., [GM]). So, in these cases, dynamic axioms can be considered as a set of inference rules defining the interpretation of $\longrightarrow$ in I; for this reason in the following we call the dynamic axioms *inference rules* or simply *rules*; moreover a dynamic axiom having form $\alpha_1 \wedge \ldots \wedge \alpha_n \supset s \xrightarrow{l} s'$ will be written $\dfrac{\alpha_1 \ \ldots \ \alpha_n}{s \xrightarrow{l} s'}$.

In this way we generalize the *SOS* approach (see [P]) by neatly separating *static* and *dynamic* properties: static properties are specified by means of axioms requiring identifications on the data types of the system, while dynamic properties are specified by means of inference rules.

As a notation, if TS is the specification of a transition system, we write "the transition system TS" for "the the initial model of the specification TS" and write α for $TS \vdash \alpha$.

1.1.2 Structured Specifications of Abstract Concurrent Machines

Processes are just the basic building blocks of complex systems, where several processes (which may also have a complex structure) interact between them. In this section we show how the method allows us to structure the specification of a complex system, by giving an abstract concurrent machine starting from the specification of its processes. The structuring is both *static* (dealing with the structure of the machine) and *dynamic* (dealing with the structural definition of the activity of the machine).

Static structuring To specify a complex concurrent system first of all we need to determine which are its subcomponents, distinguishing them in *active components* or *process components* and *passive components* or *global information* (as shared memories, buffers, communication media and so on). Active components are specified as transition systems (and hence their behaviour is determined by a set of dynamic axioms), while passive components are represented by data types specified by static axioms. So, complex systems are specified by using a subclass of transition systems, which we call *abstract concurrent machines*, whose states couples of the form $<p_1 \mid \cdots \mid p_n, i>$ where

- $p_1 \mid \cdots \mid p_n$ is a multiset of *processes*, which represent the active components of the machine,
- each process p_i is a state of a *Basic Transition System* (*BTS*), representing the process activity,
- i is a *global information*, i.e. a value representing the status of the (shared) passive components.

Dynamic structuring It is important to have a criterium for defining the transitions of an abstract concurrent machinestarting from those of the active components. Our method splits this procedure into three *steps*: *synchronization*, *parallelism* and *monitoring*. Informally, we start from the basic transitions that processes can perform (e.g. performing an internal action, attempting to read a shared variable, starting a handshaking communication). In the synchronization step we specify which sets of such transitions *must* occur simultaneously and which is their effect on the global information (for example, the action of a process P starting a handshaking communication must occur simultaneously to a corresponding action of another process). In the parallelism step we specify which groups of such synchronized actions *can* occur simultaneously (for example, two τ actions are susually allowed to occur simultaneously, while the simultaneous access to a same shared variable can be forbidden). Finally, in the monitoring step, we select which parallel compositions of synchronized actions are allowed to become transitions of the abstract concurrent machine (for example, we can handle here scheduling policies).

Formally, starting from the specification of the basic transition system, one builds the specifications of three transition systems corresponding respectively to the synchronization step, the parallelism step and the monitoring step (the last system is the abstract concurrent machine). The states of these systems have the common form $<p_1 \mid \cdots \mid p_n, i>$ and their transition relations represent the kind of transitions we have informally explained above; the methodology gives the schemas that the three specifications have to follow. Again, these schemas correspond to formal and behavioural guidelines.

Synchronization In this step starting from a transition system *BTS* with transition relation $- \overset{-}{\longrightarrow} -$: *process* $\times$ *label* $\times$ *process* (modelling the behaviour of the active components) and from a specification *INF* (modelling the global information) we build a new transition system, called *STS* (see figure 2), whose

> **spec** STS = **enrich** $BTS + INF + SLABEL$ **by**
> **sorts** $state$, $mset(process)$
> **opns** $-: process \to mset(process)$
> $- \mid -: mset(process) \times mset(process) \to mset(process)$
> $<-,->: mset(process) \times inf \to state$
> **preds** $- \overset{-}{\Longrightarrow} -: state \times slabel \times state$
> **axioms** $pms_1 \mid pms_2 = pms_2 \mid pms_1, (pms_1 \mid pms_2) \mid pms_3 = pms_1 \mid (pms_2 \mid pms_3), Ax_{STS}$

Figure 2: The synchronization schema

states are the states of the concurrent machine and whose transitions model the synchronized transitions of processes. The labels of *STS* are given as an extra specification *SLABEL*; we introduce a new set of labels since the amount of information about a synchronous transition which is left visible can be different from what is visible out of each basic transition considered on his own; for example, a rendezvous which is taking place between two processes could become nonobservable.

The method requires each inference rule in Ax_{STS} to be of the following form

$$\frac{p_1 \xrightarrow{l_1} p'_1 \ \cdots \ p_n \xrightarrow{l_n} p'_n}{<p_1 \mid \cdots \mid p_n, i> \overset{sl}{\Longrightarrow} <p'_1 \mid \cdots \mid p'_n, i'>} \quad \text{cond}$$

where all the p_j's and p'_j's are variables for process states, and its intuitive meaning is: whenever the (global) condition cond holds, the n actions $p_1 \xrightarrow{l_1} p'_1, \ldots, p_n \xrightarrow{l_n} p'_n$ of the component processes can be synchronized and they result in the synchronized transition $<p_1 \mid \cdots \mid p_n, i> \stackrel{sl}{\Longrightarrow} <p'_1 \mid \cdots \mid p'_n, i'>$ labelled by sl (notice that the global information changes from i to i').

The formal requirement of p_j's and p'_j's to be variables corresponds to the method requirement that the transitions of the abstract concurrent machine we are building depend at most on the labels of the transitions of the active subcomponents and on the global information, i.e. labels are all and only what is visible of the transitions of the active components, since we are not allowed to explicitly make the evolution of the system on the structure depending of such components.

Parallelism In this step we start from the transition system STS and build a new transition system PTS (see figure 3) having the same states as STS and as transitions the "parallel compositions" of transitions

> **spec** PTS = **enrich** STS **by**
> **opns** $- // -: slabel \times slabel \rightarrow slabel$
> **axioms** $sl // sl' = sl' // sl, sl // (sl' // sl'') = (sl // sl') // sl'', Ax_{PTS}$

Figure 3: The parallelism schema

of STS. We introduce new labels by means of the operation $- // -: slabel \times slabel \rightarrow slabel$ for labelling parallel compositions of synchronous transitions.

The method requires each inference rule contained in Ax_{PTS} to be of the following form

$$\frac{<pms_1, i> \stackrel{sl_1}{\Longrightarrow} <pms'_1, i'_1> \quad <pms_2, i> \stackrel{sl_2}{\Longrightarrow} <pms'_2, i'_2>}{<pms_1 \mid pms_2, i> \xRightarrow{sl_1 // sl_2} <pms'_1 \mid pms'_2, i'>} \quad \text{cond}$$

where pms_1, pms_2, pms'_1, pms'_2 are variables. The intuitive meaning of such rules is: whenever the condition cond holds, the two actions $<pms_1, i> \stackrel{sl_1}{\Longrightarrow} <pms'_1, i'_1>$ and $<pms_2, i> \stackrel{sl_2}{\Longrightarrow} <pms'_2, i'_2>$ can be performed in parallel and they result in the transition $<pms_1 \mid pms_2, i> \xRightarrow{sl_1 // sl_2} <pms'_1 \mid pms'_2, i'>$. Notice that sl_1 and sl_2 can be either labels of synchronized transitions or, in turn, of parallel compositions of synchronized transitions; the transformation of the global information from i into i' in general depends on the transformations performed by the transitions labelled by sl_1 and sl_2. The requirement of pms_1, pms_2, pms'_1, pms'_2 to be variables has the same motivation as in the synchronization step.

Monitoring In the monitoring step we can specify a form of global control on the activity of the process components of the machine. We start from the transition system PTS and build the abstract concurrent machine ACM (see figure 4) having the same states as PTS and as transitions the "local actions" (transitions

> **spec** ACM = **enrich** $PTS + CLABEL$ **by**
> **preds** $- \stackrel{-}{\Longrightarrow} -: state \times clabel \times state$
> **axioms** Ax_{ACM}

Figure 4: The monitoring schema

of PTS) which are allowed by the global control. The labels of ACM are given as an extra specification $CLABEL$ since the amount of information which is left visible of a monitored transition can be different from that which is visible of the parallel one.

The method requires each inference rule contained in Ax_{ACM} to be of the form

$$\frac{<pms, i> \stackrel{sl}{\Longrightarrow} <pms', i'>}{<pms \mid pms_1, i> \stackrel{cl}{\Longrightarrow} <pms' \mid pms_1, i'>} \quad \text{cond}$$

The intuitive meaning of such rules is: whenever the condition cond holds, the monitoring allows the local action $<pms, i> \stackrel{sl}{\Longrightarrow} <pms', i'>$ to become the transition $<pms \mid pms_1, i> \stackrel{cl}{\Longrightarrow} <pms' \mid pms_1, i'>$ of the abstract concurrent machine, labelled by cl, where all the subcomponents in pms_1 stay idle.

1.2 Formal Specification of Concurrent Languages

Our method supports also the specification of concurrent languages; given a language $\mathcal{L}$, the specification of the semantics of $\mathcal{L}$ is split into two steps.

In the first step one determines what the *Underlying Concurrent Abstract Machine* of the language (shortly $\mathsf{UACM}^{\mathcal{L}}$): this is the (initial model of a) specification of an abstract concurrent machine whose states correspond to the intermediate stages of the executions of $\mathcal{L}$ programs. The basic components could correspond, for example, to tasks, while the global information could represent a shared memory. The terms representing the states of the $\mathsf{UACM}^{\mathcal{L}}$ form a language that we call the *Behaviour Language associated with* $\mathcal{L}$ ($\mathsf{BL}^{\mathcal{L}}$).

Then the second step consists in giving a translation in a compositional way (formally: a homomorphism) from the syntax of the language into $\mathsf{BL}^{\mathcal{L}}$. This translation is given by a set of inductive clauses called Language Semantic Clauses ($\mathsf{LSC}^{\mathcal{L}}$).

By composition of the two steps one defines the semantics of a program *prog* of $\mathcal{L}$ to be behaviour associated with the translation of *prog*.

2 Tools

Our method is supported by a full set of development tools whose goal is to help the user of the method engaged in giving the formal definition of architectures or languages during the development phase. The tools, that are currently under development, are:

A Concurrent Rapid Prototyping System (CRP) that allows the user to debug the specification of an abstract concurrent machine by making experiments with a running prototype of the specified machine (corresponding to the initial model of the specification).

A Concurrent Language Translator (CLT) for the part of the method related to the definition of concurrent languages, that taken in input the semantic clauses $\mathsf{LSC}^{\mathcal{L}}$ for a given language $\mathcal{L}$ and a program *prog* of $\mathcal{L}$, returns the state s_{prog} of the $\mathsf{UACM}^{\mathcal{L}}$ corresponding to the beginning of the execution of *prog*.

CLT together with *CRP* can be used to test programs written in a concurrent language with a semantics given following our approach. Indeed, *CLT* translates programs of the language $\mathcal{L}$ into programs of the behaviour language $\mathsf{BL}^{\mathcal{L}}$ (states of the $\mathsf{UACM}^{\mathcal{L}}$), which can be seen as a "machine code" for the $\mathsf{UACM}^{\mathcal{L}}$. *CRP* "executes" this machine code simulating the evolution of that machine, thus generating the labelled tree corresponding to all possible program executions. *CLT* is a standard interpreter for inductive clauses, that does not present particular interesting features, while *CRP* is the real critical part of the tool set; hence in the following we will focus on it.

2.1 The Concurrent Rapid Prototyping System *CRP*

The specification of an *ACM* individuates a labelled transition system (its initial model) with transition relation $\longrightarrow$ characterized by the property $s \xrightarrow{l} s'$ iff $ACM \vdash s \xrightarrow{l} s'$. When ACM and one state s_0 are given, we can infer its behaviour starting from s_0 by finding all states s' and labels l such that $s_0 \xrightarrow{l} s'$. In general, if s_0 does not correspond either to a deadlocked state or to a final state, then there is more than one couple (l, s') s.t. $s_0 \xrightarrow{l} s'$, corresponding to a nondeterministic choice between possible transitions of the system. We can repeat the above procedure for each state s' s.t. $s_0 \xrightarrow{l} s'$ and so on. In this way we build a labelled tree, the so called *transition* or *execution tree*, whose nodes correspond to intermediate states reachable from s_0 and whose labelled arcs correspond to the possible transitions. This tree is what we call *the symbolic execution associated with* s_0 *by* ACM. The goal of the *CRP* system is the construction and the analysis of this transition tree.

CRP comprises two distinct tools: a *tree builder* and a *tree walker*. They respectively deal with the automatic generation and with the inspection of the transition tree. Walking on the transition tree the user can take a global view of the machine behaviour; such view helps to locate problems. When the user finds an unexepected behaviour, he can switch to a finer analysis and walk up and down the transition tree inspecting every node. In this way he can understand more deeply what the machine can do starting from a given state, how it can get to that state and (possibly) what is wrong with it.

This interaction schema implies an offline generation of the (part of the) transition tree we want to examine before starting the interactive session. For this reason our tool is split into the two utilities mentioned above; these utilities are compatible, so that the user can traverse the tree while it is still being built and can decide to stop the tree builder when has obtained relevant information.

The tree builder is a noninteractive tool, so that the generation of the tree can proceed offline for hours without assistance. This is mandatory since the tree generation for non-trivial applications takes usually a huge amount of time. The tree builder takes care not to create the same state twice, therefore the generation will not terminate iff the tree contains an infinite number of distinct states.

The tree walker is instead an interactive tool that uses the output produced by the tree builder to display the part of the tree so far generated. In this tool we concentrate all the "bells and whistles" of a friendly user interface. The implementation of this component does not pose any theoretically relevant problems, so we concentrate next on the hard problem, i.e. the automatic generation of the execution tree.

The basic step in the construction of the execution tree is the following: given s_0, find all l and s' such that $s_0 \xrightarrow{l} s'$. When facing this problem for nontrivial specifications one has to cope with problems of memory size and execution time. We have tried many approaches to solve this efficiency problem; these experiments, together with a theoretical analysis and practical considerations, revealed that presently it is not feasible to build a prototyping system for *all* abstract concurrent machine specifications. Therefore we specialized *CRP* on a decidable subclass of specifications, the *separable specifications*, for which a very efficient and yet complete deduction system exists: the *separate conditional narrowing* (SCNA).

Separable specifications are a subclass of predicate specifications that satisfy a decidable set of syntactic restrictions; such class is broad enough to contain abstract concurrent machine specifications of practical interest. In separable specifications we assume a clean semantic distinction between functions and predicates. A function can be used to compute a value only when all its arguments are given, while a predicate defines a relation and may appear in queries, where some arguments may be unknown. If this distinction is satisfied then SCNA, can be used to perform deductions in these theories; a a more detailed presentation can be found in [GiMo]. The implementation of *CRP* is written in Pascal and is based the RAP system [H].

3 Developing a Correct Implementation. A Worked Example: Semaphores

Semaphores are a software mechanism introduced by Dijkstra to handle mutual exclusion problems. A semaphore is an object with an internal status ranging over two values: free and busy. The semaphore is initially set to free. Only two operations are possible on a semaphore: "P" and "V". A process may start a P operation at any time but this operation completes only when the semaphore is free. The P operation sets the semaphore to busy. A process may execute a V operation at any time, a V operation completes immediately regardless of the previous state of the semaphore and sets this state to free. Semaphores are very important in operating system kernels, but it is not difficult to find subtle bugs in published algorithms for implementing them. Here we show how our approach can be used to develop "correct" implementations of a semaphore using a shared bit. The problem can be stated in the following way.

We have a simple concurrent language for building concurrent systems $\mathcal{EL}^{[\,]}$, where the calls to the operating system are given as a parameter. Then, we consider two actualizations of $\mathcal{EL}^{[\,]}$: $\mathcal{EL}^{[\mathrm{Sem}]}$, where the low level primitives are the semaphore operations P and V, and $\mathcal{EL}^{[\mathrm{Bit}]}$, where the low level primitives are the read, write and test&set operations on a shared bit. We consider the problem of implementing the semaphore using $\mathcal{EL}^{[\mathrm{Bit}]}$; precisely, we want to find the $\mathcal{EL}^{[\mathrm{Bit}]}$ code corresponding to P and V, i.e. to find *Pbit* and *Vbit* fragments of $\mathcal{EL}^{[\mathrm{Bit}]}$ programs s.t. for all *prog* in $\mathcal{EL}^{[\mathrm{Sem}]}$ we have that *prog* and $prog[Pbit/\mathrm{P}, Vbit/\mathrm{V}]$ could reasonably be considered "equivalent".

To solve this problem we proceed throughout the following phases.

Formal specification of $\mathcal{EL}^{[\,]}$. We give the parameterized underlying abstract concurrent machine $\mathsf{UACM}[\,]$, describing the concurrent structure of $\mathcal{EL}^{[\,]}$, and the parameterized language semantic clauses $\mathsf{LSC}[\,]$, translating $\mathcal{EL}^{[\,]}$ programs into states of $\mathsf{UACM}[\,]$.

Formal specification of $\mathcal{EL}^{[\mathrm{Sem}]}$ *and* $\mathcal{EL}^{[\mathrm{Bit}]}$. We give the formal definitions of the two languages by instantiating $\mathsf{UACM}[\,]$ and $\mathsf{LSC}[\,]$ with the specification of a semaphore and of a shared bit respectively.

Specification checking and understanding by means of tools. Using the tools associated with the method we check whether the specifications given above are adequate; moreover by experimenting with them we can get a better understanding of the object to be implemented and of the implementation language.

Trial implementations. We propose some implementations of P and V and check whether they are adequate or not; here we first propose an apparently reasonable implementation and by using the tools show that it is not correct and thereafter we give a better one.

3.1 The $\mathcal{EL}^{[\,]}$ language

$\mathcal{EL}^{[\,]}$ is a simple concurrent language where the primitives corresponding to interactions with the operating system (including the constructs for the process interactions) are given as a parameter. $\mathcal{EL}^{[\,]}$ comprises usual sequential commands: $c_1; c_2$ (sequencing), $x := e$ (assignment), **while** e **do** c **end** (while), **if** e **then** c_1 **else** c_2 (conditional) and **continue** (null command). New processes (which are themselves commands) can be created with the command **create process** c.

The abstract syntax of the language is given by the following BNF-like rules.

$$\begin{array}{rcl} p & ::= & \mathbf{program}\; c \\ c & ::= & id := e \;\mid\; \mathbf{continue} \;\mid\; c_1; c_2 \;\mid\; \mathbf{if}\; e \;\mathbf{then}\; c_1 \;\mathbf{else}\; c_2 \;\mid \\ & & \mathbf{while}\; e \;\mathbf{do}\; c \;\mathbf{end} \;\mid\; \mathbf{create\ process}\; c \end{array}$$

We do not further specify the non terminals *id* and *e* defining respectively the syntax for the variable identifiers and for the expressions.

Following our method the semantics of $\mathcal{EL}^{[\,]}$ is given in two steps. We first define the underlying concurrent model of $\mathcal{EL}^{[\,]}$, formally represented by the abstract concurrent machine UACM[ ], which describes the evolutions of the $\mathcal{EL}^{[\,]}$ programs. Then we give a translation associating in a compositional way with each program of $\mathcal{EL}^{[\,]}$ a state of UACM[ ]. We omit this translation, since in this case it is not particularly interesting (and it is obvious to see which is the state of UACM[ ] associated with each program).

3.1.1 The abstract concurrent machine UACM[ ]

UACM[ ] is given following the method illustrated in section 1.1. The active components correspond to the $\mathcal{EL}^{[\,]}$ processes; while the passive component, not further defined here, is a parameter representing the information needed to handle possible additional features.

Process components of UACM[ ] The states of BTS[ ] are couples of the form $<c, s>$, where c is an element of sort *com* and s of sort *store*; c models the code to be executed by the process (a command) and s is a store, i.e. a finite map from variable identifiers into values, that models a process local memory. We do not report the full specification of the states and of the labels of BTS[ ]: just note that elements of sort *com* correspond to $\mathcal{EL}^{[\,]}$ commands (they have the same syntax without boldface). We also use the (obvious) evaluation operation $E\colon exp \times store \to val$ such that $E(e, s)$ is the result of evaluating e when the values for the variables are given by the store s. Then we give the static axioms and the dynamic rules of BTS[ ].

The execution of the command **continue** corresponds to performing an internal transition and then to evolve to a particular state unable to perform any further transition (note that, as said in section 1.1.1, τ labels internal transitions).

$$<\text{continue}, s> \xrightarrow{\tau} <\text{skip}, s>$$

The execution of an assignment $id := e$ is an internal transition that modifies the local store from s to $s[E(e, s)/id]$. Therefore the rule defining $:=$ is

$$<id := e, s> \xrightarrow{\tau} <\text{skip}, s[E(e, s)/id]>.$$

Control flow commands choose to execute a subcommand depending on the value of a boolean expression. They are described by the following static axioms,

$$\begin{array}{rcl}
E(e, s) = \text{true} & \supset & <\text{if } e \text{ then } c_1 \text{ else } c_2, s> = <c_1, s> \\
E(e, s) = \text{false} & \supset & <\text{if } e \text{ then } c_1 \text{ else } c_2, s> = <c_2, s> \\
E(e, s) = \text{true} & \supset & <\text{while } e \text{ do } c \text{ end}, s> = <c; \text{while } e \text{ do } c \text{ end}, s> \\
E(e, s) = \text{false} & \supset & <\text{while } e \text{ do } s \text{ end}, s> = <\text{continue}, s>
\end{array}$$

A sequence of commands, where the first is skip, is equivalent to the second, i.e. skip; $c = c$; otherwise, the sequence of two commands is described by the rule

$$\frac{<c_1, s> \xrightarrow{l} <c_1', s'>}{<c_1; c_2, s> \xrightarrow{l} <c_1'; c_2, s'>}.$$

The command **create process** is described by the following rule, where the operation creating: $com \to lab$ defines the corresponding labels. The actual creation of the process is handled in the synchronization step.

$$<\text{create process } c, s> \xrightarrow{\text{creating}(c)} <\text{skip}, s>$$

Synchronization We define the transition system STS[ ] following the schema of figure 2. The elements of sort *state* are couples $<pms, i>$ where pms is a group of synchronizing processes in our architecture and i, the status of the passive component, depends on the parameter. Here we report the rules defining the transitions of STS[ ], i.e. the synchronized transitions of UACM[ ].

Process internal transitions and process creation are described by the following two rules; a process creating a new process adds to the system a process having form $<c, \Lambda>$, which in turn will execute the command c over an empty store Λ.

$$\frac{p \xrightarrow{\tau} p'}{<p, i> \xRightarrow{\tau} <p', i>} \qquad \frac{p \xrightarrow{\text{creating}(c)} p'}{<p, i> \xRightarrow{\tau} <p' \mid <c, \Lambda>, i>}.$$

The global information, which depends on the parameter is not changed by the above rules. The transitions are labelled with τ: we are modeling an architecture where creation can be modelled within the synchronization step and is not visible outside of it.

Parallelism The description of parallel activities is formalized giving a new transition system PTS[ ] following the schema of figure 3. In this case we choose to forbid any parallel composition of synchronized actions, thus PTS[ ] is defined by chosing the set of rules $Ax_{PTS} = \emptyset$.

Monitoring UACM[ ] corresponds to an architecture with parallelism simulated by interleaving the synchronized actions of the component processes. If more than one synchronized action is possible, then the system may choose one of them whatever. This constraint is formally expressed by choosing the rule defining the transition relation of UACM[ ] in the following way.

$$\frac{<pms, i> \stackrel{l}{\Longrightarrow} <pms', i'>}{<pms \mid pms_1, i> \stackrel{\tau}{\Longrightarrow} <pms' \mid pms_1, i'>}$$

3.2 The $\mathcal{EL}^{[\mathrm{Sem}]}$ language

$\mathcal{EL}^{[\mathrm{Sem}]}$ is a concurrent language obtained by adding to $\mathcal{EL}^{[\,]}$ a semaphore. $\mathcal{EL}^{[\mathrm{Sem}]}$ processes can interact with the semaphore, initially set to free, using the **P** and **V** commands. The $\mathcal{EL}^{[\mathrm{Sem}]}$ abstract syntax is hence derived from that of $\mathcal{EL}^{[\,]}$ by adding two commands **P** and **V**. The underlying abstract concurrent machine UACM[Sem] is similar to UACM[ ]: the active subcomponents are the language processes and the passive component is the semaphore status (free or busy). To get the specification of UACM[Sem] we add the rules describing the execution of **P** and **V**.

Basic Transition System To describe the behaviour of a process executing either a **P** or a **V** command we introduce two new labels π, ν: $\rightarrow lab$, which are not internal transitions. Indeed the executions of **P** and **V** depend on the context in which the process is running; here we can only say that if a process executes **P** (**V**), then it has the capability of performing a transition labelled by π (ν), i.e.

$$<\mathrm{P}, s> \stackrel{\pi}{\longrightarrow} <\mathrm{skip}, s> \qquad <\mathrm{V}, s> \stackrel{\nu}{\longrightarrow} <\mathrm{skip}, s>.$$

Synchronization The semaphore status is affected by transitions labelled with ν and π. A ν transition completes regardless of the semaphore status and sets the semaphore to free. A process can perform a π transition iff the semaphore is free. This transition sets the semaphore to busy. Hence we have

$$\frac{p \stackrel{\nu}{\longrightarrow} p'}{<p, ss> \stackrel{\tau}{\Longrightarrow} <p', \mathrm{free}>} \qquad \frac{p \stackrel{\pi}{\longrightarrow} p'}{<p, \mathrm{free}> \stackrel{\tau}{\Longrightarrow} <p', \mathrm{busy}>}.$$

Parallelism and monitoring We assume that operations on the semaphore cannot proceed in parallel and that parallelism is simulated by interleaving the transitions of the processes. With these assumptions we do not need to change the UACM[ ] definitions for the parallelism and monitoring to define the corresponding steps for UACM[Sem]. Thus, if simultaneously more than one process needs to access the semaphore, the system scheduler nondeterministically chooses one of them.

Experimenting with the tools A reader with some industrial experience will argue that the above specification is of little help to the programmer in charge of developing an implementation of semaphores. The tools are an answer to this kind of criticism since they allow a programmer not acquainted with formal methods to use the $\mathcal{EL}^{[\mathrm{Sem}]}$ specification to gain an intuitive understanding of the semaphore mechanism by executing symbolically some $\mathcal{EL}^{[\mathrm{Sem}]}$ test programs.

For example, once the formal methods specialist has supplied *CLT* and *CRP* with the $\mathcal{EL}^{[\mathrm{Sem}]}$ semantic definition, the programmer is able to execute a test, launching, for example, in parallel two executions of the command: **while** true **do P** ;**V end**. That means to test the program $\mathrm{TEST}^{\mathrm{Sem}}$:

program
  create process while true **do P** ;**V end**;
  while true **do P** ;**V end**

Accordingly to the semaphore description we expect that after the execution of a **P** a process prevents the other from executing semaphore commands until it completes a subsequent **V**. The behaviour resulting from the *CRP* simulation confirms our expectation and shows that a process cannot be interrupted between a **P** and a **V**. *CRP* produces the transition tree shown in figure 5.

The tool allows us to walk on this tree inspecting the transitions. We see that during transition 0→1 the second process is created, while during cycle 1→2→1 a process performs in succession the commands **P** and **V**.

Figure 5: The execution tree of $\mathrm{TEST}^{\mathrm{Sem}}$

3.3 The $\mathcal{EL}^{[\mathrm{Bit}]}$ language

$\mathcal{EL}^{[\mathrm{Bit}]}$ is a concurrent language obtained by adding to $\mathcal{EL}^{[\,]}$ a shared bit. $\mathcal{EL}^{[\mathrm{Bit}]}$ processes can interact with the shared bit, initially set to false, using one of the commands **write**(e) (writes the value of the boolean expression e into the shared bit), **read**(id) (copies the value of the shared bit into the local variable id), and **tst**(id) (executes a *test and set* operation on the shared bit; this command corresponds to a **read**(id) immediately followed by a **write**(true)). The three commands are *not interruptable*, and two processes cannot perform simultaneously such commands; **tst**(v) is equivalent to **read**(v); **write**(true) iff no other operation on the shared bit is executed between **read**(v) and **write**(true) by another process. The test and set operation is a basic feature of a real concurrent architecture: an atomic test and set command is present in all CPU instruction sets, since it is necessary to implement operating system kernel primitives.

The $\mathcal{EL}^{[\mathrm{Bit}]}$ abstract syntax is derived from that of $\mathcal{EL}^{[\,]}$ by adding the commands **write**(e), **read**(id) and **tst**(id). The underlying abstract concurrent machine UACM[Bit] is similar to UACM[ ]; the active subcomponents are the language processes, while the passive component is the shared bit status (true or false). To get the specification of UACM[Bit] we add the following rules.

Basic Transition System

$$<\mathrm{write}(e), s> \xrightarrow{\mathrm{writing}(E(e,s))} <\mathrm{skip}, s> \qquad <\mathrm{read}(x), s> \xrightarrow{\mathrm{reading}(v)} <\mathrm{skip}, s[v/x]>$$

$$<\mathrm{tst}(x), s> \xrightarrow{\mathrm{test\&setting}(v)} <\mathrm{skip}, s[v/x]>$$

The labels defined by the operations writing, reading, test&setting: $val \to lab$ are introduced to model the capability of a process to interact with the shared bit.

Synchronization A process performing a transition labelled by "writing(b')" sets to b' the shared bit. A transition labelled by "reading(b)" starts only when the bit status is b and does not change such status. A transition labelled by "test&setting(b)" starts only when the bit status is b and sets such status to true.

$$\frac{p \xrightarrow{\mathrm{writing}(b')} p'}{<p, b> \stackrel{\tau}{\Longrightarrow} <p', b'>} \qquad \frac{p \xrightarrow{\mathrm{reading}(b)} p'}{<p, b> \stackrel{\tau}{\Longrightarrow} <p', b>} \qquad \frac{p \xrightarrow{\mathrm{test\&setting}(b)} p'}{<p, b> \stackrel{\tau}{\Longrightarrow} <p', \mathrm{true}>}$$

Parallelism and monitoring We assume that operations on the shared bit cannot proceed in parallel and that parallelism is simulated by nondeterministically interleaving the transitions of the processes. With this assumption we do not need to change the UACM[ ] definitions for the parallelism and monitoring to define the corresponding steps for UACM[Bit]. Thus, if simultaneously more than one process needs to access the shared bit, the system scheduler nondeterministically chooses one of them.

3.4 Testing Implementations of Semaphores in $\mathcal{EL}^{[\mathrm{Bit}]}$ Using the Tools

We try to implement the semaphore mechanism of $\mathcal{EL}^{[\mathrm{Sem}]}$ in $\mathcal{EL}^{[\mathrm{Bit}]}$ using the shared bit to store the status of the semaphore, the busy semaphore being the bit set to true. We propose two ways of implementing the **P**, **V** commands and using the tools we show that the first is wrong.

A First Attempt A **V** command is implemented by **write**(false); a **P** command is implemented by continuously reading and writing the shared bit till it becomes false (i.e. the semaphore is free). Hence the code of the implementation of **P** is **read**(x); **write**(true); **while** x **do** **read**(x); **write**(true) **end**.

We test the correctness of this implementation running *CRP* on the $\mathcal{EL}^{[\mathrm{Bit}]}$ program $\mathrm{TEST}^{\mathrm{Bit}}$ obtained by replacing in $\mathrm{TEST}^{\mathrm{Sem}}$ the commands **P** and **V** with their implementations. We get a transition tree showing some problems. The critical part of this tree is shown in figure 6 (the local memory is simply represented by the content of the variable x); the subtree starting from node 7 is complete.

If the system gets into state 7, then it is trapped into cycles not corresponding to perform **P**;**V**. The transitions that lead from node 0 to node 7 show how the system gets into problems. In transition 0→1 the second process is created. Transitions from 1 to 3 are executed by the first process and correspond to the

Figure 6: The critical part of the execution tree of TEST$^{\text{Bit}}$ (wrong implementation)

execution of a **P** command. Inspecting the status of the shared bit in state 3 we see that the bit is set to true. During transition 3→4 the second process reads the shared bit and stores true into the local variable x. Transition 4→5 corresponds to the execution of the **V** command by the first process. The second process gains again the control and completes the **P** command with transition 5→6, writing true in the shared bit. The next two transitions are executed by the first process. The first process is now again at the beginning of a **P** command, it stores true into its local variable x (6→7) and write true into the bit (7→8). However, in state 7 the bit is set to true and the two processes are executing the loops in the **P** command with true in the local variable x (7→8→7, 7→9→7). This is a deadlock: the two processes are both executing a **P** command and none of them will ever complete this command. This can be seen in the transition tree. Cycles round state 7 show that in this state the two processes will keep on reading and writing the shared bit.

We ask the reader to reflect a little on the example shown, tracing the execution of the two concurrent processes. It is clear that this implementation does not satisfy our requirement. Indeed the system may enter a state in which all processes are cycling and waiting for ever the semaphore becoming free. This happens when the semaphore is busy (i.e. the shared bit is set to true), a process executing a **P** command is interrupted after the **read**(x) operation by a **V** command and after the **V** command the interrupted process resumes its execution. When this happens, the resumed process enters an infinite loop and all processes trying to gain the resource by a **P** command and will be locked out in the same way. This results in a deadlock that is difficult to trace using conventional debuggers. Bugs, like the one described above, are quite serious and too subtle to be discovered during testing; *CRP* can hence be used to discover bugs in critical concurrent software by *formal testing*.

A Second Attempt The **V** command is implemented as in the previous attempt, while the **P** command is implemented by the sequence of commands **tst**(x); **while** x **do** **tst**(x) **end** Using this solution, the process that attempts a **P** on a busy semaphore enters a *busy waiting* cycle continuously executing the test and set operation. When an access *conflict* takes place, i.e. the two processes simultaneously start a **P** command, the process that completes the **P** command gains the resource while the other is *delayed* continuously testing the shared bit. This solution is similar to the first proposed, but here the sequence **read**(x); **write**(true) is performed by a **tst**(x) operation. This removes the problem of the previous implementation, since it is not possible to interrupt a process performing a **P** command between **read**(x) and **write**(true). Obviously this mechanism may be unfair, since the delayed process may be continuously *locked out* while the other continuously gains the resource, but this behaviour is allowed by the specification of section 3.2.

To have some hints on the behaviour of this implementation, we submit to *CRP* the TEST$^{\text{Bit}}$ program obtained by replacing in TEST$^{\text{Sem}}$ the **P** and **V** commands with their new implementations. Now the subpart of the transition tree corresponding to what shown shown in the previous section in figure 5 is reported at the lefthand side of figure 7. The first transition (0→1) creates the second process. During transition 1→2

Figure 7: The critical part of the execution tree of $\mathrm{TEST}^{\mathrm{Bit}}$ (right implementation)

the first process executes a **P** command. In transition 2→3 the second process starts a **P** command and it is locked out in state 3, continuously executing a **tst** command (3→3). Cycle 4→3→4 corresponds to a **P**;**V** sequence executed by a process while the other is delayed. Transition 4→6 is performed by the delayed process when it completes the **P** command. Similarly cycle 5→6→5 represents the **P**;**V** sequence executed when no conflict is present, while transition 6→3 is the first **tst** command of a busy waiting cycle.

The above graph becomes much more simpler if we forget local memory status mapping every process $< c, s >$ into c: states 1,4,5 and 2,3,6 becomes equivalent and the tree reduces to the one at the righthand side of figure 7 This tree is quite close to our intuition; transition 0→1 again represents process creation. During cycle 1→2→1 a process gets the resource and performs the sequence corresponding to **P**;**V**, while the other may be delayed testing the shared bit cycling round node 2.

These outcomes show that our implementation behaves correctly on this test independently of the non-determinism introduced by concurrency. Again we note that such a degree of confidence on a single test, although highly desirable, cannot be reached for concurrent systems by usual testing methods. Hence testing with *CLT* and *CRP* provides a real benefit in term of system reliability.

References

[AR1] Astesiano E.; Reggio G. "SMoLCS Driven-Concurrent Calculi", *Proc. TAPSOFT'87, Vol. I*, LNCS 249, 1987.

[AR2] Astesiano E.; Reggio G. "Direct Semantics for Concurrent Languages in the SMoLCS Approach", *IBM Journal of Research and Development*, 31, 5 (1987), pp. 512-534.

[AR3] Astesiano E.; Reggio G. *A Structural Approach to the Formal Modelization and Specification of Concurrent Systems*, Technical Report n. 0, Formal Methods Group, University of Genova, 1990.

[GM] Goguen J.; Meseguer J. "Models and Equality for Logical Programming", *Proc. of TAPSOFT '87, Vol. II*, LNCS 250, 1987.

[GiMo] Giovini A.; Morando F. "How to Use Effectively Equational Logic to Integrate Logic and Functional Programming", *Rivista Informatica* vol. XIX, n. 3, 1989.

[H] Hussman H. " Unification in Conditional Equational Theories", *EUROCAL 85*, LNCS 204, 1985.

[M] Milner R. "Operational and algebraic semantics of concurrent processes", *Handbook of Theoretical Computer Science , Vol. B: Formal Models and Semantics*, pp.1201-1242, Elsevier, 1990.

[P] Plotkin G. *A structural approach to operational semantics*, Lecture notes, Aarhus University, 1981.

Contents