Automatic Verification of a Hydroelectric Power Plant[1]

Rosario Pugliese
Dip. di Scienze dell'Informazione
Università di Roma "La Sapienza"
via Salaria, 113, 00198 Roma, Italy
pugliese@dsi.uniroma1.it

Enrico Tronci
Dip. di Matematica Pura ed Appl.
Università di L'Aquila
Coppito, 67100 L'Aquila, Italy
tronci@univaq.it

Abstract. We analyze the specification of a hydroelectric power plant by ENEL (the Italian Electric Company). Our goal is to show that for the specification of the plant (its control system in particular) some given properties hold.
We were provided with an informal specification of the plant. From such informal specification we wrote a formal specification using the CCS/Meije process algebra formalism. We defined properties using μ-calculus. Automatic verification was carried out using model checking. This was done by translating our process algebra definitions (the model) and μ-calculus formulas into BDDs.
In this paper we present the informal specification of the plant, its formal specification, some of the properties we verified and experimental results.

1 Introduction

Computer controlled systems are more and more widespread. In safety critical applications this situation calls for formal verification of correctness with respect to the given specifications. Because of the cost of modifying the finished product it is essential that design errors are detected as early as possible in the design process. For this reason formal verification is also used to guarantee that for the specification of the plant some given properties hold.

Development of formal methods should go hand in hand with realistic case studies. Case studies are useful to assess the applicability of a verification technique and to guide research on new verification techniques.

In this paper we report on the analysis of a hydroelectric power plant by ENEL (the Italian Electric Company). Our goal is to guarantee that for the specification of the plant some given properties hold. We were provided with an informal specification of the plant [ENEL 92]. From this we derived a formal specification written using the CCS/Meije [AB 84] process algebra formalism. A previous version of such a formal specification was given in [LP 94]. To define properties we used μ-calculus (see, e.g., [BCMDH 92]) since it is a clean and expressive logic. Automatic verification was carried out using model checking.

[1] This work has been partially supported by the EUROFORM network and MURST funds.

This was done by translating process algebra definitions (the model) and μ-calculus formulas into Binary Decision Diagrams (BDDs, see [Bry 86]).

In this paper we present the informal specification of the plant, part of its formal specification, some of the properties we verified and experimental results. A full version of the formal specification is in [PT 95].

The main difficulties we had to face were: ambiguities in the informal specification and state explosion (our system has about 10^{52} states).

We succeeded in automatically verifying relevant properties of the formal specification. Our experiments were carried out on a SUN Sparc LX with 72MB RAM. Verification required building BDDs with up to $2 \cdot 10^6$ vertices. We needed 174 boolean variables to code the state of the system and 15 boolean variables to code the actions of the system. Thus the transition relation *(present state, action, next state)* of the overall system had 363 boolean variables. We did not have problems in building such transition relation, but its size forced us to be very careful during verification. In particular to avoid running out of memory during automatic verification a careful choice of the logic formula representing the property we wanted to verify was necessary.

The rest of the paper is organized as follows. In section 2 we give an informal description of the plant. In section 3, using CCS/Meije, we present a formal description of part of the plant. In section 4 we give μ-formulas for some of the formal properties we verified. In section 5 we give experimental results. In appendix A we define the syntax and sketch the operational semantics of CCS/Meije.

2 The case study: an informal description

Our reactive system is composed of a hydroelectric power plant and its control system. The control system drives the engines producing electrical power, handles the basin sluices, checks for safe working of the plant and monitors the water level in the basin. Moreover the control system interacts with an operator which solves by hand critical situations arising in the plant administration. The informal specification of the overall system was provided by ENEL (the Italian Electric Company) in [ENEL 92]. Such informal specification was quite ambiguous and incomplete. Since we wanted to study the power of CCS/Meije as a specification methodology we simplified the original specification by using small abstract domains as data values. Ambiguities were also removed. Nevertheless the resulting case study is meaningful and nontrivial. In the rest of this section we present such simplified version of the original specification provided by ENEL.

The environment of the *control system* includes a *hydroelectric power plant*, an *operator* and the aspects of the *surrounding environment* that have a prominent influence on the plant. The plant is composed of a *catch basin* and a *power plant*. The power plant is composed of *energy production engines* and a *penstock* to ensure a pressurized water flow from the basin to the power plant. Each *energy production engine* has a *generator* and an embedded *controller*.

The power plant can be directed by the operator (*manual administration*) or by the control system (*automatic administration*). The main goals of the control system are:

- Managing hydraulic resources so that the plant produces as much power as possible;
- Obtaining the best efficiency from production engines during daily production periods (which, in turn, are transmitted by the operator);
- Saving up hydraulic power outside of daily production periods without violate plant constraints (e.g. do not exceed minimum and maximum levels in the basin).

The control system achieves these goals by interacting with the power production engines and the basin sluices. In the following, we will ignore the interaction with the sluices and we will only concentrate on interaction with the production engines.

2.1 The catch basin

The shape and dimensions of the basin are known and, for the sake of simplicity, we will suppose that the water volume fluctuation can be computed by means of the water level fluctuation, that, in turn, is evaluated by means of a *transducer*. Hence, in the following, we will make no distinction between water volume or level in the basin. Water level increases due to rain and inflow of water through affluents. It decreases due to overflow from barrages and outflow through the penstock connecting the basin to the engines. Some water levels have a particular importance for the working of the plant. They are:

VMS (*) overflowing level;
VML maximum working level;
INT_1 (*) maximum working level when the transducer is broken;
INT_2 (*) minimum working level when the transducer is broken;
Vml minimum working level;
VMIN (*) minimum capacity level.

Some *sensors* send signals to the control system when one of the levels labelled with * is reached by the surface of water. When the transducer is broken it is possible to control the plant, even if in a less accurate way, using such sensors.

2.2 The hydroelectric power plant

The hydroelectric power plant is formed by some energy production engines, each of which contains a generator and an embedded controller.

The generator A generator is characterized by a finite state automaton whose states can be partioned into two classes: stable states and unstable states. Here are the generator states accordingly to such partition.

- *stable states*:
  S (stopped): the generator is stopped;
  G (generating): the generator is working;
  Bt (temporary block): the generator is blocked because of a fault that can be automatically removed (i.e. without the intervention of the operator);
  Bp (persistent block): the generator is blocked because of a failure that only the operator can fix. After removing such failure the operator has to restart the generator and signal to the control system that recovery has taken place.
- *unstable states*:
  sTg: transition state from **S** to **G** (representing the situation in which the generator is going from state **S** (stopped) to state **G** (working));
  gTs: transition state from **G** to **S** (representing the situation in which the generator is going from state **G** (working) to state **S** (stopped));
  gTbt: transition state from **G** to **Bt** (representing the situation in which the generator is going from state **G** (working) to state **Bt** (temporary block));
  gTbp: transition state from **G** to **Bp** (representing the situation in which the generator is going from state **G** (working) to state **Bp** (persistent block)).

To each generator the control system can send two kind of commands:

- *state commands*: **start/halt** to change the state of the generator;
- *position commands*: **inc/dec** to increase/decrease electric power production.

During the generation phase (state **G**), a generator may produce different amounts of power at different times, accordingly to the energy production plan. The working point of a generator can be adjusted on different positions. Each position selects a different amount of water intake and thus a different amount of power that can be produced in a period of time.
A generator can make the following transitions:

- from **S** to **sTg** when it receives **start** and from **sTg** to **G** when the command is executed;
- from **G** to **gTs** when it receives **halt** and from **gTs** to **S** when the command is executed;
- from **G** to **gTbt** when a temporary fault occurs and from **gTbt** to **Bt** when the generator stops;
- from **G** to **gTbp** when a failure whose repairing needs the operator intervention occurs and from **gTbp** to **Bp** when the generator gets stuck because of such failure;
- from **Bt** to **S** when the trouble causing the block disappears;
- from **Bp** to **S** when the operator has repaired the blocked generator.

The controller The controller behaves as a transparent interface between the control system and the generator. Indeed, the control system sends a command to the controller and this one transmits the command to the related generator and sends to the control system one of the following signals:

- The generator state is changing (until the expected state is reached). This happens when the generator has received a command asking for a change of state.
- The generator has not correctly changed its generation position. This happens when the generator has received a command asking for a change in the generation position and the generator has not successfully executed it.

The controller accepts and sends one command at a time. Moreover, for hydraulic reasons, the control system can send only one kind of command (change state or change position) at a time. This means that at most one generator at a time may execute a command to change state and at most one generator at a time may execute a command to change position.

With respect to the control system, the controller appears in one of two possible states:

- *available*, when it is prepared to automatic administration: it accepts commands from the control system and transmits them to the generator;
- *nonavailable*, when the generator is directed by the operator or a (temporary or persistent) block happened: in such a case the controller will ignore the commands from the control system.

Interactions between power production engines, control system and operator The control system handles the generator and its controller as one single entity. That is the control system interacts only with the controller which then transmits commands to the generator.

A production engine goes from the state nonavailable to the state available when the controller receives an automatic administration signal and no block (temporary or persistent) occurred. A production engine goes from available to nonavailable when a manual administration signal arrives or a block occurs.

After a temporary block, a generator is available again as soon as a variation in the storage curve (see section 2.3: Governing task) is expected. After a persistent block, a generator is available again only if explicitly required by the operator (following recovery of the generator).

Handling of state commands The signal **busy** is the answer of the controller to a state command. If the control system does not receive such answer it tries again sending the command at most twice. If both trials fail, it sends **halt** without verifying its outcome and declares that generator is out of order by sending a signal **alarm_gr_bad** to the operator. After having received a signal **busy** (following a state command), each minute for at most five minutes, the control system checks if the generator has reached the expected state. If this is

the case then the control system sends a position command accordingly to the storage curve (see section 2.3: Governing task), otherwise, at the end of the fifth minute, it sends **halt** without verifying the outcome and declares that generator is out of order by sending a signal **alarm_gr_bad** to the operator.

Handling of position commands Position commands are sent each sampling time (see section 2.3: Activities of the control system) or, if necessary, after a successful state command. A minute after a position command has been sent, the control system checks the actual position of the generator. If such position is not the one expected, the control system tries again sending the command for at most two times and waiting for a successful outcome each time. If the second trial also fails, the system sends **halt** without verifying the outcome and declares that generator is out of order by sending a signal **alarm_gr_bad** to the operator.

2.3 The control system

Activities of the control system All control system activities are timed: the operations are executed *each minute* or *each five minutes* or *each sampling time.* For example, each minute or each five minutes, the control system records the evaluations of some quantities, and, each sampling time, computes their expected values and takes the appropiate actions.

Parameters When the control system is activated, it receives the initialization values for parameters from the operator. These parameters are: *water level* in the basin, *sampling time, time, daily production periods* and *connection priorities* of generators.

Working The control system usually monitors the basin and the power plant; but, if it receives a *managing consent* from the operator, then it has also to actively manages them.

At any moment the control system can receive data from the operator (e.g. time and water level), a managing agreement or its annulment, and a signal saying that a bad generator has been repaired.

Monitoring task The monitoring task is reading and updating periodically data about water level, state and position of generators.

Governing task The governing task consists of the following operations:

- check if there exist available generators;
- check the state and position of each available generator;
- if there is at least a generator available then
  - compute the basin fluent discharge;

- recompute the production program by using the data previously acquired.

We now examine in more detail the above mentioned activities. When the control system is enabled to direct the plant, it computes a *production program* and then, conforming to such program, establishes the amount of power the plant must produce during the day and sends the appropriate commands to generators. Production program inputs are the initialization values for parameters and *fluent discharge*. Fluent discharge is the average value of the variation of water volume in the basin (owing to rain and affluents) as recorded in the previous 24 hours. Production program output is the *storage curve*, that is a set of pairs (*volume, time*) specifying the average value of water volume each sampling time. To each pair (*volume, time*) is associated information about the expected state and position of generators.

The production activity of the plant should maintain water level in the interval between **Vml** and **VML**. This is the normal working range. If fluent discharge deviates from the expected values and the water level is not in the normal range then the control system has to take measures in order to drive the level back into the normal range. In such a case, it sends a signal **alarm_level_bad** to the operator, leaves the daily program for power production and follows a program trying to comply with the expected variations of water volume. In particular the control system estimates volume fluctuation in the next time interval on the basis of fluent discharge in the previous sampling time. It does not use the average fluent discharge from the previous 24 hours since in this situation it is not a reliable forecast. Finally the control system establishes a power (not greater than the maximum one the plant admits) to drive the volume back into the normal working range and sends appropriate commands to the generators.

INT_2 and **INT_1** are used in place of, respectively, **Vml** and **VML** when the transducer is broken.

Timed activities There is a clock that sends periodic timeout signals to some of the processes forming the system. Such timeouts are sent: each minute, each five minutes and each sampling time.

Activities accomplished every minute

- Update the output buffer towards the operator. If such updating is not successful then send a signal **alarm_bo_broken** to the operator and go on with the administration.
- Read and update availability of generators.
- Check the result of possible state or position commands previously sent.

Activities accomplished every five minutes

- Read transducer and sensors.

- If water level is out of the normal working range, send **alarm_out_work_int** and recompute the storage curve (using the fluent discharge evaluated in the previous sampling time instead of the daily average value (see section 2.3: Governing task)).
- If the level fluctuation estimated by the transducer disagrees with the signals coming from sensors, send a signal **alarm_transducer_broken** and compute again the storage curve using **INT_2** and **INT_1** as minimum and maximum levels.
- If the level is **VMS** (overflow), order to all available generators to produce the maximum power.
- If the level is **VMIN** (minimum level) halt all generators.
- Obtain new parameter values from the operator.

Activities accomplished every sampling period

- Acquire the value of the level in the basin and behave accordingly to the storage curve (see section 2.3: Governing task).
- Send state and position commands to the available generators to obtain a situation in agreement with the one prescribed by the storage curve.
- Compute fluent discharge in the sampling time and update the mean flow in the last 24 hours.

3 Formal specification

In this section we present the CCS/Meije formal specification for the informal specification in section 2. A description of CCS/Meije is in appendix A. With respect to the informal description in sec. 2 we make the following assumptions.

- We assume that the number of production engines is 3 (no value is specified in the informal description in section 2).
- To simplify clocking we assume that the sampling time is *about* five minutes. Note that sampling time is not specified in the informal description in sec. 2.
- We assume that system components synchronize without exchanging values. This means that we are not considering value dependent computations. Note that no numerical value is defined in the informal description in section 2.

In the rest of this section we only present the CCS/Meije term we have used to formally define the specification for the generator in section 2.2. For the interested reader the complete formal specification is in [PT 95].

Process *Generator* in figure 1 models the generator described in section 2.2. Process *Generator* defines a finite state automaton (see figure 2) with initial state S. Transitions are defined with equations defining present-state, action, next-state. Actions have form a? (input action) or a! (output action). Processes communicate using synchronization. Thus a process can execute action a? (a!)

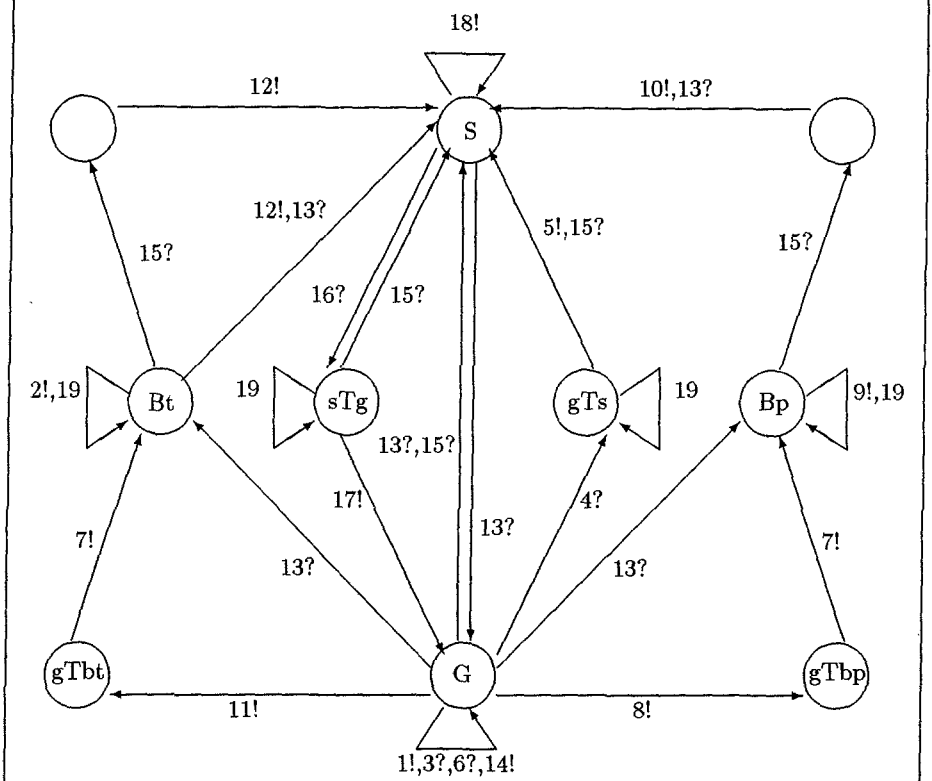
```
Generator = let rec {
        S = startGr?: sTg + stopped!: S + operator?: G
and     G = noise!: gTbt + lock!: gTbp + haltGr?: gTs + incGr?: G + decGr?: G
              + producing!: G + act_gr_pos!: G + repaired_generator?: S
              + operator?: S + operator?: Bt + operator?: Bp
and     Bt = noisegone!: S + broken!: Bt + τ: Bt
              + repaired_generator?: noisegone!: S + operator?: S
and     Bp = repaired_generator?: (lockgone!: S + operator?: S)
              + locked!: Bp + τ: Bp
and     sTg = startOk!: G + τ: sTg + repaired_generator?: S
and     gTbt = ko!: Bt
and     gTbp = ko!: Bp
and     gTs = haltOk!: S + τ: gTs + repaired_generator?: S
              } in S.
```

Fig. 1. Process *Generator*

only if some other process can execute action a! (a?). E.g. equation "gTbt = ko!: Bt", says that if we are in state gTbt then performing output action ko we reach state Bt. Nondeterminism is also possible. E.g. equation "S = startGr?: sTg + stopped!: S + operator?: G" says that from state S we can: perform input action startGr and reach state sTg or perform output action stopped and reach state S or perform input action operator and reach state G. Action τ models an action that is internal to the process performing it. Thus it is *invisible* to the other processes. We usually use τ's to model stuttering.

In the following we show how the definition of process *Generator* in this section links to the informal specification in section 2.2 (The generator). The informal meaning of all states has been already given in section 2.2 (The generator). The intended meaning of the signals (actions) is defined in the following.

- **startGr** requires the generator to start. This happens via a synchronization between process *Generator* and the process modelling the control system.
- **haltGr** requires the generator to halt. This happens via a synchronization between process *Generator* and the process modelling the control system.
- **incGr** requires the generator to increase the amount of produced power. This happens via a synchronization between process *Generator* and the process modelling the control system.
- **decGr** requires the generator to decrease the amount of produced power. This happens via a synchronization between process *Generator* and the process modelling the control system.
- **startOk** signals to the process modelling the control system that the generator has begun producing power.
- **haltOk** signals to the process modelling the control system that the generator has stopped producing power.
- **noise** signals to the process modelling the control system that a temporary failure has occurred and that the generator is going to stop.



Legend: 1 ≡ act_gr_pos, 2 ≡ broken, 3 ≡ decGr, 4 ≡ haltGr, 5 ≡ haltOk, 6 ≡ incGr, 7 ≡ ko, 8 ≡ lock, 9 ≡ locked, 10 ≡ lockgone, 11 ≡ noise, 12 ≡ noisegone, 13 ≡ operator, 14 ≡ producing, 15 ≡ repaired_generator, 16 ≡ startGr, 17 ≡ startOk, 18 ≡ stopped, 19 ≡ τ.

An edge labelled $l_1, \ldots, l_k$ represents k edges labelled, respectively, $l_1, \ldots, l_k$.

Fig. 2. Process *Generator* Automaton

- **lock** signals to the process modelling the control system that a permanent (i.e. requiring operator's intervention) failure has occurred and that the generator is going to stop.
- **noisegone** signals to the process modelling the control system that temporary failure has disappeared and that the generator is ready to start producing power again.
- **lockgone** signals to the process modelling the control system that permanent failure has been repaired and the generator is ready to start producing power again.
- **broken** signals to the process modelling the control system that the generator is not producing any power at all due to a temporary failure.

- **locked** signals to the process modelling the control system that the generator is not producing any power at all due to a permanent failure.
- **stopped** signals to the process which has the job of sending commands to generators and to the process modelling the control system that the generator is not producing any power at all.
- **producing** signals to the process which has the job of sending commands to generators and to the process modelling the control system that the generator is producing some amount of power.
- **act_gr_pos** signals to the process modelling the activity of management of the produced power and to the process modelling the control system the amount of power the generator is producing. We do not use a value–passing calculus. Thus action act_gr_pos is an abstraction for the real actions in which values are involved.
- **ko** is a visible (output) signal. That is process *Generator* does not have to synchronize with another component of the system in order to execute it. Action ko is used to signal (e.g. to a human operator) that something wrong happened and the generator is going to stop (temporarily or permanentely).
- **operator** is a visible (input) signal. That is process *Generator* does not have to synchronize with another component of the system in order to execute it. Signal operator is used to represent the possibility that the generator state changes because of a request coming from outside of the system (i.e. the operator).
- **repaired_generator** is a visible (input) signal. It is used to represent the fact that the generator has been repaired by the operator.

In the following we comment each of the equations defining process *Generator* in figure 1.

From state S (stopped) the generator can perform the following actions.

- **startGr?** When this signal is received the generator goes to unstable state sTg.
- **stopped!** The generator can output signal stopped and go (i.e. stay) to state S.
- **operator?** When this signal is received (from the operator) the generator goes in state G.

From state G (generating) the generator can perform the following actions.

- **noise!** Due to a temporary failure the generator goes to unstable state gTbt.
- **lock!** Due to a persistent failure the generator goes to unstable state gTbp.
- **haltGr?** The generator is required to stop. Thus it goes to unstable state gTs.
- **incGr?** When signal incGr is received the generator has to increase the power which it is producing going to another generation state. Remember that we abstract away from any kind of value thus we represent this situation by using state G again.

- **decGr?** When signal incGr is received the generator has to increase the power which it is producing going to another generation state. Remember that we abstract away from any kind of value thus we represent this situation by using state G again.
- **producing!** The generator signals it is producing and stays in the same state.
- **act_gr_pos!** The generator signals the amount of power which it is producing and stays in the same state. Since we abstract away from any kind of value we represent this situation by using only a signal act_gr_pos.
- **repaired_generator?** The generator receives (from the operator) signal repaired_generator and goes to the state S.
- **operator?** This (input) signal represents the possibility that the generator is managed by the operator. After receiving this signal the generator can go in one of states G, Bt or Bp.

From state Bt (temporary block) the generator can perform the following actions.

- **noisegone!** The generator has recovered from a temporary failure and goes to state S.
- **broken!** The generator signals it is not producing due to a temporary failure and stays in the same state.
- τ The generator performs invisible action τ and stays in the same state.
- **repaired_generator?** The generator signals that it has been repaired (by the operator) and goes to an anonymous state from which it can perform only action noisegone! going to state S.
- **operator?** The generator can change state going to state S by performing the action operator? which represents the possibility the generator is managed by the operator.

From state Bp (permanent block) the generator can perform the following actions.

- **repaired_generator?** The generator has been repaired (by the operator) and goes to an anonymous state from which it can perform either action lockgone! going to state S or action operator? going to state S.
- **locked!** The generator signals it is not producing due to a persistent failure and stays in the same state.
- τ The generator performs invisible action τ and stays in the same state.

From unstable state sTg the generator can perform the following actions.

- **startOk!** The generator outputs signal startOk and goes to state G (generation).
- τ The generators performs invisible action τ and stays in the same state.
- **repaired_generator?** The generator receives (from the operator) signal repaired_generator and goes to the state S.

From unstable state gTBt the generator can perform only the output action ko! and go to state Bt.
From unstable state gTBp the generator can perform only the output action ko! and go to state Bp.
From unstable state gTs the generator can perform the following actions.

- **haltOk!** The generator signals it is going to halt and go to state S.
- τ The generator performs invisible action τ and stays in the same state.
- **repaired_generator?** The generator receives (from the operator) signal repaired_generator and goes to the state S.

4 Properties

4.1 Basic definitions

We carry out automatic verification using (BDD based) model checking on the Boolean Domain $Boole = \{0, 1\}$. Thus, as far as automatic verification is concerned, each property is represented with a computation on boolean functions (namely those defining the transition functions of the processes). One way of defining such computations is by using μ-calculus. Many temporal logics can be uniformly translated into μ-calculus (see [BCMDH 92]). E.g. temporal operators are just μ-terms. Thus temporal logics can also be used to define properties in our setting. Note, however, that in general there are many μ-terms representing the same temporal operator. Thus using directly μ-calculus rather then a temporal logic allows a finer control on the verification process. In our case this was essential to succesfully complete our verification task. To define properties we use μ-calculus as defined in [BCMDH 92]. Roughly speaking we can say that μ-calculus on a Boolean Domain can be seen as First Order Logic on a Boolean Domain augmented with the least fixpoint operator μ. In the following symbol $\equiv$ denotes syntactic equality.

We use vectors of boolean variables to represent actions and states. We usually denote boolean vectors with capital letters (e.g. X, Y). Let $X \equiv x_1, \ldots x_k$, and Q be a binding operator (e.g. $\exists$, $\forall$, λ). We write QX for $Qx_1, \ldots x_k$. Let $Y \equiv y_1, \ldots y_k$ and op be a binary boolean operator (e.g. $\vee$, $\wedge$, $=$). We write $(X\ op\ Y)$ for $((x_1\ op\ y_1) \wedge \ldots (x_k\ op\ y_k))$. We write $F(x_1, \ldots x_m)$ to denote a formula which free variables are among $x_1, \ldots x_m$. Moreover we denote with $F(t_1, \ldots t_m)$ the formula obtained from $F(x_1, \ldots x_m)$ by simultaneosly substituting variables $x_1, \ldots x_m$ with terms $t_1, \ldots t_m$.

To use model checking we need to represent a process algebra as a μ-calculus model. We do this as in [EFT 91]. We use boolean vectors of size r $(= 15)$ to represent actions and boolean vectors of size n $(= 174)$ to represent states. We represent the initial state with a boolean vector having all components equal to 0. Thus the set of initial states (a singleton) is represented by the formula $S0(x_1, \ldots x_n) \equiv (x_1 = 0) \wedge \ldots (x_n = 0)$.

The transition relation of the overall system is represented with a predicate symbol S with arity $r + 2 * n$ $(= 363)$. Thus $S(X, A, X')$ holds iff from state X performing action A it is possible to reach state X'.

Because of the size of the BDD representing S time/space performance of automatic verification of a property strongly depends on the logic formula we choose to represent such property. To speed up reachability analysis it was our intention to use the iterative squaring technique in [BCMDH 92]. However we could not follow such approach because of memory overflow. We saved memory space (at the expense of computation time) by representing our properties with carefully chosen logic formulas. In particular we avoided building fixpoints of predicate symbols with arity greater than, say, $r + n (= 15 + 174 = 189)$. Lack of space prevents us from illustrating all of the properties we verified. In the following we give the logic formulas we used to define and automatically verify some of the properties we studied. This should be sufficient to illustrate our approach.

It will be useful to consider the set of states reachable in one step from a given state. Such set can be represented with the formula $S1(X, X') \equiv \exists A\ S(X, A, X')$.

We will be interested in the set of states reachable from a given state without performing a given action, say B. Such set can be represented with the formula $S2(B, X, X') \equiv \exists A\ (\neg(A = B) \wedge S(X, A, X'))$.

Let $V(X)$ be a formula representing a set of states. The set of states reachable from a state satisfying $V(X)$ is the least solution to the fixpoint equation (unknown: G) $G(X) = (V(X) \vee \exists Z[G(Z) \wedge S1(Z, X)])$. Such solution can be denoted with the μ-calculus formula $G(X)$ defined as follows: $G(X) \equiv \mu g[\lambda X[V(X) \vee \exists Z[g(Z) \wedge S1(Z, X)]]](X)$. E.g. the set $V0$ of states reachable from the initial states can be represented with the formula $G0(X)$ defined as follows: $G0(X) \equiv \mu g[\lambda X[S0(X) \vee \exists Z[g(Z) \wedge S1(Z, X)]]](X)$.

Let A be an action. The set $H(A)$ of states from which action A will be performed on at least one computation can be described with the formula $S3(A, X) \equiv \mu g[\lambda X[\exists Z[S(X, A, Z) \vee (S1(X, Z) \wedge g(A, Z))]]](X)$.

We will need to observe synchronizations between processes. To this end we slightly extend CCS/Meije by considering actions of the following forms: $a!$, $a?$ and a_τ. Thus when two processes synchronize on action a we will be able to observe a_τ instead of just τ (as in CCS/Meije). In the following we represent actions $a!$, $a?$ and a_τ with, respectively, boolean vectors (of size r) **a_out**, **a_in** and **a_tau**.

4.2 A safety property

We are now ready to define the first property we will study here. It is a safety property. Its informal statement is:

> *If a generator is out of order because of a persistent block then that generator will not be used until the operator has repaired it.*

Using actions the above property (for generator 1) can be expressed as follows. If the system performs the following actions: **ko_1_in** (generator 1 is out of order), **locked_1_tau** (the system has detected that generator 1 is out of order) **noavilable_1_tau** (generator 1 is declared unusable for automatic administration) then action **available_1_tau** (generator 1 is available again for automatic

administration) cannot be performed until action **repaired_generator_1_in** (the operator repaired generator 1) is performed.

In the following we build a formula describing such property.

The set $V0$ of states reachable from the initial state can be represented with the formula $G0(X)$ defined in section 4.1. The set $V1$ of states reachable from $V0$ by performing action **ko_1_in** is represented with the formula $G1(X)$ defined as follows: $G1(X) \equiv \exists Z[G0(Z) \wedge S(Z, \mathbf{ko\_1\_in}, X)]$.

The set $V2$ of states reachable from $V1$ is represented with the formula $G2(X)$ defined as follows: $G2(X) \equiv \mu g[\lambda X[G1(X) \vee \exists Z[g(Z) \wedge S1(Z, X)]]](X)$.

The set $V3$ of states reachable from $V2$ by performing action **locked_1_tau** is represented with the formula $G3(X)$ defined as follows: $G3(X) \equiv \exists Z[G2(Z) \wedge S(Z, \mathbf{locked\_1\_tau}, X)]$.

The set $V4$ of states reachable from $V3$ is represented with the formula $G4(X)$ defined as follows: $G4(X) \equiv \mu g[\lambda X[G3(X) \vee \exists Z[g(Z) \wedge S1(Z, X)]]](X)$.

The set $V5$ of states reachable from $V4$ by performing action **nonavailable_1_tau** is represented with the formula $G5(X)$ defined as follows: $G5(X) \equiv \exists Z[G4(Z) \wedge S(Z, \mathbf{nonavailable\_1\_tau}, X)]$.

The set $V6$ of states reachable from $V5$ without performing action **repaired_generator_1_in** is represented with the formula $G6(X)$ defined as follows:

$G6(X) \equiv \mu g[\lambda X[G5(X) \vee \exists Z[g(Z) \wedge S2(\mathbf{repaired\_generator\_1\_in}, Z, X))]](X)$.

The set $V7$ of states reachable from $V6$ by performing action **available_1_tau** is represented with the formula $G7(X)$ defined as follows: $G7(X) \equiv \exists Z[G6(Z) \wedge S(Z, \mathbf{available\_1\_tau}, X)]$.

Our safety property requires that $V7$ be empty. This is expressed by the formula $G8$ defined as follows: $G8 \equiv \neg \exists X G7(X)$.

To verify our property we have to check that G8 holds. We do this by computing a BDD representation for G8 and testing that the result is the (unique) BDD representing the boolean function identically equal to 1.

4.3 A liveness property

The second property we study is a liveness property. Its informal statement is:

> *If the plant is managed by the control system and a persistent block occurs on generator 1 and the operator repairs it then generator 1 will become usable again.*

Using actions the above property can be expressed as follows. If the system performs the following actions: **consent_in** (the plant is managed by the control system), **locked_1_tau** (generator 1 is unusable because of a persistent block), **repaired_generator_1_in** (generator 1 has been repaired by the operator) then action **available_1_tau** (generator 1 is usable) will be performed on at least one computation.

In the following we build a formula describing such property.

The set $V0$ of states reachable from the initial state can be represented with the formula $G0(X)$ defined in section 4.1. The set $V1$ of states reachable from

$V0$ by performing action **consent_in** is represented with the formula $G1(X)$ defined as follows: $G1(X) \equiv \exists Z[G0(Z) \wedge S(Z, \textbf{consent\_in}, X)]$.

The set $V2$ of states reachable from $V1$ is represented with the formula $G2(X)$ defined as follows: $G2(X) \equiv \mu g[\lambda X[G1(X) \vee \exists Z[g(Z) \wedge S1(Z, X)]]](X)$.

The set $V3$ of states reachable from $V2$ by performing action **locked_1_tau** is represented with the formula $G3(X)$ defined as follows: $G3(X) \equiv \exists Z[G2(Z) \wedge S(Z, \textbf{locked\_1\_tau}, X)]$.

The set $V4$ of states reachable from $V3$ is represented with the formula $G4(X)$ defined as follows: $G4(X) \equiv \mu g[\lambda X[G3(X) \vee \exists Z[g(Z) \wedge S1(Z, X)]]](X)$.

The set $V5$ of states reachable from $V4$ by performing action **repaired_generator_1_in** is represented with the formula $G5(X)$ defined as follows: $G5(X) \equiv \exists Z[G4(Z) \wedge S(Z, \textbf{repaired\_generator\_1\_in}, X)]$.

The set $V6$ of states reachable from $V5$ is represented with the formula $G6(X)$ defined as follows: $G6(X) \equiv \mu g[\lambda X[G5(X) \vee \exists Z[g(Z) \wedge S1(Z, X)]]](X)$.

The set $V7$ of states from which action **available_1_tau** will be performed on at least one computation can be represented with the formula $S3(\textbf{available\_1\_tau}, X)$ (see section 4.1).

Our liveness property requires that if a state is in set $V6$ then it is also in set $V7$. This can be represented with the formula $G8 \equiv \forall X[G6(X) \rightarrow S3(\textbf{available\_1\_tau}, X)]$.

To verify our property we have to check that G8 holds. We do this by computing a BDD representation for G8 and testing that the result is the (unique) BDD representing the boolean function identically equal to 1.

4.4 One more property

The informal statement of this property is:

> *If the control system sends a signal to stop generator 1 then in at most 6 minutes either generator 1 is stopped or an alarm is sent to the operator.*

A logic formula for such property can be obtained as in the previous sections.

5 Experimental Results

In this section we describe our experimental results. We represent process P by representing, with BDDs, the transition relation of the automata defined by P. The BDD representing a process is obtained (manually) from the CCS/Meije syntax as illustrated in [EFT 91]. We only use standard BDD manipulation functions. Thus any BDD package can be used to carry out automatic verification. We used an in-house BDD package developed as part of a Boolean Functional Programming language [Tro 95]. Our BDD package is similar to the one described in [BRB 90], but we use shifted BDDs as in [MIY 90]. The main reason to use an in–house BDD package is source code availability. This allowed us to tune our package parameters to avoid running out of memory. We use a cache size of 19,997 and a hash table size of 200,003 with a load factor of 10. Garbage

collection is called each time there is at least a deletable BDD vertex and BDD size is greater than 2,000,030. This is the main reason for our long verification times. In fact, after about half an hour of computation most of the computation time is spent doing garbage collection.

To convince ourselves that our formal specification was a faithful representation of the given informal specification we ran experiments on subsystems of the overall system. This was done by trying to verify suitable properties (e.g. of set of reachable states, of admissible traces, ...) for each subsystem. This allowed us to find errors in the formal specification (i.e. our formal specification did not correctly represent the considered subsystem) as well as in the formulations of the properties we expected to hold. On the base of such experiments we revised our formal specification. The formal specification thus obtained was used to carry out the verification experiments (for the overall system) reported in this paper.

We define properties with μ-calculus formulas. We carry out automatic verification via model checking. Our model is the transition relation S *(present state, action, next state)* of the overall system. We use 15 boolean variables $a_0, \ldots a_{14}$ to code actions. State coding requires 174 boolean variables: $x_0, \ldots x_{173}$. Thus S is a boolean function of 15 + 2*174 = 363 boolean variables and represents a system with about 10^{52} states. Variable ordering was as in [EFT 91], i.e.: $a_0, \ldots a_{14}, x_0, y_0, x_1, y_1, \ldots x_{173}, y_{173}$, where $y_0, \ldots y_{173}$ are boolean variables representing the "next state".

When using BDDs the state space size is not a good measure of complexity since BDD size depends on the (symmetries of the) system transition relation (not just on its arity). Nevertheless it is worth noting that our state space size (10^{52}) is quite big compared to usual academic examples and to other published case studies in the process algebras area. E.g.: an 18 process Milner scheduler has about $5 * 10^6$ states (e.g. see [DB 95]); the alternating bit protocol (with buffer capacity 4 x 2) has 18278 states (e.g. see [DB 95]); the security management system verified in [CRB 94] has 312 states. Moreover our system does not have a symmetric structure (e.g. as Milner scheduler). Note however that for hardware systems industrial applications larger than ours have been studied. E.g. see [BCLMD 94, CGHJLMN 95].

Given a BDD representation for S, verification of a μ-calculus formula F amounts to evaluation of F in model S. This is done as in [BCMDH 92].

Building a BDD representing S was possible for systems with up to 3 generators. Experimental results are in the left table of figure 3. However when we try to automatically verify a property on a system with more than one generator we run out of memory. Thus to automatically verify the properties we studied we simplify our system as follows. We assume that only one generator is present in the system and that the overall system is working in automatic administration mode. Note that our plant can only work in two administration modes: automatic and manual. However the safety critical one is the automatic administration mode.

Our experiments were carried out with a SUN Sparc LX with 72MB RAM.

Note that this is a relatively small (and easily affordable) machine when compared with 512MB machines often used for automatic verification. Experimental results for verification of the properties in section 4 are in the right table of figure 3. Column CPU gives the time spent verifying a property after a BDD representation for the system has been built. Note that beside the above mentioned semplification of the system our approach is completely automatic and does not require any user expertise on the verification tool.

clocked_sys	CPU	Max BDD size
1 generator	20	1,251,627
2 generators	41	2,000,031
3 generators	289	2,611,348

1 generator/automatic	CPU	Max BDD size
property section 4.2	10032	2,000,031
property section 4.3	8322	2,000,031
property section 4.4	14431	2,000,031

Fig. 3. Experimental results on SPARC LX with 72MB RAM. CPU times are in minutes.

6 Conclusions

We have shown a formal analysis of a specification for a hydroelectric power plant. Starting from the informal specification we developed a formal one written using the CCS/Meije process algebra. For such formal specification we automatically verified that some given properties hold. We defined properties using μ-calculus. Verification was carried out using model checking and BDDs.

Our experience shows that automatic verification of modest size plants is feasible. However the size of the BDDs we had to handle ($> 2 \cdot 10^6$ vertices) shows that if we want to study larger systems we need global optimization techniques to automatically transform a verification problem into an easier one. This is particularly true if values (data path) are to be considered. Studying the possibility of using global optimization techniques as in [Tro 95] to avoid state explosion will be our next step.

Acknowledgements

We are grateful to anonymous referees for helpful comments on a previous version of this paper.

References

[AB 84] D. Austry, G. Boudol, *Algebre de processus et synchronisation*, Theoretical Computers Science, 1(30), 1984.

[BCLMD 94] J.R. Burch, E.M. Clarke, D.E. Long, K.L. McMillan, D.L. Dill, *Symbolic Model Checking for Sequential Circuit Verification*, IEEE Trans. on Computer-Aided Design, Vol.13, N.4, pp. 401–424, Apr. 1994.

[BCMDH 92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, L.J. Hwang, *Symbolic Model Checking: 10^{20} states and beyond*, Information and Computation, 98, (1992).

[Bry 86] R. Bryant, *Graph–Based Algorithms for Boolan Function Manipulation*, IEEE Trans. on Computers, Vol.C-35, N.8, Aug. 1986.

[BRB 90] K.S. Brace, R.L. Rudell, R.E. Bryant, *Efficient Implementation of a BDD Package*, 27th ACM/IEEE Design Automation Conference, 1995.

[CGHJLMN 95] E.M. Clarke, O. Grumberg, H. Haraishi, S. Jha, D.E. Long, K.L. McMillan, L.A. Ness, *Verification of the Futurebus+ Cache Coherence Protocol*, Formal Methods in System Design, Vol.6, N.2, pp. 217–232, Mar. 1995.

[CRB 94] O. Cherkaoui, N. Rico, A. Bernardi, *Specification and Analysis of a Security Management System*, FME 94, LNCS 873, Springer–Verlag.

[DB 95] A. Dsouza, B. Bloom, *Generating BDD Models for Process Algebra Terms*, CAV 95, LNCS 939, Springer–Verlag.

[EFT 91] R. Enders, T. Filkorn, D. Taubner, *Generating BDDs for Symbolic Model Checking in CCS*, Proceedings of CAV'91, Lecture Notes in Computer Science, 575, Springer–Verlag, 1991.

[ENEL 92] ENEL, *Descrizione informale di un caso di studio tratto dalle specifiche funzionali di un automatismo coordinatore delle manovre degli impianti idroelettrici*, Centro di Ricerca in Automatica, Rapporto Interno, Febbraio 1992.

[LP 94] S. Larosa, R. Pugliese, *Using the specification language CCS/Meije for a case study: a Software Control System of a Hydroelectric Power Plant*, Nota Interna B4-58, Istituto di Elaborazione dell'Informazione - CNR, Pisa, 1994.

[MIY 90] S. Minato, N. Ishiura, S. Yajima, *Shared Binary Decision Diagram with Attributed Edges for Efficient Boolean Function Manipulation*, 27th ACM/IEEE Design Automation Conference, 1995.

[PT 95] R. Pugliese, E. Tronci, *Automatic Verification of a Hydroelectric Power Plant*, Research Report SI/RR - 95/15, 1995.

[dSV 89] R. de Simone, D. Vergamini, *Aboard Auto*, Rapports Techniques 111, INRIA, Sophia Antipolis, 1989.

[Tro 95] E. Tronci, *Hardware Verification, Boolean Logic Programming, Boolean Functional Programming*, Proceedings of LICS 95, IEEE Computer Society.

A Meije: Syntax and Semantics

In this section, we give a brief presentation of the syntax and informal semantics of the CCS/Meije process algebra for reactive systems [AB 84]. More specifically we describe the subset of CCS/Meije we used in this paper to give our formal description of the ENEL Hydroelectric Power Plant in [ENEL 92]. We adopt the syntax used in the AUTO/MAUTO tools [dSV 89].

The syntax of the calculus is based on a set of elementary and uninterpreted actions that processes can perform and on a set of operators that permit to build complex processes from simpler ones. The syntax permits a two–layered design of *process terms*. The first level is related to *sequential regular terms*, the second one to *networks* of parallel sub–processes supporting communication and action renaming or restriction.

- *Act* is the set of atomic signal names ranged over by alphanumeric strings. Such names represent emitted signals if they are terminated by "!" or received ones if they are terminated by "?";
- τ denotes a special action not belonging to *Act*. Action τ representes the unobservable action (to model internal process communications);
- $Act_\tau = Act \cup \{\tau\}$, ranged over by a, denotes the full set of actions that a process can perform;
- X, ranged over by X, is the set of term variables.

The following grammar generates all regular terms, ranged over by R, and all network terms, ranged over by P:

$$R ::= \textbf{stop} \mid \mathrm{X} \mid a : R \mid R + R \mid \textbf{let rec } \mathrm{X} = R \text{ [}\textbf{and } \mathrm{X} = R\text{]} \textbf{ in } \mathrm{X}$$

$$P ::= R \mid P//P \mid P\backslash a \mid P[a/b] \mid \textbf{let } \mathrm{X} = P \text{ [}\textbf{and } \mathrm{X} = P\text{]} \textbf{ in } \mathrm{X}$$

where [...] denotes an optional and repeatable part of the syntax.

We give an intuitive semantics for the above constructs:

- **stop** is the process which does nothing;
- $a : R$ is the term that first executes action a and then behaves like R;
- $R + R$ is the nondeterministic composition between two regular terms;
- the construct $\mathrm{X} = R$ bounds the process variable X to the term R; then the **let rec** construct allows recursive definitions of processes;
- $P//P$ is the parallel composition between two network processes;
- $P\backslash a$ behaves like P apart from action a that can only be performed within a communication;
- $P[a/b]$ behaves like P apart from action b that is renaimed with a;
- the construct $\mathrm{X} = P$ bounds the process variable X to the network P; the let construct bounds nonrecursive definitions of process variables.

See [AB 84] for a more complete and formal description of Meije.