Specification and Analysis of Asynchronous Systems using CADP

Radu Mateescu

INRIA / VASY project-team
Faculté des Sciences Mirande, bât. LE2I, F-21000 Dijon, France
Radu.Mateescu@inria.fr

Abstract. The design of complex industrial critical systems involving asynchronous parallelism requires the use of formal methods, assisted by appropriate verification tools, in order to detect and correct errors as early as possible. In this paper, we illustrate the use of the CADP toolbox for the formal modeling and verification of such systems by considering as an example a unit dedicated to the drilling of metal products. We describe in the LOTOS language two different versions of the unit, supervised by a sequential and a parallel controller, respectively. Then, we perform the generation and minimisation of the two underlying state spaces, and also the inspection (*visual checking*) of the smaller one, corresponding to the version equipped with a sequential controller. Finally, we analyse the behaviour of the two versions of the drilling unit by means of two complementary verification methods, based on bisimulations (*equivalence checking*) and temporal logics (*model checking*).

1 Introduction

Industrial systems that involve asynchronous parallelism, such as communication protocols, embedded systems, and multiprocessor hardware architectures, exhibit complex behaviours. Besides, systems of this kind are often critical, since their malfunctioning may entail the loss of human lives or important damages. In order to detect possible errors as early as possible in the development cycle of these systems, the usage of formal specification and verification methods, assisted by suitable software tools, becomes mandatory.

The so-called model-based verification method offers a good cost-performance tradeoff, which explains its successful application in industry. This method consists of building, from a formal description of the system, a semantic model (state space) on which the correctness properties, expressed using a suitable formalism (automata, temporal logics), are verified automatically by means of specialised algorithms. Although limited to finite-state systems, model-based verification allows to detect errors in complex systems rapidly and economically, being particularly useful during the first phases of the design process, when errors are likely to occur more frequently.

Complex industrial systems often contain asynchronous parts, consisting of several entities physically distributed that communicate and synchronize by exchanging messages, as well as "hard" real-time parts, governed by strong temporal constraints (e.g., delays). In this paper, we focus on the asynchronous aspects only; the modelling and analysis of hard real-time aspects can be carried out using specific techniques and tools [5, 8, 36].

The CADP toolbox [17] for the engineering of asynchronous systems offers a large spectrum of functionalities, which assist the design process effectively: specification, simulation, rapid prototyping, verification and test generation. The underlying tools have been designed following a modular architecture, centered around the generic OPEN/CÆSAR [14] environment for on-the-fly state space exploration, which ensures language-independence and favors the reuse of components due to well-defined software interfaces. Although several functionalities for performance evaluation have been recently added to CADP [15], we illustrate here only functional verification, which takes into account the logical ordering of events during the execution of the system.

We demonstrate the specification and analysis methodology promoted by CADP by considering the example of an industrial critical system dedicated to the drilling of metallic products. More precisely, we detail the design of the software controller in charge of driving the various physical devices that compose the drilling unit. This system has been studied as a common example for comparing the suitability of several specification languages and the power of their associated verification tools [7, 6].

The paper is organized as follows. Section 2 briefly describes the LOTOS language and several verification tools of CADP used for this case-study. Next, Section 3 details the specification in LOTOS of the drilling unit and Section 4 presents its functional verification by means of bisimulations and temporal logics. Finally, Section 5 concludes the paper and gives some current research directions in the field of model-based verification techniques.

2 The CADP toolbox

CADP[1] (*Construction and Analysis of Distributed Processes*) [17] is a very efficient toolbox for the specification and verification of parallel asynchronous systems. These systems consist of several entities (processes or agents) that run in parallel, synchronize and communicate by message-passing. To model the execution of these systems, CADP uses the interleaving semantics, which relies upon the fact that each action (or event) is atomic and only one action can be observed at a given moment. Examples of asynchronous systems are: telecommunication protocols, operating systems, distributed databases, multiprocessor architectures, embedded softwares, etc.

2.1 The LOTOS language

CADP accepts as input several specification formalisms, ranging from high-level languages, such as LOTOS, to lower-level languages, such as networks of commu-

[1] See `http://www.inrialpes.fr/vasy/cadp`

nicating automata. LOTOS (*Language Of Temporal Ordering Specification*) [23] is a formal description technique standardised by ISO. Although initially defined for the description of communication protocols according to the OSI model, the LOTOS language has revealed equally suitable for describing other classes of asynchronous systems, such as those aforementioned. LOTOS consists of two "orthogonal" parts:

A "data" part, based on algebraic abstract data types, and more specifically on the ACTONE language [11]. This part allows to describe the data structures handled by the system by means of sorts and algebraic operations, defined using equations and pattern-matching;

A "control" part, which combines the best primitives of the process algebras CCS [33] and CSP [9]. This part allows to describe the parallel processes composing the system as terms constructed by applying algebraic operators (action prefix, choice, parallel composition with handshake synchronisation, hiding, etc.).

CADP contains two compilers for LOTOS: CÆSAR.ADT [13] translates the data part of a LOTOS specification in C (the LOTOS sorts and operations are translated as C types and functions, respectively) and CÆSAR [20] translates the control part into a C program that can be embedded in a real system or used for simulation, verification or test generation.

2.2 Labelled transition systems

The labelled transition systems (LTSs) are the semantic model underlying the specification formalisms used by CADP. An LTS M is a quadruple (S, A, T, s_0) consisting of a set of states S, a set of actions (transition labels) A, a transition relation $T \subseteq S \times A \times S$, and an initial state $s_0 \in S$. A special invisible action $\tau \notin A$ allows to model the internal (unobservable) activity of the system. A transition $(s_1, a, s_2) \in T$, also noted $s_1 \xrightarrow{a} s_2$, indicates that the system can move from state s_1 to state s_2 by performing action a. CADP provides two complementary representations for LTSs:

An explicit representation as the list of transitions contained in the LTS, stored in a file of a compact binary format called BCG (*Binary Coded Graphs*). The BCG environment offers a set of tools and libraries for the manipulation of BCG files (reading/writing, graphical visualisation, minimisation, conversion to other formats, etc.). This explicit representation of LTSs is suitable for global (or enumerative) verification, which proceeds by a backward exploration of the transition relation and therefore requires the prior construction of the entire LTS;

An implicit representation as the successor function of the LTS, encoded as a C program conforming to the application programming interface defined by the OPEN/CÆSAR environment [14]. Besides the C types implementing the LTS states, actions, and transitions, equipped with basic operations (comparison, hashing, enumeration of successor states, etc.), OPEN/CÆSAR offers also libraries dedicated to the on-the-fly exploration of LTSs (state tables, transition lists, stacks, etc.). This implicit representation of LTSs is suitable for local (or on-the-fly) verification, which proceeds by a forward exploration of the transition relation and therefore allows an incremental construction[2] of the LTS.

The on-the-fly verification is a simple manner of combating state explosion (prohibitive size of the LTS for systems containing many parallel processes and complex data types), allowing to detect errors even when the complete construction of the LTS exceeds the available computing resources.

2.3 Some verification tools

CADP offers a large palette of tools dedicated to the analysis of LTSs, covering the whole spectrum of functionalities necessary to assist the design process: interactive and guided simulation, random execution, minimisation modulo various equivalence relations, partial order reduction, equivalence checking and model checking, conformance test case generation. Here we briefly present some of the CADP tools that we used for this case-study and whose functioning will be illustrated in the sequel.

Bcg_Min performs the minimisation of an LTS, represented as a BCG file, modulo various equivalence relations, such as strong bisimulation or branching bisimulation. BCG_MIN also handles probabilistic and stochastic LTSs [15].

Cæsar_Solve [29, 31] is a generic software library of the OPEN/CÆSAR environment, dedicated to the on-the-fly verification of alternation-free boolean equation systems (BESs). These BESs consists of several blocks of equations with boolean variables in their left-hand sides and propositional formulas (disjunctions or conjunctions over boolean variables) in their right-hand sides. Each equation block represents the minimal or the maximal fixed point of the functional taking as input the variables present in the left-hand sides of equations and returning the values of the formulas in the right-hand sides of equations.

A boolean variables defined by an equation depends upon the variables occurring in the right-hand side of that equation. A block depends upon another block if it defines a variable depending upon some variable of the other block; the alternation-free condition means the absence of cyclic dependencies between the blocks of a BES. This class of BESs benefits from resolution algorithms having a time and space complexity linear in the size of the BES (number of variables and boolean operators), still being sufficiently general to represent several types of LTS analyses (equivalence checking, model checking, partial order reduction, etc.).

[2] Of course, an explicit LTS already constructed can also be explored on-the-fly; within CADP, this is done by means of the BCG_OPEN tool, which implements a representation of BCG files compatible with the OPEN/CÆSAR interface.

The on-the-fly resolution of a Bes consists in computing the value of a boolean variable of interest by exploring incrementally only the part of the Bes necessary to determine the value of that variable. The underlying algorithms can be developed in a more intuitive way by representing Bess as *boolean graphs* [1] whose vertices and edges denote boolean variables and dependencies between them, respectively. Reformulated in this context, resolution algorithms perform a forward exploration of the boolean graph, starting at the variable of interest, intertwined with a backward propagation of stable variables (whose values have been determined) along dependencies.

Cæsar_Solve currently provides four on-the-fly resolution algorithms having a linear complexity in the size of the Bes. Algorithms A1 and A2, based respectively on depth-first and breadth-first traversals of the boolean graph, can solve general Bess, without imposing constraints on the boolean formulas in the right-hand sides of the equations. Algorithms A3 and A4, based on depth-first traversals, are optimised to solve acyclic and disjunctive/conjunctive Bess (frequently encountered in practice) with a lower memory consumption, by storing only boolean variables and not the dependencies between them. All these algorithms also produce *diagnostics*, i.e., portions of the boolean graph illustrating the result of the resolution, following the approach proposed in [28]. Due to the breadth-first traversal, algorithm A2 exhibits small-depth diagnostics, which are easier to interpret.

Cæsar_Solve defines an implicit representation of boolean graphs by means of an application programming interface in C similar to the interface for Ltss defined by Open/Cæsar: the C type encoding the boolean variables is equipped with primitives allowing to explore the boolean graph (comparison and hashing of variables, enumeration of successor variables, etc.). Diagnostics of resolutions are also produced as boolean subgraphs represented implicitly as their successor function. Cæsar_Solve is currently used within Cadp as computing engine for several on-the-fly verification tools, two of them being presented below.

Bisimulator [29, 31] is an equivalence checker that compares on-the-fly two Ltss w.r.t. an equivalence or preorder relation. The first Lts, represented implicitly as an Open/Cæsar program, denotes the behaviour of a system (*protocol*), whereas the second Lts, represented explicitly as a Bcg file, denotes the external behaviour (*service*) expected for the system. Bisimulator implements seven equivalence relations between Ltss: four bisimulations (strong, branching, observational, and $\tau^*.a$) and three simulation equivalences (safety, trace, and weak trace), being one of the richest on-the-fly equivalence checkers currently available. For each relation, the tool can determine both the equivalence of Ltss and the inclusion of one Lts into the other modulo the corresponding preorder.

The method used by Bisimulator consists in reformulating the verification problem as the resolution of a Bes containing a single block of maximal fixed point equations, directly derived from the mathematical definition of the equivalence relation. The tool is structured in two independent parts: a front-end in charge of translating the comparison modulo an equivalence in terms of a Bes and of interpreting the diagnostic of the resolution in terms of the two Ltss being compared; and a back-end (CÆSAR_SOLVE library) responsible for computing the variable of interest, which represents the fact that the initial states of the two LTSs are equivalent or related modulo the preorder considered.

This modular architecture facilitates the addition of new equivalence relations (each relation is implemented as a separate module containing the BES translation and the diagnostic interpretation) and does not penalize performance, BISIMULATOR competing favourably with other implementations of algorithms dedicated to on-the-fly equivalence checking [4]. The diagnostics (counterexamples) issued by the tool when the LTSs are not equivalent (or not included) are acyclic graphs containing all the sequences that, simultaneously executed in the two LTSs, lead to non equivalent states.

BISIMULATOR employs all the resolution algorithms provided by CÆSAR_SOLVE: A1 and A2 can be applied to all equivalences (A2, being based on a breadth-first traversal, has the practical advantage of exhibiting small-depth counterexamples); A3, optimised in memory for solving acyclic BESs, serves to verify the inclusion of execution sequences or trees in an LTS; and A4, optimised in memory for solving conjunctive BESs, is useful when one LTS is deterministic (for strong bisimulation) and does not contain invisible transitions (for weak equivalences).

Evaluator 3.5 [29, 31] is a model checker that evaluates on-the-fly a temporal logic formula on an LTS. The logic accepted as input is the regular alternation-free μ-calculus [32], which consists of boolean operators, possibility and necessity modalities containing regular expressions over action sequences (similar to those of PDL [12]), and fixed point operators of modal μ-calculus [26]. The alternation-free condition, meaning the absence of mutual recursion between minimal and maximal fixed point operators, leads to verification algorithms having a linear complexity w.r.t. the size of the formula (number of operators) and the LTS (number of states and transitions). The regular alternation-free μ-calculus enables a concise and intuitive description of classical properties over LTSs (safety, liveness, as well as certain forms of fairness). The tool also allows to define reusable libraries containing derived temporal operators, such as those of ACTL (*Action-Based* CTL) [34] and the generic property patterns proposed in [10].

The method used by EVALUATOR 3.5 consists in reformulating the verification problem as the resolution of a BES containing an equation block for each temporal operator contained in the formula. The BES is obtained after several transformation phases applied to the formula (translation in positive normal form, elimination of derived operators, translation to modal equation systems, elimination of regular expressions contained in modalities, simplification).

The tool is structured in two independent parts: a front-end in charge of translating the evaluation of the formula in terms of a BES and of interpreting the diagnostic of the resolution in terms of the LTS being analyzed; and a back-end (CÆSAR_SOLVE library) responsible for computing the variable of interest, which represents the fact that the initial state of the LTS satisfies the formula. The diagnostics (examples and counterexamples) issued by the tool are subgraphs of the LTS illustrating the truth value of the formula on the initial state of the LTS.

EVALUATOR 3.5 employs all the resolution algorithms provided by CÆSAR_SOLVE: A1 and A2 can be applied to all formulas of regular alternation-free μ-calculus (A2, being based on a breadth-first traversal, has the practical advantage of exhibiting small-depth diagnostics); A3, optimised in memory for solving acyclic BESs, serves to verify any μ-calculus formula on LTSs representing execution or simulation scenarios; and A4, optimised in memory for solving disjunctive/conjunctive BESs, is applied for evaluating formulas of ACTL and PDL, frequently encountered in practice.

3 Specification of a drilling unit

We develop in this section a LOTOS specification of a drilling unit for metallic products. This example of industrial critical system [7, 6] has served as support for experimenting the modelling capabilities of various description languages (χ [38], Promela [22], timed automata [3], μCRL [21]) and the functionalities of their associated verification tools (SPIN [22], UPPAAL [3], CADP). The LOTOS specification presented below was derived from the χ description initially proposed in [7], with some details (e.g., the presence of the `TT3` sensor) inspired from the more elaborated description given in [6]. The drilling unit, illustrated in Figure 1, consists of a turning table, a drill equipped with a clamp, and a tester.

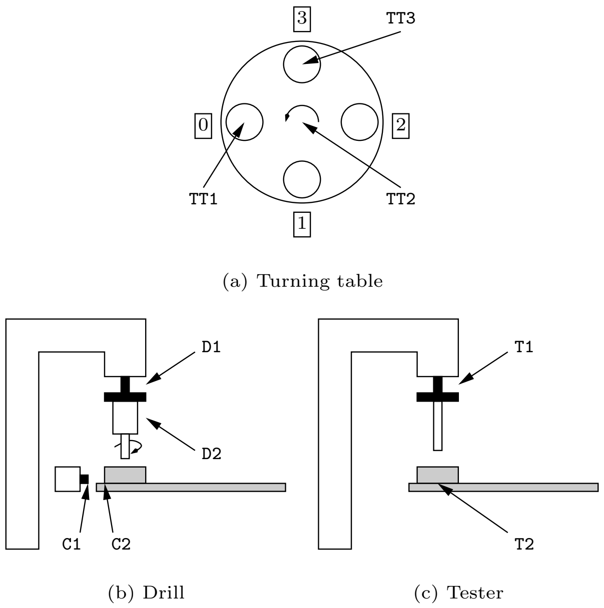

Fig. 1. Drilling unit for metallic products

Table 1. Dialog between local controllers and physical devices

Gates for sending commands to actuators		
Device	Gate	Function
Table	`TurnOn`	Starts a 90 ° rotation
Clamp	`COnOff`	Blocks or releases the clamp
Drill	`DOnOff`	Starts or stops the engine of the drill
	`DUpDown`	Starts the ascending or descending movement
Tester	`TUpDown`	Starts the ascending or descending movement

Gates for receiving signals from sensors		
Device	Gate	Function
Table	`TT1`	Product present at position 0
	`TT2`	Rotation of 90 ° completed
	`TT3`	Product absent at position 3
Clamp	`C1`	Clamp released
	`C2`	Clamp blocked
Drill	`D1`	Drill in upper position
	`D2`	Drill in lower position
Tester	`T1`	Tester in upper position
	`T2`	Tester in lower position

The turning table (a) transports the metallic products to the drill and the tester. It is circular and has four slots, each one possibly containing at most one product. Each slot can be in one of four positions: entry position (0), drilling position (1), testing position (2), and exit position (3). Three sensors `TT1`, `TT2`, and `TT3` attached to the table indicate if a product is present in the slot at position 0, if the table just completed a 90 ° counterclockwise rotation, and if the slot at position 3 is empty, respectively.

The drill (b), located at position 1, is equipped with a clamp allowing to lock the product during the drilling operation. Two sensors `D1` and `D2` attached to the drill detect whether it is in the upper or lower position, respectively. Two other sensors `C1` and `C2` attached to the clamp indicate whether it is released or blocked, respectively.

The tester (c), located at position 2, serves to detect whether a product was correctly drilled or not. It is equipped with two sensors `T1` and `T2`, which detect if the tester is in the upper or lower position, respectively. If the tester is in the lower position, this means that either the product present in the slot at position 2 was correctly drilled, or there is no product in this slot.

Each physical device (turning table, clamp, drill, tester) is equipped with a local controller in charge of driving the device. Local controllers receive signals from the sensors and send commands to the actuators attached to physical

Table 2. Dialog of the main controller with the local controllers and the environment

Gate	Signal	Meaning
CMD	`Turn`	Rotation of 90 ° of the table
	`Drill`	Drilling of the product at position 1
	`Lock`	Blocking of the clamp
	`Unlock`	Release of the clamp
	`Test`	Test of the product at position 2
INF	`Turned`	Rotation of 90 ° of the table completed
	`Present`	Product present at position 0
	`Drilled`	Drilling of the product at position 1 completed
	`Locked`	Clamp blocked
	`Unlocked`	Clamp released
	`Tested`	Product at position 2 tested
	`Absent`	Product absent at position 3
REQ	`Add`	Input of a product at position 0
	`Remove`	Output of a product at position 3

devices. Table 1 indicates the communication channels (called *gates* in LOTOS) used by local controllers.

The turning table is controlled through the gate `TurnOn`, which commands a 90 ° counterclockwise rotation. Thus, the products are transported from the entry position to the drilling position, then to the testing position, and finally to the exit position. The clamp, the drill, and the tester are controlled through gates that model *switching mode* commands, so-called because they have two different effects, which change at each new invocation. For example, the command `COnOff` triggers either the blocking of the clamp if it is released, or the release of the clamp if it is blocked.

The global functioning of the drilling unit is handled by a main controller, which is responsible for coordinating the activity of the various devices and for interacting with the environment. The main controller communicates with the local controllers associated to the physical devices through the gates `CMD` (emission of commands) and `INF` (reception of information) and with the environment through the gate `REQ` (emission of requests). Table 2 indicates the various signals (denoted by values of en enumerated type `Sig`) sent on these gates.

The specification assumes that the environment reacts correctly to the requests issued by the main controller: in particular, the products are put at position 0 and got at position 3 after every `Add` and `Remove` signal, respectively.

3.1 Architecture

The architecture of the system specified in LOTOS is illustrated in Figure 2. Boxes represent parallel processes and arrows indicate communication gates.

Each physical device and its associated local controller are modelled as couples of processes: TT and TTC (turning table), D and DC (drill), C and CC (clamp), T and TC (tester). The main controller is modelled by the process MC.

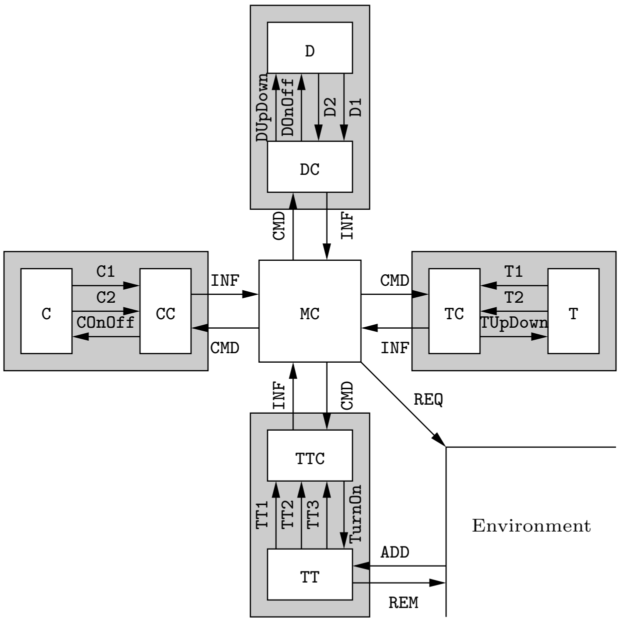

Fig. 2. Architecture of the drilling unit specified in LOTOS

The process TT communicates with the environment through the gates ADD and REM, which model the input of a product in the slot at position 0 and the output of the product from the slot at position 3, respectively.

The LOTOS description of the architecture is illustrated below. Each element is represented by a process call parameterised by communication gates and possibly by data values. The concurrent execution of processes is described using the parallel composition operators "|||" and "|[...]|" of LOTOS, which denote parallel execution without synchronisation and with synchronisation on a set of gates, respectively. For instance, the parallel processes TT and TTC synchronise on the gates TT1, TT2, TT3, and TurnOn, but they execute asynchronously w.r.t. the other processes D, DC, etc. The gates used for the dialog between physical devices and their local controllers are hidden (i.e., renamed into the invisible action, noted "i" in LOTOS) using the "**hide**" operator.

The processes TT, D, C, and T representing the physical devices have data parameters (whose meaning will be defined in the following sections) recording the current state of these processes: when the system starts its execution, the values of these parameters indicate that all slots of the turning table are empty, the drill is stopped and in the upper position, the clamp is released and the tester is in the upper position. The process MC modelling the main controller also has data parameters reflecting the current state of the system.

```
( (  hide TT1, TT2, TT3, TurnOn, D1, D2, DUpDown, DOnOff,
          C1, C2, COnOff, T1, T2, TUpDown in
      (  (
            TT [TT1, TT2, TT3, TurnOn, ADD, REM]
               (false, false, false, false)
            |[TT1, TT2, TT3, TurnOn]|
            TTC [TT1, TT2, TT3, TurnOn, INF, CMD]
         )
         |||
         (
            D [D1, D2, DUpDown, DOnOff] (false, true)
            |[D1, D2, DUpDown, DOnOff]|
            DC [D1, D2, DUpDown, DOnOff, INF, CMD]
         )
         |||
         (
            C [C1, C2, COnOff] (false)
            |[C1, C2, COnOff]|
            CC [C1, C2, COnOff, INF, CMD]
         )
         |||
         (
            T [T1, T2, TUpDown] (true)
            |[T1, T2, TUpDown]|
            TC [T1, T2, TUpDown, INF, CMD]
         )
      )
   )
   |[INF, CMD]|
   MC [REQ, INF, CMD] (false, false, false, false, false)
)
|[REQ, ADD, REM]|
Env [REQ, ADD, REM, ERR]
```

3.2 Physical devices and local controllers

We give below the Lotos specification of the various physical devices and of the local controllers that drive their functioning. For simplicity, the processes corresponding to the local controllers are illustrated graphically in Figure 3 (the initial states are marked with bold lines).

Turning table The process TT has four boolean parameters p0, p1, p2, and p3, which are set to true if a product is present in the slot at the corresponding position of the turning table and are set to false otherwise. At any time, TT can perform one of the following behaviours: it receives a rotation command from its local controller, it carries out the rotation (this is not explicitly modelled here[3]), and then sends back to the controller the corresponding response; if the slot at position 0 of the table is empty, it can input a product from the environment and signal to its controller that the slot at position 0 became occupied; if the slot at position 3 of the table is occupied, it can output the corresponding product to the environment. After performing one of these behaviours, TT updates its parameters and continues its execution cyclically.

```
process TT [TT1, TT2, TT3, TurnOn, ADD, REM] (p0, p1, p2, p3:Bool) :
          noexit :=
   TurnOn;
      TT2;
         TT [TT1, TT2, TT3, TurnOn, ADD, REM] (p3, p0, p1, p2)
   []
   [not (p0)] -> ADD;
      TT1;
         TT [TT1, TT2, TT3, TurnOn, ADD, REM] (true, p1, p2, p3)
   []
   [p3] -> REM;
      TT3;
         TT [TT1, TT2, TT3, TurnOn, ADD, REM] (p0, p1, p2, false)
endproc
```

The process TTC (see Figure 3) executes the following session cyclically: when the process TT informs it about the presence of a product in the slot at position 0 of the table, it passes this information to the main controller; when it receives a rotation command from the main controller, it transmits this command to the process TT, waits for its response, and then passes it to the main controller; finally, when the process TT informs it about the presence of a product in the slot at position 3 of the table, it transmits this information to the main controller.

Clamp The process C has one boolean parameter locked, which is set to true when the clamp is blocked and to false otherwise. C executes the following behaviour cyclically: it receives a block or release command from its local controller, executes it (this is not explicitly modelled here) and, according to its current state, send the appropriate response to the controller.

```
process C [C1, C2, COnOff] (locked:Bool) : noexit :=
   COnOff;
   (  [locked] -> C1;
         C [C1, C2, COnOff] (not (locked))
      []
      [not (locked)] -> C2;
         C [C1, C2, COnOff] (not (locked))
   )
endproc
```

[3] In a system specification that takes into account the real-time aspects, the actions TurnOn and TT2 (as well as the commands sent to the actuators and the responses of the sensors attached to the other physical devices) would be separated by a delay.

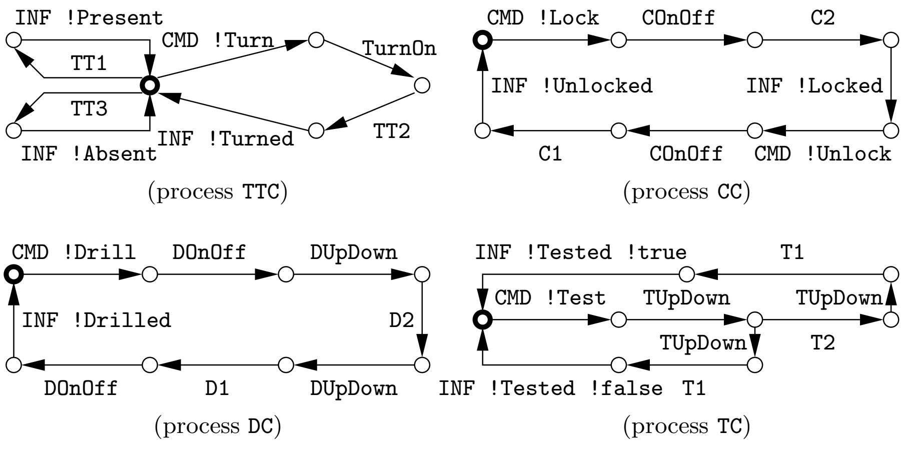

Fig. 3. LTSs modelling the behaviour of local controllers

The process `CC` (see Figure 3) executes the following session cyclically: when it receives a block command from the main controller, it commands the blocking of the clamp, waits for the response from the process `C`, and then transmits this response to the main controller; then, it waits for a release command from the main controller, it commands the release of the clamp, waits for the response from the process `C`, and propagates it to the main controller.

Drill The process `D` has two boolean parameters `on` and `up`, which are set to `true` if the drill is started and in the upper position, and are set to `false` otherwise. At any time, `D` can perform one of the following behaviours: it receives from its local controller a start or stop command; if the drill is started, it receives a command for moving up or down, executes it (this is not explicitly modelled here), and sends back the appropriate response to its controller. After one of these behaviours, `D` updates its parameters and continues its execution cyclically.

```
process D [D1, D2, DUpDown, DOnOff] (on, up:Bool) : noexit :=
   DOnOff;
      D [D1, D2, DUpDown, DOnOff] (not (on), up)
   []
   [on] -> (
      DUpDown;
      (  [up] -> D2;
            D [D1, D2, DUpDown, DOnOff] (on, not (up))
         []
         [not (up)] -> D1;
            D [D1, D2, DUpDown, DOnOff] (on, not (up))
      )
   )
endproc
```

The process DC (see Figure 3) executes the following session cyclically: when it receives a drill command from the main controller, it commands the start of the drill engine, then the descending movement of the drill, and waits for the response of the process D; then, it commands the ascending movement of the drill, waits for the response from D, then commands the engine to stop; finally, it sends to the main controller the response meaning that the drilling operation has been accomplished.

Tester The process T has a boolean parameter up, which is set to true if the tester is in the upper position and is set to false otherwise. T performs the following behaviour cyclically: it receives a command for moving up or down from its local controller; it executes the command (this is not explicitly modelled here) and, according to its current state, sends an appropriate response to its controller. The fact that the product located in the slot at position 2 of the table was correctly drilled or not is modelled as a nondeterministic choice consisting in sending the response to the controller (product correctly drilled) or sending no response (product incorrectly drilled).

```
process T [T1, T2, TUpDown] (up:Bool) : noexit :=
   TUpDown;
   (  [up] -> (
         T2;                                (* Correct drilling *)
            T [T1, T2, TUpDown] (not (up))
         []
         T [T1, T2, TUpDown] (not (up))     (* Incorrect drilling *)
      )
      []
      [not (up)] -> T1;
         T [T1, T2, TUpDown] (not (up))
   )
endproc
```

The process TC (see Figure 3) executes the following session cyclically: when it receives a test command from the main controller, it commands the descending movement of the tester, then it waits for the response from the process T (an absence of response indicates a product incorrectly drilled); afterwards, it sends an ascending movement command to the tester, then it waits for the response from the process T; finally, it sends an appropriate response to the main controller, indicating by means of a boolean value true or false that the product in the slot at position 2 of the table was correctly drilled or not.

3.3 Main controller — sequential version

We describe below a first version of the main controller, in which the various phases of dialog with the local controllers are performed sequentially, one after the other. The process MC has five boolean parameters: p0, p1, p2, and p3 indicate the presence of products in the slots at the corresponding positions of the turning table; `tr` is set to `true` if a product correctly drilled occupies the slot at position 3 and is set to `false` otherwise. `MC` performs the following session cyclically, consisting of five control phases executed one after the other:

1. If the slot at position 0 of the table is empty, it sends a product input request to the environment (which is supposed to react immediately by supplying a product), then it waits for the appropriate response from the controller `TTC`;
2. If a product is present in the slot at position 1, it sends a clamp block command to the controller `CC` and waits for its response, then it sends a drill command to the controller `DC` and waits for its response, and finally it sends a clamp release command and waits for the response of the controller `CC`;
3. If a product is present in the slot at position 2, it sends a test command to the controller `TC` and then it waits for the corresponding response;
4. If a product is present in the slot at position 3, it sends a product output request to the environment (which is supposed to react immediately by removing the product), then it waits for the appropriate response from `TTC`;
5. It sends a table rotation command to the controller `TTC`, then it waits for the response before updating its parameters and restarting its cyclic execution.

The chaining of the five control phases is modelled using the LOTOS sequential composition operators: "**exit**" and "**exit** (V_1, ..., V_n)" denote the termination of a behaviour without returning any result and by returning values V_1, ..., V_n, respectively; "B_1 `>>` B_2" expresses the execution of behaviour B_2 after the execution of B_1 has successfully terminated by an "**exit**"; and "B_1 `>>` **accept** x_1:T_1, ..., x_n:T_n **in** B_2" denotes the execution of B_2 after B_1 successfully terminated its execution by performing an "**exit** (V_1, ..., V_n)" of which the resulting values are assigned to the variables x_1, ..., x_n, subsequently used by B_2.

```
process MC [REQ, INF, CMD] (p0, p1, p2, p3, tr:Bool) : noexit :=
   (  [not (p0)] -> REQ !Add; INF !Present;
         exit (true)
      []
      [p0] -> exit (p0)
   )
   >>
   accept new_p0:Bool in
   (  (  [p1] -> CMD !Lock; INF !Locked;
            CMD !Drill; INF !Drilled;
               CMD !Unlock; INF !Unlocked;
                  exit
         []
         [not (p1)] -> exit
      )
      >>
      (  [p2] -> CMD !Test; INF !Tested ?r:Bool;
            exit (r)
         []
         [not (p2)] -> exit (tr)
      )
```

```
      >>
      accept new_tr:Bool in
      (  (  [p3] -> REQ !Remove !tr; INF !Absent;
               exit (false)
            []
            [not (p3)] -> exit (p3)
         )
         >>
         accept new_p3:Bool in
            CMD !Turn; INF !Turned;
               MC [REQ, INF, CMD] (new_p3, new_p0, p1, p2, new_tr)
      )
   )
endproc
```

3.4 Main controller — parallel version

We describe below a second version of the main controller, in which the phases of dialog with the local controllers are performed in parallel. The behaviour of this version is more complex (because of the interleavings of the various control phases), but also more efficient than the sequential version, the complete processing of a product being faster (this can be confirmed by evaluating the throughput of the turning table, following the approach proposed in [15]). The new process `MC` is obtained by composing the first four control phases (associated to the positions of the turning table) described in the previous section, by using the asynchronous parallel composition operator "`|||`" of LOTOS.

```
process MC [REQ, INF, CMD] (p0, p1, p2, p3, tr:Bool) : noexit :=
   (  (  [not (p0)] -> REQ !Add; INF !Present;
            exit (true, p1, p2, any Bool, any Bool)
         []
         [p0] -> exit (p0, p1, p2, any Bool, any Bool)
      )
      |||
      (  [p1] -> CMD !Lock; INF !Locked;
            CMD !Drill; INF !Drilled;
               CMD !Unlock; INF !Unlocked;
                  exit (any Bool, p1, p2, any Bool, any Bool)
         []
         [not (p1)] -> exit (any Bool, p1, p2, any Bool, any Bool)
      )
      |||
      (  [p2] ->  CMD !Test; INF !Tested ?r:Bool;
            exit (any Bool, p1, p2, any Bool, r)
         []
         [not (p2)] -> exit (any Bool, p1, p2, any Bool, tr)
      )
      |||
      (  [p3] -> REQ !Remove !tr; INF !Absent;
```

```
            exit (any Bool, p1, p2, false, any Bool)
          []
          [not (p3)] -> exit (any Bool, p1, p2, p3, any Bool)
        )
    )
    >> accept new_p0, new_p1, new_p2, new_p3, new_tr:Bool in
       CMD !Turn; INF !Turned;
          MC [REQ, INF, CMD] (new_p3, new_p0, new_p1, new_p2, new_tr)
endproc
```

The execution of each phase changes the state of the turning table, which is recorded by the boolean parameters `p0`, `p1`, `p2`, `p3`, and `tr` of the main controller. These changes are modelled using the "**exit**" operator, which allows to indicate, for each parameter, either its new value obtained after executing the corresponding processing phase, or the fact that the parameter is modified by another phase (pattern "**any**"). The occurrences of the "**exit**" operator at the end of the four parallel phases must be compatible, i.e., each parameter must have the same value or be filtered by "**any**" in each of the four "**exit**" statements (for instance, parameter `p0` is set to `true` or left unchanged in the first phase and filtered by "**any**" in the other phases). This is necessary in order to ensure the correct synchronisation of the four parallel phases upon their termination (*join*), as specified by the semantics of the "`|||`" operator.

Once the four parallel phases terminated, the main controller commands the rotation of the table, waits for the corresponding response from the local controller, then updates its parameters (with the values catched using the "`>>` **accept** ... **in**" operator) and restarts its cyclic execution.

3.5 Environment

The last element that we must specify in order to obtain a complete description of the system is the environment, which handles the input and output requests of the main controller by inserting and removing products into and from the slots at positions 0 and 3 of the turning table, respectively.

```
process Env [REQ, ADD, REM, ERR] : noexit :=
    REQ !Add;
       ADD;
           Env [REQ, ADD, REM, ERR]
    []
    REQ !Remove ?r:Bool;
    (  [r] -> REM;
          Env [REQ, ADD, REM, ERR]
       []
       [not (r)] -> ERR; REM;
          Env [REQ, ADD, REM, ERR]
    )
endproc
```

When a product is removed, the main controller also indicates to the environment whether the product has been correctly drilled or not, this latter case being signaled by the environment through the gate `ERR`.

4 Analysis of the functioning of the drilling unit

Once the drilling unit has been specified in Lotos, we can analyse its behaviour by using the verification tools of Cadp. We study first the coherence between the two versions of the system, equipped with the sequential and the parallel main controller, respectively. Then, we identify a set of correctness properties of the drilling unit, express them in temporal logic and verify them on the two versions of the system.

4.1 Equivalence checking

We begin by constructing, using the two Lotos compilers Cæsar and Cæsar.adt, the Lts model M_{seq} of the drilling unit equipped with the sequential main controller, in order to estimate its size and possibly to attempt its visual inspection. This Lts, minimized modulo branching bisimulation with the Bcg_Min tool and displayed graphically using the Bcg_Edit tool, is illustrated in Figure 4. It has 69 states and 72 transitions, and thus an average branching factor (number of transitions going out of a state) of $1,04$, which reflects the sequential nature of this version of the system.

Starting at the initial state (numbered 0) of the Lts, we observe a sequence of actions modelling the insertion of products in the slots of the turning table (which was initially empty) until the permanent functioning regime is reached (all the slots of the table are occupied). This sequence is followed by two different execution branches, corresponding to the fact that the product present at position 2 was correctly drilled (branch at the left) or not (branch at the right). These to branches denote a similar behaviour, excepting the presence of an `ERR` action on the branch at the right, indicating the output of an incorrectly drilled product to the environment.

We then build the Lts model M_{par} of the drilling unit equipped with the parallel main controller. This model is much larger than the model corresponding to the sequential controller: it has 24 346 states and 85 013 transitions[4], its average branching factor being $3,49$. This increase in size is caused by the interleaving of the four processing phases (corresponding to the positions of products on the turning table), which previously were chained sequentially. The size of the new Lts makes its visual inspection impractical; therefore, the use of verification tools becomes mandatory in order to assess the good functioning of the system.

A first verification consists to ensure that the behaviours of the two versions of the system modelled by M_{seq} and M_{par} are coherent. Intuitively, since the

[4] A reduction of M_{par} by τ-confluence [35] using the Reductor tool of Cadp would reduce its size to 5 373 states and 17 711 transitions.

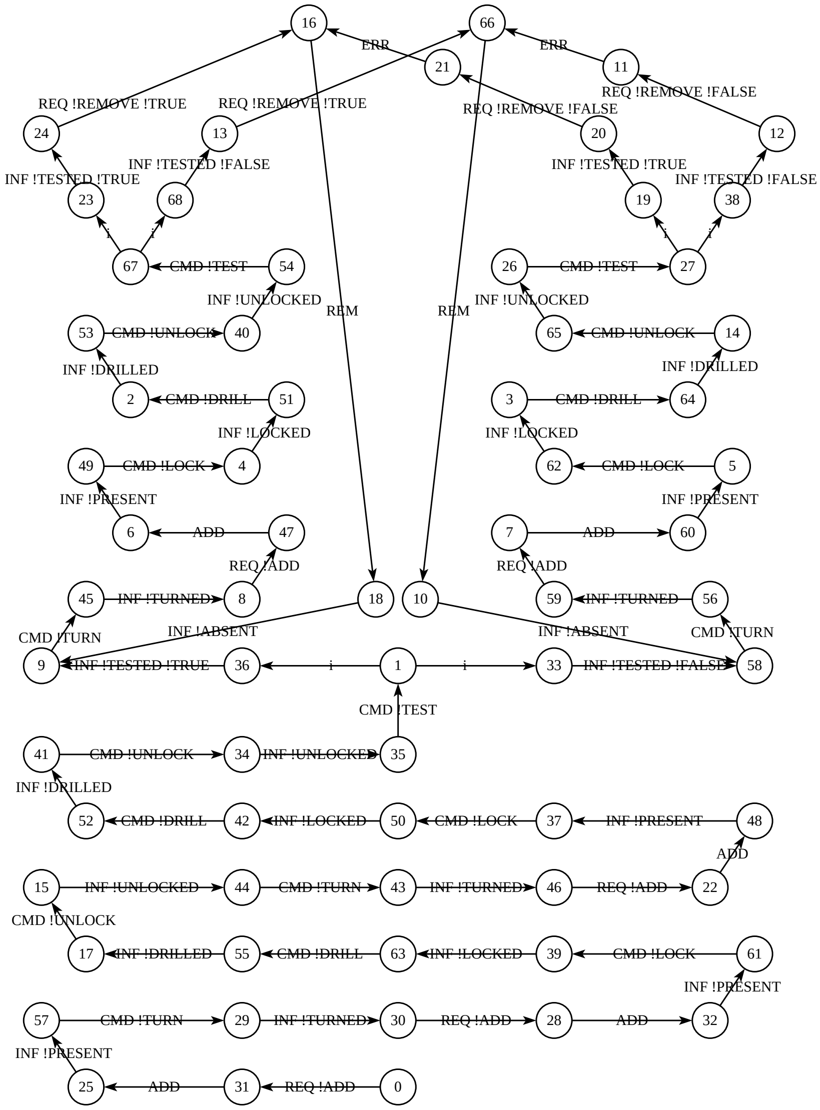

Fig. 4. Lts of the drilling unit equipped with the sequential controller

sequential chaining of the processing phases is a particular case of their parallel interleaving, all behaviours of the system equipped with the sequential controller should be "simulated" by the system equipped with the parallel controller. This is indeed the case: using Bisimulator, we can check that M_{seq} is included in M_{par} modulo the preorder of branching bisimulation. In other words, if we abstract away from the actions modelling the dialogue with the sensors and the actuators (see Section 3.1), the execution trees contained in M_{seq} are also contained in M_{par}. This verification can be done by traversing M_{par} on-the-fly; the underlying Bes explored by Bisimulator has 515 boolean variables and 934 operators.

On the other hand, the two Ltss M_{seq} and M_{par} are equivalent modulo none of the seven equivalence relations implemented by Bisimulator. Indeed, the following execution sequence (restricted to visible actions only) is present in M_{par} but not in M_{seq}:

$$s_0 \xrightarrow{\texttt{REQ !ADD}} s_1 \xrightarrow{\texttt{ADD}} s_2 \xrightarrow{\texttt{INF !PRESENT}} s_3 \xrightarrow{\texttt{CMD !TURN}} s_4 \xrightarrow{\texttt{INF !TURNED}} s_5 \xrightarrow{\texttt{CMD !LOCK}} s_6$$

This sequence, obtained automatically as a counterexample by trying to check the weak trace equivalence of M_{seq} and M_{par}, shows that after the input of the first product and the first rotation of the table, the sequential controller does not begin the drilling of the product (currently located at position 1) by commanding the block of the clamp, but restarts its cyclic functioning by handling the insertion of a new product in the slot at position 0 (currently empty). On the contrary, the parallel controller is able to command the drilling before the insertion, because it authorizes the concurrent execution of the four processing phases.

4.2 Model checking

The equivalence checking provided some indication about the coherence of the two versions of the system; however, it does not guarantee their correct functioning. In this section, we identify several temporal properties characterizing the correct ordering of actions during the execution of the system, and we express them in regular alternation-free μ-calculus [32], the temporal logic accepted as input by the Evaluator 3.5 model checker. We consider two kinds of classical properties (illustrated graphically in Figure 5):

Safety properties intuitively specify that "something bad never happens" during the execution of the system. They can be expressed in regular μ-calculus by the "$[R]$ false" formula, where R is a regular expression (defined over the alphabet of predicates on Lts actions) characterizing the undesirable action sequences that violate the safety properties. The necessity modality above states that all execution sequences going out of the current state and satisfying R must lead to states satisfying false; since there are no such states, the corresponding sequences do not exist either;

Liveness properties intuitively specify that "something good eventually happens" during the execution of the system. Most of the liveness properties that we will use here can be expressed in regular μ-calculus by (variants of) the "$[R]$ $\mathsf{inev}(A, B, P)$" formula, where R is a regular expressions over action sequences, A and B are action predicates, and P denotes a state formula. The formula above states that all execution sequences going out of the current state and satisfying R must lead to states from which all sequences are made of (zero or more) actions satisfying A, followed by an action satisfying B and leading to a state satisfying P. In other words, after every sequence satisfying R, it is inevitable to arrive (after some actions satisfying A, followed by an action satisfying B) at a state satisfying P.

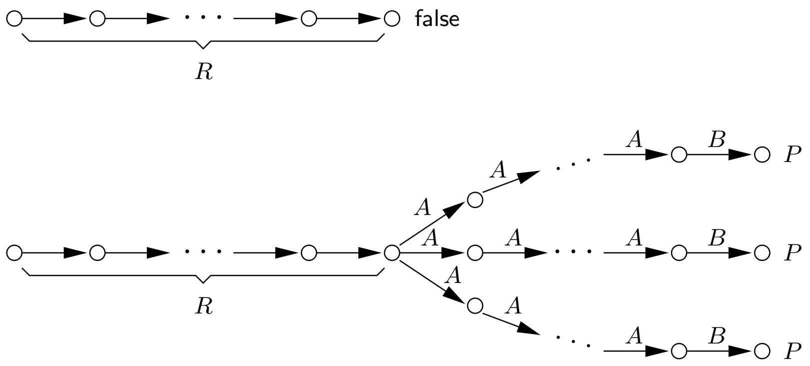

Fig. 5. Illustration of the "$[R]$ false" and "$[R]$ $\mathsf{inev}(A, B, P)$" operators

Table 3 shows seven safety properties of the drilling unit, together with their definitions in regular alternation-free μ-calculus. Evaluator 3.5 allows to express basic action predicates by using character strings surrounded by double quotes " (denoting a single action) or by using regular expressions over character strings (denoting a set of actions). Properties P_1–P_5 characterize the order of the processing phases performed on a product by the drilling unit, induced by the counterclockwise rotation (insertion, locking, drilling, unlocking, testing, and removal). Property P_6 expresses the safety of the drilling unit w.r.t. the testing of the products. Property P_7 expresses a constraint over the order of drilling and testing execution when the corresponding positions are both occupied.

Using Evaluator 3.5, we can check that all properties P_1–P_6 are satisfied by the Ltss M_{seq} and M_{par} corresponding to the two versions of the main controller. The size of the underlying Bess explored by Evaluator 3.5 varies from 321 variables and 336 operators (for property P_2 on M_{seq}) to 48 712 variables and 171 645 operators (for property P_4 on M_{par}). However, property P_7 is satisfied by M_{par} but not by M_{seq}: by running Evaluator 3.5 with the breadth-first

Table 3. Safety properties of the drilling unit

No.	Formula	Description
P_1	[true* . "INF !PRESENT" . (not "INF !TURNED")* . "INF !TURNED" . (not "INF !LOCKED")* . "CMD !DRILL"] false	After the input of a product and a rotation of the table, the main controller cannot command a drilling before the clamp has been blocked.
P_2	[true* . "INF !DRILLED" . (not 'INF !UNLOCKED.*')* . "CMD !TURN"] false	After the drilling of a product, the main controller cannot command a rotation before the clamp has been released.
P_3	[true* . "INF !UNLOCKED" . (not "INF !TURNED")* . "INF !TURNED" . (not 'INF !TESTED.*')* . "CMD !TURN"] false	After the release of the clamp on a product and a rotation of the table, the main controller cannot command another rotation before the product has been tested.
P_4	[true* . 'INF !TESTED.*' . (not "INF !TURNED")* . "INF !TURNED" . (not "INF !ABSENT")* . "CMD !TURN"] false	After the test of a product and a rotation of the table, the main controller cannot command another rotation before the product has been removed.
P_5	[true* . "INF !ABSENT" . (not "INF !TURNED")*. "INF !TURNED'' . (not "INF !PRESENT")*. "CMD !TURN"] false	After a product remove and a rotation of the table, the main controller cannot command another rotation before a new product has been supplied.
P_6	[true* . "INF !TESTED !TRUE" . (not "INF !TURNED")* . "INF !TURNED" . (not "INF !TURNED")* . "ERR"] false	Every time the tester detects a correctly drilled product, no error will be signaled during the next processing cycle.
P_7	[true* . "INF !PRESENT" . true* . "INF !PRESENT" . true* . "INF !PRESENT" . true* . "CMD !TEST" . (not "INF !TURNED")* . "CMD !DRILL"] false	After the testing and drilling positions of the table have been occupied, the main controller cannot command a test before commanding a drill.

strategy, we obtain a minimal counterexample sequence consisting of 23 visible actions. This sequence, absent from M_{seq} but present in M_{par}, models three insertions of products and two rotations of the table (which ensures that the slots at the drilling and testing positions become occupied), a clamp blocking command (which prepares the drilling), and a testing command followed by a drilling command. Indeed, the sequential controller handles the products present on the table in a precise order — the testing of the product at position 2 being carried out after the drilling of the product present at position 1 — whether the parallel controller is able to handle the various processing phases simultaneously.

Although the safety properties described above forbid the malfunctionings of the drilling unit, they do not guarantee the completion of the various processing phases on the products: thus, a drilling unit whose turning table does not move satisfies all these safety properties. In order to ensure the start and the progress of the processing phases, the system must also satisfy certain liveness properties. Table 4 shows seven liveness properties of the drilling unit, together with their corresponding temporal formulas. Property P_8 describes the starting sequence of the system, during which the products are inserted in all the slots of the turning table, which reaches the permanent functioning regime. Properties P_9–P_{12} are the counterparts of properties P_1–P_4: they characterize the progress of each processing phase and also the rotation of the table. Property P_{13} indicates the responses to the commands and requests of the main controller, which are sent by the system or by the environment during each processing cycle. Finally, property P_{14} specifies the correct reaction of the system when the drilling of some products failed.

Using Evaluator 3.5, we can verify that all properties P_8–P_{14} are satisfied by the Ltss M_{seq} and M_{par} corresponding to the two versions of the main controller. The size of the underlying Bess varies from 650 variables and 947 operators (for property P_8 on M_{seq}) to 275 277 variables and 606 747 operators (for property P_{13} on M_{par}).

5 Conclusion and future work

By means of the drilling unit example detailed in this paper, we tried to illustrate the specification and verification methods offered by the Cadp toolbox for analysing the functional properties of industrial critical systems that involve asynchronous parallelism. The Lotos language appears to be suitable for describing in a succinct and abstract manner the functioning of the controllers in charge of driving the physical devices. At the present time, Cadp has been used for analysing 104 industrial case-studies[5] and the various generic components of the Bcg and Open/Cæsar environments have enabled the development of 32 derived tools[6]. The feedback received from these applications led to the development of new tools as well as to the improvement of existing tools as regards performance and user-friendliness.

[5] See the online catalog `http://www.inrialpes.fr/vasy/cadp/case-studies`

[6] See the online catalog `http://www.inrialpes.fr/vasy/cadp/software`

Table 4. Liveness properties of the drilling unit

No.	Formula	Description
P_8	inev (not "`CMD !TURN`", "`REQ !ADD`", inev (not "`CMD !TURN`", "`CMD !TURN`", inev (not "`CMD !TURN`", "`REQ !ADD`", inev (not "`CMD !TURN`", "`CMD !TURN`", inev (not "`CMD !TURN`", "`REQ !ADD`", inev (not "`CMD !TURN`", "`CMD !TURN`", inev (not "`CMD !TURN`", "`REQ !ADD`", true)))))))	Initially, the main controller eventually commands the insertion of products in all the slots of the turning table.
P_9	[true* . "`INF !PRESENT`"] inev (not "`CMD !TURN`", "`CMD !TURN`", inev (not "`CMD !TURN`", "`CMD !LOCK`", inev (not "`CMD !TURN`", "`CMD !DRILL`", inev (not "`CMD !TURN`", "`CMD !UNLOCK`", true))))	Each product inserted will be drilled after the next rotation of the table.
P_{10}	[true* . "`INF !UNLOCKED`"] inev (not "`CMD !TURN`", "`CMD !TURN`", inev (not "`CMD !TURN`", "`CMD !TEST`", true))	Each product drilled will be tested after the next rotation of the table.
P_{11}	[true* . '`INF !TESTED.*`'] inev (not "`CMD !TURN`", "`CMD !TURN`", inev (not "`CMD !TURN`", '`REQ !REMOVE.*`', true))	Each product tested will be removed after the next rotation of the table.
P_{12}	[true* . "`INF !ABSENT`"] inev (not "`CMD !TURN`", "`CMD !TURN`", inev (not "`CMD !TURN`", "`REQ !ADD`", true))	Each product removal will be followed by the insertion of a new product after the next rotation of the table.
P_{13}	[true*] (["`REQ !ADD`"] inev (not "`INF !TURNED`", "`INF !PRESENT`", true) and ["`CMD !LOCK`"] inev (not "`INF !TURNED`", "`INF !LOCKED`", true) and ["`CMD !DRILL`"] inev (not "`INF !TURNED`", "`INF !DRILLED`", true) and ["`CMD !UNLOCK`"] inev (not "`INF !TURNED`", "`INF !UNLOCKED`", true) and ["`CMD !TEST`"] inev (not "`INF !TURNED`", '`INF !TESTED.*`', true) and ['`REQ !REMOVE.*`'] inev (not "`INF !TURNED`", "`INF !ABSENT`", true) and ["`CMD !TURN`"] inev (not "`INF !TURNED`", "`INF !TURNED`", true))	Each command (resp. request) sent by the main controller to the physical devices (resp. to the environment) will be eventually followed by its acknowledgement before the next rotation of the table.
P_{14}	[true* . "`INF !TESTED !FALSE`"] inev (not "`CMD !TURN`", "`CMD !TURN`", inev (not "`CMD !TURN`", "`ERR`", true))	Every time the tester detects an incorrectly drilled product, an error will be eventually signaled during the next processing cycle.

We presented here only a few basic tools of CADP, which implement classical verification methods (equivalence checking and model checking) on LTSs. CADP also offers sophisticated analysis functionalities in order to deal with large-scale systems: compositional verification using the EXP.OPEN 2.0 tool [27], distributed verification on clusters using the DISTRIBUTOR and BCG_MERGE tools [19, 18] and distributed BES resolution algorithms [24, 25], partial order reduction [35, 30]. These functionalities are orthogonal and operate on-the-fly, being based upon the implicit representation of LTSs defined by OPEN/CÆSAR: consequently, they can be combined in order to cumulate their benefits and scale up the analysis capabilities to large systems. Moreover, CADP offers the SVL language [16] and its associated compiler, which enable a succinct and elegant description of complex verification scenarios, involving hundreds of invocations of the verification tools.

The research and development activities around CADP are currently pursued along several directions. The rise of massively parallel computing architectures such as clusters and grids requires the design of specific distributed verification algorithms as well as the definition of LTS representations adequate for the distribution [19]. CADP can also play the role of analysis engine for other languages, namely those dedicated to the description of asynchronous hardware [37]. Finally, the application of the verification techniques promoted by CADP in other domains, such as bioinformatics, appears particularly promising [2].

References

1. Henrik R. Andersen. Model Checking and Boolean Graphs. *Theoretical Computer Science*, 126(1):3–30, April 1994.
2. Grégory Batt, Damien Bergamini, Hidde de Jong, Hubert Garavel, and Radu Mateescu. Model Checking Genetic Regulatory Networks using GNA and CADP. In Susanne Graf and Laurent Mounier, editors, *Proc. of the 11^{th} Int. SPIN Workshop on Model Checking of Software SPIN'2004 (Barcelona, Spain)*, volume 2989 of *Lecture Notes in Computer Science*, pages 156–161. Springer Verlag, April 2004.
3. Gerd Behrmann, Alexandre David, and Kim G. Larsen. A Tutorial on Uppaal. In Marco Bernardo and Flavio Corradini, editors, *Proc. of the 4^{th} Int. School on Formal Methods for the Design of Computer, Communication, and Software Systems SFM-RT'04 (Bertinoro, Italy)*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer Verlag, September 2004.
4. Damien Bergamini, Nicolas Descoubes, Christophe Joubert, and Radu Mateescu. BISIMULATOR: A Modular Tool for On-the-Fly Equivalence Checking. In Nicolas Halbwachs and Lenore Zuck, editors, *Proc. of the 11^{th} Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2005 (Edinburgh, Scotland, UK)*, volume 3440 of *Lecture Notes in Computer Science*, pages 581–585. Springer Verlag, April 2005.
5. Bernard Berthomieu, Florent Peres, and François Vernadat. *Time Petri Nets — Analysis Methods and Verification with TINA*. In Stephan Merz and Nicolas Navet, editors, *Modeling and Verification of Real-Time Systems — Formalisms and Software Tools*, chapter 1, pages 19–50. ISTE publishing / John Wiley, 2008.
6. Elena Bortnik, Nikola Trcka, Anton J. Wijs, S. P. Luttik, Joanna M. van de Mortel-Fronczak, Jos C. M. Baeten, Willem J. Fokkink, and J. E. Rooda. Analyzing a χ

Model of a Turntable System using Spin, CADP and UPPAAL. *Journal of Logic and Algebraic Programming*, 65(2):51–104, November 2005.

7. V. Bos and J.J.T. Kleijn. *Formal Specification and Analysis of Industrial Systems*. Phd thesis, Technical University of Eindhoven, March 2002.
8. Marius Bozga, Susanne Graf, Laurent Mounier, and Iulian Ober. *Modeling and Verification of Real-Time Systems using the IF Toolset*. In Stephan Merz and Nicolas Navet, editors, *Modeling and Verification of Real-Time Systems — Formalisms and Software Tools*, chapter 10, pages 319–352. ISTE publishing / John Wiley, 2008.
9. S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A Theory of Communicating Sequential Processes. *Journal of the ACM*, 31(3):560–599, July 1984.
10. Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in Property Specifications for Finite-State Verification. In Barry Boehm, David Garlan, and Jeff Kramer, editors, *Proc. of the 21st Int. Conference on Software Engineering ICSE'99 (Los Angeles, CA, USA)*, pages 411–420. ACM, May 1999.
11. H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1 — Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer Verlag, 1985.
12. M. J. Fischer and R. E. Ladner. Propositional Dynamic Logic of Regular Programs. *Journal of Computer and System Sciences*, 18(2):194–211, 1979.
13. Hubert Garavel. Compilation of LOTOS Abstract Data Types. In Son T. Vuong, editor, *Proc. of the 2nd Int. Conference on Formal Description Techniques FORTE'89 (Vancouver B.C., Canada)*, pages 147–162. North Holland, December 1989.
14. Hubert Garavel. OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation, and Testing. In Bernhard Steffen, editor, *Proc. of the First Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'98 (Lisbon, Portugal)*, volume 1384 of *Lecture Notes in Computer Science*, pages 68–84. Springer Verlag, March 1998.
15. Hubert Garavel and Holger Hermanns. On Combining Functional Verification and Performance Evaluation using CADP. In Lars-Henrik Eriksson and Peter A. Lindsay, editors, *Proc. of the 11th Int. Symposium of Formal Methods Europe FME'2002 (Copenhagen, Denmark)*, volume 2391 of *Lecture Notes in Computer Science*, pages 410–429. Springer Verlag, July 2002.
16. Hubert Garavel and Frédéric Lang. SVL: a Scripting Language for Compositional Verification. In Myungchul Kim, Byoungmoon Chin, Sungwon Kang, and Danhyung Lee, editors, *Proc. of the 21st IFIP WG 6.1 Int. Conference on Formal Techniques for Networked and Distributed Systems FORTE'2001 (Cheju Island, Korea)*, pages 377–392. IFIP, Kluwer Academic Publishers, August 2001.
17. Hubert Garavel, Frédéric Lang, and Radu Mateescu. An Overview of CADP 2001. *European Association for Software Science and Technology (EASST) Newsletter*, 4:13–24, August 2002. also available as INRIA Technical Report RT-0254.
18. Hubert Garavel, Radu Mateescu, Damien Bergamini, Adrian Curic, Nicolas Descoubes, Christophe Joubert, Irina Smarandache-Sturm, and Gilles Stragier. DISTRIBUTOR and BCG_MERGE: Tools for Distributed Explicit State Space Generation. In Holger Hermanns and Jens Palberg, editors, *Proc. of the 12th Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2006 (Vienna, Austria)*, Lecture Notes in Computer Science. Springer Verlag, March 2006.

19. Hubert Garavel, Radu Mateescu, and Irina Smarandache. Parallel State Space Construction for Model-Checking. In Matthew B. Dwyer, editor, *Proc. of the 8^{th} Int. SPIN Workshop on Model Checking of Software SPIN'2001 (Toronto, Canada)*, volume 2057 of *Lecture Notes in Computer Science*, pages 217–234. Springer Verlag, May 2001.
20. Hubert Garavel and Joseph Sifakis. Compilation and Verification of LOTOS Specifications. In L. Logrippo, R. L. Probert, and H. Ural, editors, *Proc. of the 10^{th} Int. Symposium on Protocol Specification, Testing and Verification (Ottawa, Canada)*, pages 379–394. IFIP, North Holland, June 1990.
21. Jan Friso Groote. The Syntax and Semantics of Timed muCRL. Technical Report SEN-R9709, CWI, Amsterdam, 1997.
22. Gerard Holzmann. *The SPIN Model Checker — Primer and Reference Manual*. Addison-Wesley Professional, 2003.
23. ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. Int. Standard 8807, Int. Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève, September 1989.
24. Christophe Joubert and Radu Mateescu. Distributed On-the-Fly Equivalence Checking. In Lubos Brim and Martin Leucker, editors, *Proc. of the 3^{rd} Int. Workshop on Parallel and Distributed Methods in Verification PDMC'2004 (London, UK)*, volume 128 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2004.
25. Christophe Joubert and Radu Mateescu. Distributed Local Resolution of Boolean Equation Systems. In Francisco Tirado and Manuel Prieto, editors, *Proc. of the 13^{th} Euromicro Conference on Parallel, Distributed and Network-Based Processing PDP'2005 (Lugano, Switzerland)*, pages 264–271. IEEE Computer Society, February 2005.
26. D. Kozen. Results on the Propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.
27. Frédéric Lang. EXP.OPEN 2.0: A Flexible Tool Integrating Partial Order, Compositional, and On-the-Fly Verification Methods. In Jaco van de Pol, Judi Romijn, and Graeme Smith, editors, *Proc. of the 5^{th} Int. Conference on Integrated Formal Methods IFM'2005 (Eindhoven, The Netherlands)*, volume 3771 of *Lecture Notes in Computer Science*, pages 70–88. Springer Verlag, November 2005.
28. Radu Mateescu. Efficient Diagnostic Generation for Boolean Equation Systems. In Susanne Graf and Michael Schwartzbach, editors, *Proc. of 6^{th} Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2000 (Berlin, Germany)*, volume 1785 of *Lecture Notes in Computer Science*, pages 251–265. Springer Verlag, March 2000.
29. Radu Mateescu. A Generic On-the-Fly Solver for Alternation-Free Boolean Equation Systems. In Hubert Garavel and John Hatcliff, editors, *Proc. of the 9^{th} Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2003 (Warsaw, Poland)*, volume 2619 of *Lecture Notes in Computer Science*, pages 81–96. Springer Verlag, April 2003.
30. Radu Mateescu. On-the-Fly State Space Reductions for Weak Equivalences. In Tiziana Margaria and Mieke Massink, editors, *Proc. of the 10^{th} Int. Workshop on Formal Methods for Industrial Critical Systems FMICS'05 (Lisbon, Portugal)*, pages 80–89. ERCIM, ACM Computer Society Press, September 2005.
31. Radu Mateescu. CAESAR_SOLVE: A Generic Library for On-the-Fly Resolution of Alternation-Free Boolean Equation Systems. *Springer Int. Journal on Software Tools for Technology Transfer (STTT)*, 8(1):37–56, February 2006.

32. Radu Mateescu and Mihaela Sighireanu. Efficient On-the-Fly Model-Checking for Regular Alternation-Free Mu-Calculus. *Science of Computer Programming*, 46(3):255–281, March 2003.
33. Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
34. Rocco De Nicola and Frits W. Vaandrager. Action versus State Based Logics for Transition Systems. In Irène Guessarian, editor, *Semantics of Systems of Concurrent Processes, LITP Spring School on Theoretical Computer Science (La Roche Posay, France)*, volume 469 of *Lecture Notes in Computer Science*, pages 407–419. Springer Verlag, April 1990.
35. Gordon Pace, Frédéric Lang, and Radu Mateescu. Calculating τ-Confluence Compositionally. In Jr Warren A. Hunt and Fabio Somenzi, editors, *Proc. of the 15th Int. Conference on Computer Aided Verification CAV'2003 (Boulder, Colorado, USA)*, volume 2725 of *Lecture Notes in Computer Science*, pages 446–459. Springer Verlag, July 2003.
36. Pascal Raymond. *Synchronous Program Verification with Lustre/Lesar*. In Stephan Merz and Nicolas Navet, editors, *Modeling and Verification of Real-Time Systems — Formalisms and Software Tools*, chapter 6, pages 171–206. ISTE publishing / John Wiley, 2008.
37. Gwen Salaün and Wendelin Serwe. Translating Hardware Process Algebras into Standard Process Algebras — Illustration with CHP and LOTOS. In Jaco van de Pol, Judi Romijn, and Graeme Smith, editors, *Proc. of the 5th Int. Conference on Integrated Formal Methods IFM'2005 (Eindhoven, The Netherlands)*, volume 3771 of *Lecture Notes in Computer Science*, pages 287–306. Springer Verlag, November 2005.
38. R.R.H. Schiffelers, D.A. van Beek, K.L. Man, M.A. Reniers, and J.E. Rooda. Formal Semantics of Hybrid Chi. In Kim G. Larsen and Peter Niebert, editors, *Proc. of the 1st Int. Workshop on Formal Modeling and Analysis of Timed Systems FORMATS'03 (Marseille, France)*, volume 2791 of *Lecture Notes in Computer Science*, pages 151–165. Springer Verlag, September 2003.