SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2000; **00**:1–7 *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

A Practical Use of Model Checking for Synthesis: Generating a Dam Controller for Flood Management‡

M.M. Gallardo, P. Merino, L. Panizo* and A. Linares§

University of Málaga
Campus de Teatinos s/n,
29071, Málaga, Spain

SUMMARY

Program synthesis with automated methods has been an active research area for many years; however, we still lack well known and accepted techniques for this software engineering task. In this case, the design space to be considered is infinite, even when the solution is restricted to software that meets the requirements. In this paper we propose the use of model checking techniques to automatically synthesize controllers. Given a goal in the evolution of a plant, model checking can be used to search for acceptable software controllers that enables the plant to evolve as desired. We also develop a realistic application in the context of a joint project with a major water reservoir management company. This application generates controllers for dam management during flood seasons. The controllers give the proper orders (open or close the outflow elements) at precise times in order to avoid disasters and to preserve the water level in the dam.

KEY WORDS: Automatic synthesis, model checking, flood management, software controller

*Correspondence to: Laura Panizo
University of Málaga, Campus de Teatinos s/n,
29071, Málaga, Spain
E-mail: laurapanizo@uma.es
‡This is a pre-print version of article Gallardo, M. M. et al. *A Practical Use of Model Checking for Synthesis: Generating a Dam Controller for Flood Management.* Software, practice & experience 41.11 (2011): 1329?1347.
https://doi.org/10.1002/spe.1048
§Befesa Agua SAU, Spain
Contract/grant sponsor: Spanish Ministry of Science, Andalusian Department of Science and Befesa Agua SAU; contract/grant number: TIN2008-05932, P07-TIC-03131)

Received
Revised

1. Introduction

Software engineers are permanently demanding more automatic tools that support task like code generation, test generation, test execution or automatic verification. The output of some of these tools is only a reduced set of solutions. For instance, verification or conformance testing checks whether the software satisfies or passes the requirements or test suites, giving a yes/no response. However, for other tasks, such as automatic synthesis, a tool should produce at least one program, if not several, that meets the expected requirements. A major problem with automated synthesis is the absence of known optimal solutions. Given a specification of what is expected from the program, if some solution exists, we can usually generate many programs that meet the expected requirements. In general, the design space to be considered is infinite. In this context, automatic software synthesis is attracting the attention of many researchers working on search-based techniques to obtain acceptable solutions by approximation. In this paper, we propose using *model checking*, the well known verification technique in the formal method community, as a novel approach to software synthesis. In particular, we develop a framework to produce efficient software controllers for dam management.

1.1. Automated Controller Synthesis

The management of many complex infrastructures such as industrial plants or water supply networks, requires the assistance of software in order to select the right control actions. It is even possible to temporarily replace the human operator with software controller in order to manage such critical systems. Thus constructing a reliable and automated software controller is also a critical task. As represented in Figure **??**, the controller has to issue orders to the real system regarding what actions have to be performed, taking into account the evolution of the real system (using sensors). The controller should also satisfy functional constraints and specific goals defined for the real system. Functional constraints are usually given by the available resources. Goals are defined by the user to optimize some aspects, such as the time or the energy needed to assess a task, the time to stop a plant, the water level in a dam, etc. It is clear that, as shown in Figure **??**, the controller's decision space is a tree and not a single sequence of actions. The aim of a good automated synthesis technique is to reduce the tree to one single sequence, or to just a few, that meets the constraints and goals.

The first approaches to controller synthesis were based on analytical methods and hand made construction of the program. For instance, [**?**], [**?**] and [**?**] entail analytical works based on control theory. These works do not provide a tool for synthesis and the proposals can only be evaluated with simulators, and therefore are not considered in the category of automated synthesis.

Only taking into account approaches to automated controller synthesis, the main differences among existing works depend on how to solve several problems:

- modeling the system under control (SUC)
- representing the constraints
- searching for a solution (a software controller)
- measuring the quality of a potential solution

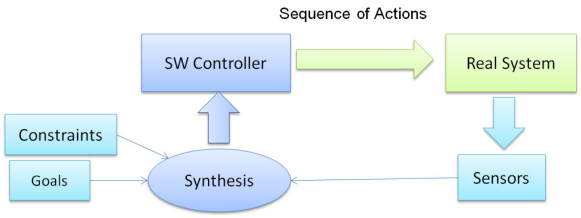


Figure 1. The role of synthesis

In the context of formal methods, most works consider automated synthesis as a constraint satisfaction problem where the SUC is modeled as a tree of global states and the search method is a variant of exhaustive search using the constraints as a guide. Each global state contains all the current information for both the SUC and the controller. This approach has been applied to scheduling industrial plants [**?**], scheduling for planned shutdown of power plants [**?**], production cells [**?**], mechanical CAD [**?**], development of an automotive cruise controller [**?**] or optimal controllers for finite state specification [**?**].

Regarding the quality of the solutions, some proposals rely on a post-synthesis analysis and others use some kind of fitness function in order to lead the process to an acceptable solution. For instance, in the first category, Bhansali and Hoar [**?**] use extensive testing and evaluation of the solvers synthesized by their solver generator. In the second category, the tool ROMAN, by Gomes et. al [**?**], ensures that a scheduling task can be safely started choosing any start time in a window, so that in practice one solution is actually a set of valid solutions.

1.2. Using Model Checking for Synthesis

Model checking (MC) [**?**, **?**] is an efficient verification technique to search and check all the potential execution paths in a concurrent system in order to locate design errors (such as deadlock, memory violations, etc.). When all traces are analyzed and no errors are found, the program is considered free of errors or verified with respect to a given set of properties. In practice, desirable properties are described with *temporal logic* and verified using *automata*. Given a temporal logic formula F to be satisfied by all execution paths, its negation *not* F is translated into the automaton A that recognizes bad behaviors. The model checker runs the automaton in parallel with every path generated by the software and produces a *counterexample* when a trace is recognized by the automaton.

The basis of MC is the controlled execution of the software, described with a modeling or a programming language, in order to generate the software's reachability graph. This controlled execution is enriched with methods to search for errors in a very efficient way. SPIN[**?**] is probably the most well known example of this type of tools. It uses PROMELA as the modeling language and temporal logic and Büchi automata for properties. PROMELA allows us to describe

software with different abstraction levels, including the use of non-determinism to represent several potential or unpredictable behaviors.

We propose using the non-deterministic features of modeling languages like PROMELA and the search capabilities of model checking as a tool for automatic software controller synthesis. The main idea is to obtain a good controller from an initial version that behaves in a non-deterministic way. Actually, maximum non-determinism in the controller implies that its reachability graph contains all the possible deterministic designs (as shown in Figure **??**). Like other approaches to synthesis, our method consists in approximating from the non-deterministic to an acceptable deterministic controller through transformations. In our case, transformation means removing non-determinism, and it is done taking advantage of MC in the following way:

- The whole system to be controlled is modeled in a precise way, using the MC tool's modeling language.
- The controller is modeled as a non-deterministic process that interacts with the rest of the system.
- The specification of the system's expected behavior produced by an acceptable controller is described with the automata for properties used by the model checking tool.
- The model checker works as usual, searching the state space produced by the model (including the non-deterministic controller) and reporting the counterexamples that fit the automata.
- The counterexamples are evaluated with some optimization criteria, and one of them is used to make the controller's code deterministic. Then this code can be easily translated to the actual programming language used by the control platform.

Note that unlike the usual interpretation of results, we consider counterexamples as good traces that guide us in the construction of the controller.

In the paper we present three main results:

1. We have implemented our approach using the model checker SPIN. The SUC is modeled as a concurrent system in PROMELA, and is extended with a model of a non-deterministic controller. Then we define a *goal automata* using the SPIN notation for Büchi automata. SPIN produces counterexamples satisfying the goal automata that are used to encode the software controller in standard programming languages, like C.
2. Apart from developing the SPIN based framework for synthesis, in the paper we develop a realistic application in the context of a joint project with a major company in water reservoir management. This application entails the generation of a controller for dam management in flood seasons. The controller gives the proper order to open and close spillways at precise times in order to prevent disasters and to preserve enough water in the reservoir. This is a critical application due to the possible negative effects of bad decisions on the population affected by the dam.
3. The dam system, like most realistic industrial plants or artificial installations to be controlled, should be modeled as a time dependent system with discrete and continuous variables. Thus we have to extend the standard use of PROMELA and SPIN to define clocks

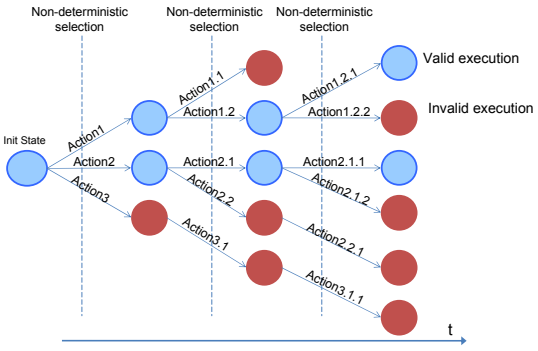


Figure 2. Controller execution graph

and also to represent the equations that governs the dam's behavior. These extensions can also be used for other applications.

Our work presents two novel contributions with respect to related work. On the one hand, our approach considers the application domain with explicit executable models for every element. Other approaches, like genetic programming for controller synthesis, do no focus on the real application and the model does not exist. From our experience, having the model allows us to consider aspects such as time and simplifies the definition of goals and constraints. On the other hand, the method is reliable in the sense that if a solution exists, it can be found. This is due to the exhaustive search produced by the model checking algorithm. Section **??** contains a detailed comparison with related work.

1.3. Organization of the paper

The paper is organized as follows: Section **??** introduces the case study, the synthesis of a controller for dam management, and also relates other traditional methods for dam management. Section **??** gives an overview of the model checking tool SPIN and its modeling language PROMELA. Section **??** explains in more detail the methodology used to synthesize the controller for dam management using the model checking tool SPIN. It also shows some experimental results obtained. Section **??** shows a comparison with related work. Finally, Section **??** summarizes the conclusions and presents future work.

2. A Running Example: a Dam System

Dams are complex water management systems composed of different elements that present a time dependent behavior. A dam can have different objectives; the most common are the regulation of floods and water supply for human consumption and irrigation. To meet

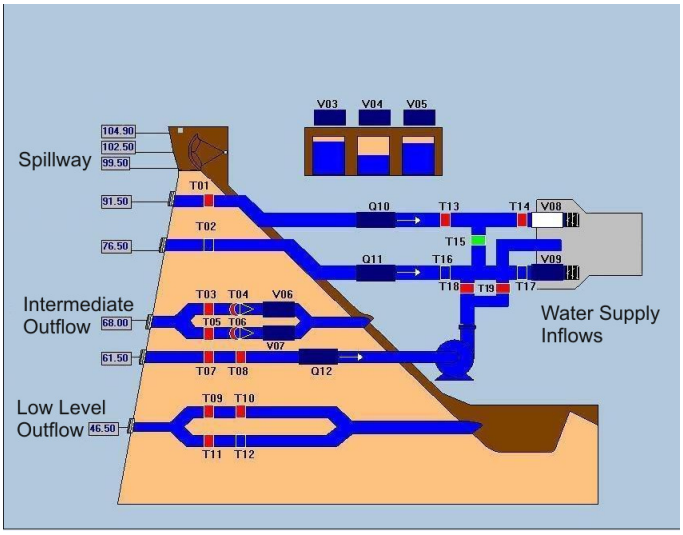


Figure 3. Dam section

these objectives, different types of outflow elements are used, Figure **??** shows some of them. Spillways, Intermediate Outflows and Low Level Outflows are gates for flood regulation. Their outflow capacity depends on the dam level and their opening degree. Water Supply Inflows are used to obtain water for human consumption and irrigation. They are part of the water supply system, which provides water independently of the water height.

Water resource management is a very controversial issue. Dam management is traditionally carried out by a human operator, who has to manage in parallel the different outflow elements (spillways, intermediate outflows and low level outflows). This task is vital and very complex, especially in flood scenarios. Decision Support Systems (DSS) are the most used systems to help in the management of river basins, dams or other systems related with water resources. Simulation models have been the primary tool for reservoir planning and management. Simulation allows a detailed and faithful representation of a real-world system with complex mathematical models. The main drawback of simulation is that it requires prior specification of the system operating policy, having to carry out a huge number of trials to establish an optimal policy. Giupponi et al. present the *mDSS* in [**?**, **?**], a DSS tool that integrates environmental models (especially hydrological) with multi-criteria evaluation procedure. In [**?**], Koutsoyiannis et al. present a DSS that incorporates a stochastic simulator block and an analysis block based on the parameterization-simulation-optimization methodology. In [**?**], Hasebe et al. studied the use of neural networks and fuzzy systems in dam management. Their DSS is divided into an operation policy module, which can be implemented with neural networks or fuzzy systems, and an operation quantity module, which can only be implemented with the fuzzy system. Bagis

and Karaboga [**?**] also implemented a fuzzy logic controller for spillway gates with an optimum rule controller. The rule base is optimally determined by using a tabu search algorithm.

$$Q_s(t) = CL\sqrt{h_s(t)^3} \tag{1}$$

$$Q_s(t) = CL(\sqrt{h_{s1}(t)^3} - \sqrt{h_{s2}(t)^3}) \tag{2}$$

$$V(t) = V(t-1) + \mathit{Inflow}(t) - \sum_{i=1}^{n} Q_{s_i}(t) \tag{3}$$

In Equation (**??**), $Q_s(t)$ defines the outflow capacity of an specific spillway s at time t. This value depends on the height $h_s(t)$ of the stored water placed over the spillway aperture which, in fact, constitutes the effective water that is being discarded. It depends on the degree of aperture of spillway s at each time instant t. In addition, C is a coefficient related to water height, and L is the useful width of the spillway. To practically calculate Q_s, we use Equation (**??**) where the heights of two distinguished points in the spillway, h_{s1} and h_{s2}, allow us to determine the effective degree of aperture of s by means of $h_{s1} - h_{s2}$. The evolution of dam volume $V(t)$ over time is given by Equation (**??**). In this expression, we assume that function $\mathit{Inflow}(t)$ gives us the water flow entering to the dam at time instant t. In addition, we suppose that the dam under analysis has n spillways, denoted as $s_1, \cdots, s_n$. The dam we are using as a case study only has 3 spillways.

The dam can be characterized as a hybrid system: it presents continuous behaviors that can be expressed by equations (outflow capacity of the different dam elements) and discrete behaviors, such as the opening of the outflow elements. Although Equation (**??**) shows that $(h_1 - h_2)$ can take any real value, in real dam management, only a finite set of openings are used. The different actions that can be performed over the real dam depend on the evolution of continuous variables, which varies depending on the discrete state. There are also non-deterministic events, such as rain inflow or malfunction of some elements of the dam, that can influence the global behavior. All these factors increase the design space of hybrid systems and incorporate more complexity regarding the synthesis of controllers.

Note that decisions need to be made and actions taken within a few hours of rain starting. It is desirable to have a controller for flood scenarios that could replace the human operator. Its main task would be to guarantee the safety of the dam and keep the water level under the correct height. In addition, the controller could achieve more sophisticated goals, such as minimizing the time needed to maintain the water level within a specific range, or reducing the number of open outflow elements.

2.1. Synthesis of a Dam Controller

The different outflow elements of a real dam can be opened and closed in multiple position and at any time. The controller results from searching among all the possible configurations of dam elements over time, one that meets all the defined objectives. These objectives are expressed as goals and constraints. Constraints are usually related to available resources: in our case study the resources are the available outflow elements. If a real gate is out of order or has limited its outflow capacity, we define a constraint about its use. Goals are related to

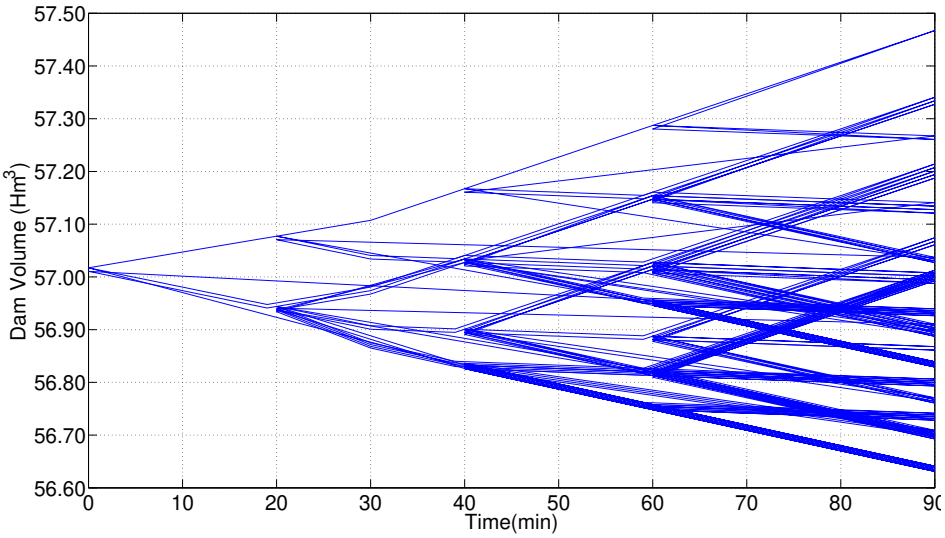


Figure 4. Evolution of dam level

objectives imposed by the dam manager, such as keeping the dam at a level for a period of time or minimizing the amount of released water. An important goal in dam management is to keep the water under the safety level to avoid its collapse.

In our case study, the controller is synthesized for a specific flood scenario, and each time this scenario is modified it is necessary to synthesize a new controller. The different configurations of dam elements cause dam parameters to evolve in different ways. Only those configurations that fulfill all the constraints and all the goals can be considered to synthesize the controller.

Figure **??** shows the evolution of dam volume over time for a specific scenario; the branches are caused by the different configurations of the dam elements. This graph reflects both the valid and invalid evolutions of dam volume, and only one of the valid evolutions will be used to synthesize the controller. The selection can be made randomly or oriented to the optimization of some parameters, such as time or similarity with the original desired behavior. In the next section we use the well known model checking tool SPIN to search for different controllers for the dam.

3. The Model Checker SPIN

In the last few years, SPIN [**?**, **?**] has become one of the most employed model checkers in both academic and industrial sectors. It supports the verification of safety properties in distributed software systems written in the modeling language PROMELA. Properties can include complex requirements expressed with Linear Temporal Logic (LTL). Given the model and the property,

SPIN performs a depth-first search to check whether the system holds the property. In this section some concepts of the SPIN model checker will be presented. The first subsection introduces the modeling language PROMELA. Then, we will explain how to define properties with LTL and Büchi automata.

3.1. PROMELA Modeling Language

PROMELA is SPIN's modeling language. It is an asynchronous and non-deterministic language. It was designed to describe systems composed of concurrent asynchronous communicating processes. A PROMELA model $P = Proc_1||...||Proc_n$ consists of a finite set of processes, global and local channels, and global and local variables. Processes communicate via messages passing through channels. Communication may be asynchronous using channels as bounded buffers or synchronous using channels with size zero. Global channels and variables determine the environment in which processes run, while local channels and variables establish the internal local state of processes.

Figure **??** shows an example of a PROMELA process modeling the basic behavior of a dam with 3 spillways. The process is defined as a sequence of possibly labeled sentences preceded by the declarative part. Basic sentences in PROMELA are those that produce a definite effect over the model state, in other words, the assignments (`id = id+1`), the instructions for sending and receiving messages (`cmd[id]?_`), and the Boolean expressions, which include tests over variables and channels contents (`cmd[id]?[open1]`). In addition, PROMELA has other non-basic sentences like the non-deterministic selection `if`, the loop `do` and the definition of `atomic` blocks to execute code atomically.

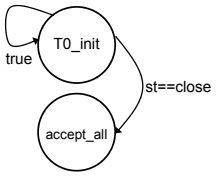
3.2. LTL and Büchi Automaton

SPIN verifies LTL formulas against PROMELA models. Well-formed formulas of linear temporal logic are inductively constructed from a set of atomic propositions (in PROMELA, propositions are tests over data, channels, or labels), the standard Boolean operators, and the temporal operators: always "$\Box$" , eventually "$\Diamond$" , next "$\bigcirc$", and until "U". Formulas are interpreted with respect to model state sequences $t_i = s_i \rightarrow s_{i+1} \rightarrow \ldots$. Each sequence expresses a possible model execution from state s_i. The use of temporal operators permits the construction of formulas that depend on the current and future states of a states sequence. By default, given the LTL formula, SPIN translates it into an automaton that represents it as an undesirable behavior (which is claimed to be impossible); this is a Büchi automaton. Using a LTL formula, we can check for instance if spillway 0, included in the dam model of Figure **??**, is never in the *close* position for all the possible executions, meaning that the spillway is always open. The LTL formula for that property is: $\Box$ (`open`), where `open` is defined as `c_expr{(sp_st[0]==open1)||(sp_st[0]==open2)||(sp_st[0]==open3)}`.

In general, SPIN implements the evaluation of properties over models by constructing the synchronous product of the property automaton (i.e. Figure **??** shows the never claim for formula $\Box$ (`open`)) and the system model. The property is satisfied only if the property automaton never reaches the acceptance state `accept_all` for all the executions. Note that the automaton has been generated by translating the negated formula.

```
#define close 0
#define open1 1
#define open2 2
#define open3 3
int n, t_dam;  //global timer and timer of Dam process
chan cmd[3] = [1] {bit};
c_decl {#include "./CCode.c"
            double contribution[3];};
c_track "&contribution" "sizeof(double[3])" "UnMatched";
...
proctype Dam(){
 int id = 0;
 do
 ::(n==-1)->break;
 ::else->do
     ::(id<max_spillway)->atomic{if
           ::(cmd[id]?[open])->if
                 ::c_expr{sp_st[PDam->id]== close}->cmd[id]?_;
                   c_code{sp_st[PDam->id]= open1;};
                 ::c_expr{sp_st[PDam->id]==open1}->cmd[id]?_;
                   c_code{sp_st[PDam->id]= open2;};
                 ::c_expr{sp_st[PDam->id]==open2}->cmd[id]?_;
                   c_code{sp_st[PDam->id]= open3;};
                 ::else->cmd[id]?_;
                 fi;
           ::(cmd[id]?[close])->if
                 ::c_expr{sp_st[PDam->id]==open1}->cmd[id]?_;
                   c_code{sp_st[PDam->id]=close;};
                 ::c_expr{sp_st[PDam->id]==open2}->cmd[id]?_;
                   c_code{sp_st[PDam->id]= open1;};
                 ::c_expr{sp_st[PDam->id]==open3}->cmd[id]?_;
                   c_code{sp_st[PDam->id]= open2;}
                 ::else->cmd[id]?_;
                 fi;
           ::else ->skip;
           fi;
           c_code{contribution[PDam->id]=getcontribution(sp_st[PDam->id], dam_h,now.ap);};
           id= id +1;}; //end atomic
     ::else-> id=0; break;
     od;
     delay(t_dam,1);
 od;}
```

Figure 5. PROMELA code of the dam model




```
#define open c_expr{(sp_st[0]==open1)||(sp_st[0]==open2)||(sp_st[0]==open3)}


never {    /* !([]  open) */
T0_init:
   if
   :: (!(open)) -> goto accept_all
   :: (1) -> goto T0_init
   fi;
accept_all:
    skip;}
```

Figure 6. N_1: Goal automaton $\Box$ (`open`)

4. Using SPIN for Controller Synthesis

In this section we explain how to use SPIN as a synthesis tool; we also apply our methodology to synthesize controllers for dam management. Figure **??** shows our approach to solve this problem. The first step is to represent the different aspects of the real system in PROMELA, including the hybrid behavior of the outflow elements into the dam model. It is also necessary to include a model of the dam's environmental inflows (river flow, rain, snow melting, etc.) and finally a non-deterministic model of the dam user/controller. These elements constitute the global PROMELA model that represents the real system in a specific flood scenario. After that, we have to define the desired evolution of some dam parameters (constraints and goals) as a goal automaton. The automaton is checked against the PROMELA model using SPIN. If the analysis of SPIN returns any counterexample, it represents a suitable behavior that matches the goals and constraints. Finally, to synthesize the controller, it is necessary to extract from the counterexample the timed behavior of the user/controller. In the rest of the section we give more details of our methodology.

4.1. The Model of the Dam

In order to develop a model of the dam, we need to specify the continuous behavior (equations) and the discrete states, and how both behaviors evolve over time. PROMELA was originally defined to model discrete systems, so that, some aspects related with continuous systems, such as time representation, were not included. Modeling hybrid systems in PROMELA entails two main challenges:

1. To implement a model of time that allows us (a) to carry out time-bounded analysis, (b) to discretize the numerical models, and (c) to model timed processes.
2. To model continuous behaviors by integrating numerical models in PROMELA.

Figure **??** shows the PROMELA process that represents the dam. In this implementation we have included 3 spillways each one with a discrete variable, called `sp_st`, that stores the opening position (*close*, *open1*, *open2* and *open3*) and a continuous variable, called `contribution`, that follows the Equation (**??**) of outflow capacity given in Section **??**. Each time the `Dam` process

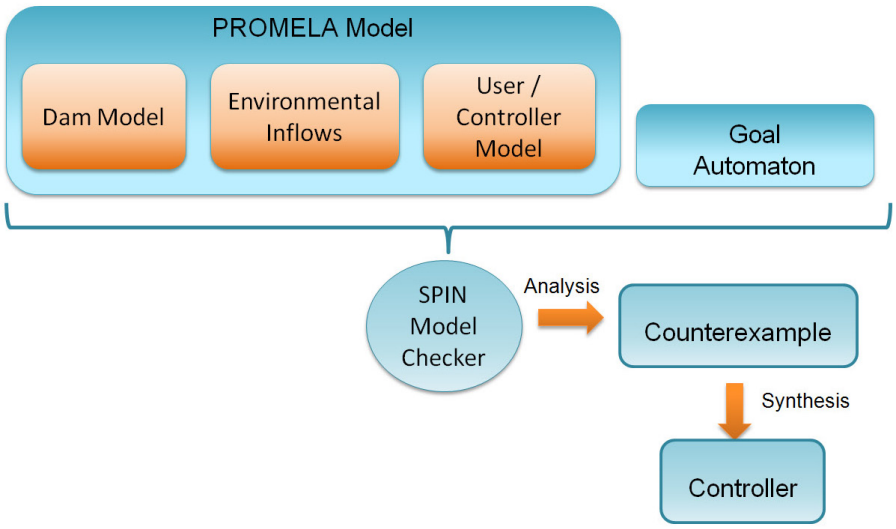


Figure 7. Synthesis tool chain

is executed, it updates the state of each spillway if any command was received, and works out its contribution to the total outflow capacity of the dam for the current time period.

4.1.1. Modeling Time

We have developed a synchronization model to solve the first challenge of modeling hybrid systems with PROMELA. As a starting point we have used the discrete time model defined in [**?**]. Figure **??** shows an example of use of this synchronization mechanism. It is based on timer variables and a synchronization process called `Timer`. This process controls the execution of the other processes (in this case instances of `hybridproc1()`). From now on, these processes are called hybrid processes. The `Timer` process is only executed when all the hybrid processes are blocked. In a well defined model, a hybrid process is blocked when it reaches the instruction `delay(timer,N)`, which suspends it during `N` time steps. Each hybrid process has its own timer variable, and all the active timers are updated when the `Timer` process is executed (by means of instruction `tick(timer)`). In addition, the time model includes a global timer, denoted as `n`, that measures the time elapsed since the beginning of the analysis. The global timer is useful to perform a time-bounded analysis and to introduce time requirements in properties.

```
#define timer int
#define set(x,y) x=y
#define expire(x) (x == 0)
#define tick(x) if :: x>=0 -> x=x-1; :: else; fi
#define on(x) (x!=-1)
#define delay(x,y)set(x,y); expire(x);
#define st1 0
#define st2 1
bit state;
timer timer1 = -1; timer n= 60;
int var1;

active proctype Timer(){
  do
  ::(timeout)->if
              ::n>0->atomic{tick(timer1);tick(n);};
              ::else->break;
              fi;
  od;}
active proctype hybridproc1(){
loop:if
     ::(n>0)->
           if ::var1<const1->state=st1;
              ::var1>const2->state=st2;
           fi;
           if ::(state==st1)->var1=var1+3;delay(timer1,1);
              ::(state==st2)->var1=var1-1;delay(timer1,1);
           fi;
           goto loop;
     ::else->skip;
     fi;}
```

Figure 8. PROMELA synchronization mechanism skeleton

4.1.2. Integrating the Equations

We have to introduce the numerical models (Equations (**??**) and (**??**)) that represent the continuous behavior of the spillway in the dam process. We have used the embedded C code extension of PROMELA to perform this task. This extension is based on a set of primitives that allows the definition of C variables (*c_define* and *c_track*), execution of C instructions (*c_code*), and the evaluation of guards with C variables (*c_expr).* In Figure **??**, the file `CCode.c` is used to declare C variables and functions, such as the `getcontribution` function or `dam_h` variable.

It is worth noting that Equation (**??**), which depicts the evolution of dam volume, is an equation in differences that allows us to work out the dam level in discrete time steps, using the synchronization mechanism presented in the previous subsection to make time pass. We assume that the outflow capacity of a spillway (Equations (**??**) and (**??**)) is constant in a time step, due to the length of a time step is smaller than the time need to modify the opening degree

```
proctype Timers()
{
  do
  ::timeout->
       if
       :: c_expr{n!=-1}-> atomic{
                c_code{
                  out_flow = time_step*(contribution[0]+contribution[1]+contribution[2]);
                  dam_level = dam_level- out_flow+ in_flow;
                  dam_h=vol2height(dam_level, dam_h);
                };
                tick(n); tick(t_dam); tick(t_user); tick(t_rain);};
       ::else-> break;
       fi;
 od;
}
```

Figure 9. Synchronization process for dam case study

of a spillway and it is also smaller than the time needed to make the height of stored water change drastically. The function `getcontribution` is responsible for computing the contribution of a spillway each time step. Both parameters, opening degree and water height, are used to work out the spillway contribution.

The C code extension of SPIN is also useful to abstract data from the model; for instance, in the code of Figure **??** we may observe that array `contribution` is tracked (i.e., it is completely stored in the state space) but, in the matching process, we use an abstraction of its current value: the sum of its three components (`out_flow` = `contribution[0]` + `contribution[1]` + `contribution[2]`) as can be seen in Figure **??**. That fact notably reduces the state space and it allows tracking the `contribution` variable.

Abstraction of some variables by means of the so-called *abstract matching* technique analyzes a subset of the original system, thus it has to be carefully carried out to avoid missing significative counterexamples and, therefore, to guarantee the completeness of the model analysis. In case of the dam analysis, we have defined this variable abstraction, because it is not relevant which spillway is contributing, but the whole outflow capacity of the dam. So, for instance, if a counterexample returns that only spillway 0 is open, we could also extend this solution to the case when only spillway 1 or 2 is open. More information of this extension and how to use it to abstract data from the model can be found in [**?**, **?**, **?**].

4.2. The non-deterministic Controller Skeleton

A key element of our approach is the initial model of the controller. It is in charge of sending commands to the different outflow elements of the dam. Its behavior is an over-approximation of the acceptable behavior of a controller that meets the dam's desired evolution. Different strategies can be used to develop such a non-refined controller. The most powerful is one that

```
proctype Controller(){
  int d_cmd= 20;
  do
  ::(d_cmd<=n)->atomic{
      if
      ::(1)-> cmd[0]!close; //Close 1 position spillway 1
      ::(1)-> cmd[0]!open; //Open 1 position spillway 1
      ::(1)-> skip; //Do nothing
      fi;
      if
      ::(1)-> cmd[1]!close; //Close 1 position spillway 2
      ::(1)-> cmd[1]!open; //Open 1 position spillway 2
      ::(1)-> skip; //Do nothing
      fi;
      if
      ::(1)-> cmd[2]!close; //Close 1 position spillway 3
      ::(1)-> cmd[2]!open; //Open 1 position spillway 3
      ::(1)-> skip; //Do nothing
      fi;
      delay(t_user,d_cmd);};
  ::else-> break;
  od;}
```

Figure 10. PROMELA non-deterministic controller

sends commands non-deterministically to all the outflow elements at any time. This approach produces a huge search state space, and can lead to state space explosion problems. Figure **??** shows an example of a non-deterministic controller model. It can open or close the position of each spillway by one degree or leave it in the last position. To reduce the state space and give a realistic behavior, we define a minimum time elapsed between two commands. It is called `d_cmd` and in that case is set to 20 minutes (one minute is equal to one time step). The `Controller` process is composed of three `if` blocks that can select randomly to send, or not, a command to each spillway using the array of global channels `cmd[]`.

4.3. Introducing Time in goal automata

We use the term *goal automaton* for the mechanism used to express the goals and constraints imposed on the dam's evolution. This goal automaton is not a SPIN structure, but can be implemented using mechanisms such as the `LTL` formulas and the Büchi automaton introduced in Section **??**.

Since we are modeling the dam as a hybrid system whose behavior has a close relation with time, it is desirable to introduce the concept of time in the goal automaton and more specifically in the final never claim that uses SPIN to carry out the analysis. In Section **??** we have discussed the concepts of Büchi automata and never claims. To understand how we deal with time in properties, it is necessary to recall some specific aspects of the semantics

```
#define open c_expr{(sp_st[0]==open1)||(sp_st[0]==open2)||(sp_st[0]==open3)}
#define inInterval (n>=10 && n<=20)

never {    /* (!([](inInterval -> open)))*/
T0_init:
   if
   :: (!(open)&&(inInterval)) -> goto accept_all
   :: (1) -> goto T0_init
   fi;
accept_all:
    skip;}
```

Figure 11. N_2: Basic goal automata with time

of never claims. The synchronous product of the property and the model proceeds whenever it is possible to transit in both automata. However, if the property automaton is blocked (no transition is enabled), the search algorithm discards the current execution trace, and it backtracks until the last state where transition is possible. This behavior is used by SPIN to prune the search when it is known that the current trace does not satisfy the property.

In this section, we extend the example given in Figure **??** to describe how to introduce time in never claims. In particular, the objective is to check whether the first spillway (`spillway[0]`) is always `open` during the time interval `[10,20]`. Assuming that variable `n` stores the global time, this property may be expressed by the LTL formula `[] (inInterval` $\rightarrow$ `open)`, where conditions `inInterval` and `open` are respectively defined as `n`$\geq$`10 && n`$\leq$`20` and `c_expr{(sp_st[0]==open1)||(sp_st[0]==open2)||(sp_st[0]==open3)}`.

Figure **??** shows the never claim automatically generated by SPIN by translating the previous temporal formula (denoted by N_2). It is worth noting that this new never claim includes the additional condition `inInterval` in the loop. Therefore, the property is only violated on a given executed trace if a state is found where the global time is in the interval `[10,20]` and the first spillway is closed. However, if this state does not exist, the trace is completely explored even if the global time has exceeded value `20`.

Figure **??** illustrates a never claim that optimizes the previous one (denoted by N_3). Observe that in this new automaton, we have substituted condition `true` (written as `1`) by the new boolean expression `BInterval` to check whether the global time has reached the beginning of the interval (`n<10`). This new form of introducing time is more efficient, since it exploits the capabilities of never claims to prune the search. As in never claim N_2, this automaton stops the search when it is exploring an execution trace and a state is found that violates the condition `(open)&&(inInterval)`. However, in contrast to N_2, automaton N_3 discards an execution trace if no state is found that violates the property before the global time reaches value `20`. If we look at the structure of N_3, that occurs when none of the guards of the `if` loop are active. As a consequence, N_3 allows SPIN to reduce the number of explored states by using the value of global time. In N_2, it is mandatory to completely explore each execution trace due to the condition `true`. In Section **??**, Table **??** shows a comparison between using the basic automaton(N_2) or the optimized (N_3).

```
#define open c_expr{(sp_st[0]==open1)||(sp_st[0]==open2)||(sp_st[0]==open3)}
#define inInterval (n>=10 && n<=20)
#define BInterval (n<10)
never {
T0_init:
   if
   :: (!(open)&& inInterval) -> goto accept_all
   :: (BInterval) -> goto T0_init
   fi;
accept_all:
    skip;}
```

Figure 12. N_3: Optimized goal automata with time

4.3.1. Complexity Issues

In this section, we discuss the complexity aspects of the nested depth search algorithm used by SPIN to perform the analysis. As commented in G.J. Holzmann's book [**?**], the use of memory in the algorithm does not significantly impact the memory requirements with respect to the linear algorithm that is applied when analyzing safety properties (the ones that should be true on each reachable state). On the other hand, when a never claim is added, the reachability analysis is increased by the number of states of the never claim automaton. In addition, if this automaton comes from an `LTL` property, the number of states of the automaton exponentially depends on the number of operators of the `LTL` formula, such as in the other algorithm proposed [**?**, **?**]. However, in the case of SPIN, this exponential effect is rarely seen in practice. In addition, as described in the previous subsection, the way we deal with global time in never claims can reduce the number of states explored during the analysis.

4.4. Experimental Results

The analysis performed by SPIN generates a huge state space that contains all the possible configurations of dam elements. We are interested in guiding the search to reduce the state space and to find a suitable evolution that matches all the goals and constraints.

We have used the following system configuration: the time horizon of the analysis is 90 or 180 minutes; in our synchronization mechanism one minute is equal to one time step. The dam is composed of 3 spillways which are initially `close`. Figure **??** shows the model. The environmental inflow model has a constant rate of $50m^3/s$ in the interval from 0 to 30 minutes and a constant rate of $100m^3/s$ until the end of the analysis. The user/controller model can send non-deterministically the same command (`open` or `close`) to the 3 spillways at the same time. The delay between two commands, denoted as `d_cmd`, is constant in each test and can take two different values, 15 and 20 minutes. The commands are sent if there is at least `d_cmd` time units of analysis. The verification was carried out in an Intel Core2 Quad with 2.40GHz and 3GB of RAM. The operating system is Windows XP Professional SP2 and the SPIN version is 5.2.0.

Table I. Results of SPIN for invalid end state analysis

		State Vector	Depth	States Stored	States Matched	Transitions	Total Memory	Elapsed Time(s)
Analysis Time=90	delay=15	88	2,830	240,133	68,846	398,979	28.099	3.200
	delay=20	88	2,812	75,550	19,600	95,150	12.376	0.625
Analysis Time=180	delay=15	88	5,614	5,020,911	1,541,156	6,562,067	521.126	64.600
	delay=20	88	5,572	1,710,746	490,578	2,201,324	168.235	19.300

This configuration is very similar to a realistic dam management scenario. Usually a dam scenario has a duration of 1 or 2 days and the dam operator has to make decision every 2 or 3 hours, so our analysis time is not far from real decision times. The dam operator waits a minimum amount of time, usually one hour, between two maneuver to see the results of the earlier decisions, so it is logical to define a `d_cmd` parameter.

4.4.1. Invalid End State Analysis

We have performed a first analysis to check if there is any invalid end state in the PROMELA model. This analysis generates the whole state space and ensures that there are no deadlocks. Figure **??** shows the evolution of the dam volume for all the possible operation sets for `d_cmd=20` and the previously specified flood scenario. The different branches are due to the presence of a non-deterministic controller. Table **??** shows the stats of the invalid end state analysis; the two first rows show the results for a time analysis of 90 minutes with `d_cmd=15` and `d_cmd=20` respectively. The third and the fourth rows show the results for an analysis time of 180 minutes. It is clear that the decrease of `d_cmd` introduces more branches in the state space tree. We can see how increasing the number of branches increase the number of states and consequently the memory used and the analysis time are higher.

4.4.2. Analysis with the goal automaton

Now we want to see how the search can be guided by means of a goal automata. For this first attempt, our goal, called (a), is to keep the dam level in the range $[57.12, 57.22]Hm^3$ ($1Hm^3 = 10^6m^3$) from time 80 to 90 using `d_cmd=20`. The SPIN analysis can find several counterexamples; all of them fulfill the goal. Figure **??** shows the evolution of dam volume and the position of the spillways for two of them. If we use the counterexample on the left to synthesize a controller, the spillways will remain in `close` position until the controller sends to all spillways the command `open` at time 40, that cause all of them to be in `open1` state until the end of the analysis. Another valid option is to use the counterexample on the right. The synthesized controller will behave differently: spillways will remain in `close` position until the controller sends `open` at time 20, and remains in state `open1` until the controller sends `close` at time step 60. As the official regulations to manage dams still gives the operator the ultimate

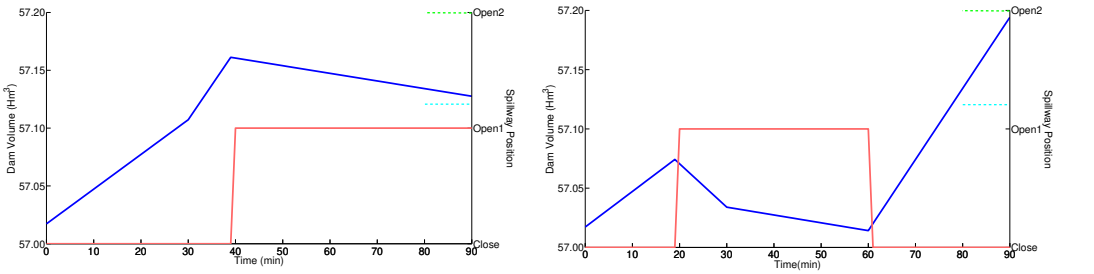


Figure 13. Valid evolutions for goal (a)

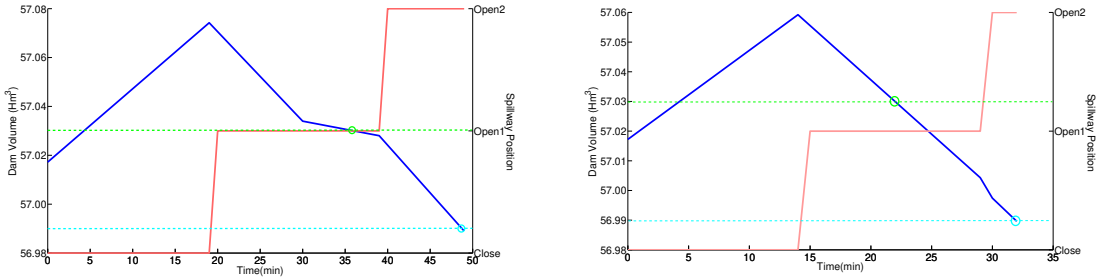


Figure 14. Counterexample goal (b) left d_cmd=20 and right d_cmd=15

decision, we do not implement a decision mechanism to automatically select the most suitable counterexample. At present we have configured spin to return all the counterexamples (or a elevate number of them) to the user. The selection should be made manually based on the dam operator's experience and the outflow policy at the dam. The operator can always reduce the number of counterexample adding more constraints and more information to the goals. These features will be considered in the future by the user friendly GUI. In this case, the use of the counterexample on the left seems to be more suitable, because the average dam level is higher than in the counterexample on the right: that means that less water is wasted. The two first rows of Table **??** show the stats of the analysis of goal (a) with the basic automaton and with the optimized goal automaton introduced in subsection **??** for the same value of `d_cmd`. We can see a reduction in the number of states when using the optimized never claim due to the fact that the search can be pruned when the global time exceeds the deadline given in the property. The general improvement can vary depending on the system under analysis and the goal. It is worth noting the analysis with the optimized goal automaton is never worse than with the basic automaton. This is because with the optimized automaton the value of the global time may help to prune some execution paths.

Table II. Results of SPIN for goal (a) and (b)

		State Vector	Depth	States Stored	States Matched	Transitions	Total Memory	Elapsed Time(s)
Goal (a)	delay=20	92	3,649	4,100	814	4,914	5.638	0.078
	delay=20 opt	92	3,622	2,935	524	3,459	5.540	0.062
Goal (b)	delay=20	96	3,651	202,558	61,441	263,999	25.169	2.500
	delay=15	96	1,376	9,385	2,719	12,104	6.224	0.171

We have defined another goal automaton, denoted (b), to detect if it is possible to decrease the dam volume from values greater than $57.03Hm^3$ to values lower than $56.99Hm^3$ in less than 20 minutes. In this case we have performed two analyses of goal (b) with different `d_cmd` values. In the first analysis `d_cmd=20` and in the second one `d_cmd=15`. Figure **??** shows the evolution of dam volume and the position of the spillways in the first counterexample returned by both analyses. We see differences in the evolution, as expected: the commands are sent at different time instants and this causes differences in the evolution of the dam level, although in both cases the commands sent are the same and the spillways are finally in position `open2` to release the desired amount of water.

The last two rows of Table **??** shows some stats of the analysis of goal (b). In that case the optimization in time of the never claim is not possible due to the fact that the goal uses relative time values, not the global time. Note that in **??** we comment that a decrement in the value of `d_cmd` increases the number of states and thus the amount of memory and time needed. However, in this case, neither counterexample are not comparable due to the fact that commands are sent in different time instants and that causes different evolution. We can see that in the case of `d_cmd=20` the trigger value it is reached at time 37, while when `d_cmd=15` this value is reached at time 23, that means that in the first case the depth at which we find the counterexample is greater than in the second one. A deeper analysis means more memory and time needed to perform the analysis. We can conclude that the increment of state space and memory have a close relation with the delay between commands (the grade of non-deterministic behavior) and with the depth at which we find the counterexample.

5. Comparison with Related Work

The idea of using model checking for program synthesis has been developed in parallel with automatic verification since the end of the 80's. This section summarizes related research in this area and compares them with our approach.

Emerson and Clarke [**?**] described a procedure to obtain the synchronization skeleton of concurrent programs from a specification given with branching time temporal logic. The procedure consists of constructing a finite directed graph with AND-OR nodes called a tableau, and then deleting portions of the tableau incrementally in order to satisfy the temporal formula

in the initial node. The resulting graph, if the formula is satisfiable, is the expected skeleton. Manna and Wolper [**?**] apply a similar tableau-like satisfiability algorithm to obtain the synchronization skeleton for communicating processes in the context of Proposition Temporal Logic specifications. Compared to these seminal works, apart from the time-related aspects discussed below, our method differs in the use of on-the-fly model checking, instead of the tableau procedure.

More recently, the same basic notion of translating temporal logic into models have been presented in specific scenarios. In [**?**], Pasareanu et al. use synthesis to support the assume-guarantee verification approach. In the assume-guarantee approach, a system to be verified should guarantee a (temporal) property ψ if the environment that interact with the system satisfies the assumption ϕ. If the model of the environment is available, then a model checker could be uses to check the LTL formula $\phi \rightarrow \psi$. In [**?**], a tableau algorithm allows the automatic translation of the assumption ϕ into the environment, thus removing ψ from the LTL formula to be checked. Our work is also close to [**?**] in the idea of generating the environment that meets an expected behaviour; however there are two main differences. On the one hand, the information that we have for every synthesis problem is the guarantee ψ (our goal), and our problem is to find the assumption ϕ (the expected environment behavior) leading to that goal. The second one is that we use on-the-fly model checking with the current model of the system and a non-restricted environment, instead of tableau methods. In that way, we can consider the hybrid nature of the problem without defining a real-time temporal logic.

Andersen [**?**] exploits the ideas of partial model checking to transform the specification of a concurrent system in order to remove the behaviors that are not necessary to satisfy a property. The system is described as a large composition of processes with process algebra operators, and behaviors are removed by moving parts of the concurrent system into the specification. The result is a method to mitigate the state explosion problem. Like [**?**], we make transformations. They consider large compositions of processes, which are often very similar, like the example taken from Milner. We model the system with a small number of different processes, and the kind of interaction makes it difficult to remove them automatically. We are not interested in removing complexity but in locating the exact actions to be executed at every moment in time.

It is also worth noting the use of synthesis in the specific field of planning and controllers. Maler et al. [**?**] make the same description of the problem that we use in this paper: they consider a generic plan P with any potential behavior and they study how to obtain a controller C that interacts properly with the plant in order to keep its behavior within a given limits (or meeting a specification). They define the problem in terms of acceptance conditions of a non-deterministic automaton that recognizes the interaction produced by P and C. Acceptance states determine evolutions of the plant following the indications of the controller. In our case, plant P is the dam and controller C is the user that operates the spillways, and the acceptance conditions are defined in the *never claim* resulting from the temporal logic formula. However, instead of using the on-the-fly model checking capabilities, the authors of [**?**] propose an iterative process to solve fixed-point equations, first with the discrete case, and as an extension considering the time aspects.

Nowadays, there is a great deal of research on planning and controller synthesis for hybrid systems. Fehnker et al. [**?**] have used UPPAAL [**?**], a model checking tool for hybrid systems based on timed automata, to solve the scheduling problem of a steel plant; it presents

many constraints on time, different steel compounds and limited resources. The scheduling problem consists of generating a controller that allows the fabrication of all the necessary steel compounds in a given time. This problem is translated into a reachability problem and solved with model checking algorithms. The schedule obtained is not time optimal due to the fact that the UPPAAL algorithm was not able to determine if the scheduler exists for several deadlines. In [**?**], Hune et al. use UPPAAL to deal with the problem of scheduling and synthesizing distributed control programs for a batch production plant. They proposed a general method that allows the user to guide the model checker according to heuristically chosen strategies by means of modifications in the system model. This guidance notably reduces the state space; however, it is the user who has to implement reduction strategies in each scenario. In this case the scheduling problem is also reformulated as a reachability problem. In our approach we try to reduce the interaction of the user application with the model, asking the user for a desired behavior, not a strategy. Although we have not yet implemented the mechanism to synthesize an optimal controller, there are some previous works that show a first possible approach to this problem with SPIN [**?**].

Based on [**?**] and using the model checking algorithm of UPPAAL, UPPAAL-TIGA [**?**] was developed to solve control problems modeled as timed game automata. In [**?**], UPPAAL-TIGA and Simulink [**?**] are used as a synthesis tool chain for a climate controller. UPPAAL TIGA automatically synthesizes safe strategies, which are transformed for input to Simulink, which is used to run simulations and generate executable code. More recently, Cassez et al. [**?**] integrate UPPAAL-TIGA in a different tool chain for the automatic synthesis of robust and optimal controllers. There are three different classes of models that integrate the tools UPPAAL-TIGA, for synthesis, PHAVER [**?**], for verification, and Simulink for simulation. This methodology is applied to industrial equipment, namely an oil pump controller. In both works, it is necessary to implement more than one model, using different input languages with different abstraction level: a first simplified and abstract level as input to UPPAAL-TIGA to synthesize the strategy and at least one other model, with more detail, that allows the verification of the strategy. A drawback of this kind of approach is the need for a translation tool from one output language to another input language, as this task is not always done automatically. In contrast with those works, we try to implement with a single model checking tool, a controller synthesis tool that ensures the correctness of the results.

Gromyko et al.[**?**] proposed a framework for the synthesis of controllers for (non-deterministic) discrete event systems with temporal logic specifications. The framework is built on top of symbolic model checking techniques and binary decision diagrams that allow dealing with considerably large problems. As well, the framework is used to simulate and verify a (controlled) plant. In this framework the proposed algorithm takes a non-deterministic plant as input, expresses requirements as CTL formulae and produces a controller as output. In our approach, the desired behavior of the plant (in this case a dam) can be expressed as `LTL` formulae or *never claim*. This last method is more expressive although it is more complex to generate.

Della Penna et al. [**?**] proposed a general methodology that exploits explicit model checking in an innovative way to automatically synthesize a (time-) optimal numerical controller from a plant specification and only apply an optimized strengthening algorithm on the most significant states, in order to reach an acceptable degree of robustness. They implemented

all the algorithms within the $CGMur\varphi$ tool, an extension of the well-known $CMur\varphi$ verifier. Our objective is not to obtain a robust controller that acts according to the different states of the dam over time. We want to develop a controller for each flood scenario based on the weather forecast and the initial state of the dam.

Morgenstern and Schneider [**?**] define the supervisor synthesis problem as the task of restricting the actions of a system in order to satisfy a given specification, and they also argue that model checking can be an effective technique to obtain a solution. However, they do not specifically develop a model checking-base method. Their work is oriented to translating the formulation of this problem from the game-structure context to Kripke structures. In particular they study how to reduce the alternating time μcalculus on game structures to model checking of μcalculus on Kripke structures, allowing many model checkers to solve the supervisor synthesis problem.

Another work that exploits model checking in control systems is the proposal by Lerda et al. [**?**]. They also use model checking to locate counterexamples, however their objective is the opposite of our work. They integrate the model checker Java Path Finder [**?**] with the numerical simulation capabilities of MATLAB/Simulink to detect possible design errors in the controller.

6. Conclusions

In this paper we have presented a methodology that performs the automatic synthesis of controllers based on model checking. We have shown how on-the-fly model checking can be used to synthesize controllers. It is worth noting that this approach guarantees the correctness of the results. If the analysis does not return a solution, we verify that a suitable solution does not exist. In addition, our approach offers mechanisms to impose additional constraints to guide the search, such as depth reduction or limiting the time step used in the time model. We satisfactorily have applied this methodology using SPIN to a real case study: the synthesis of controllers for dam management. Since the dam has been modeled as a hybrid system, we have modified the model to allow its analysis with SPIN, which was not originally designed to analyze hybrid systems.

Traditional management systems for water resources are mainly based on the simulation of complex hydrological models which constitute a suitable management mechanism for systems with continuous evolution. However, dams behave as hybrid systems with a discrete component that is not properly handled by hydrological models. Traditional dam management systems are usually oriented to long term management, with long time step periods. Finding a safe strategy can entail a huge number of simulations and high computational time. Our approach only focuses on dam management, and does not consider in detail hydrologic models upstream and downstream, but it can significantly reduce the analysis time and computational costs due to the use of the exhaustive search of abstract model's state space.

The performance of the analysis obtained with our approach can have different interpretations depending on the application domain. In our case study the results are promising, in the sense that most dam management tools need more computational resources to work and more time to make decisions. These results show that our approach is able to

manage the dam in flood scenarios.Scalability may be discussed from two points of view. On the one hand, the model of the dam may be implemented adding more complexity, with more spillways and elements. In this case, the size of the state space would increase, and we could have problems with the size of the state space. Part of a future work is focused on using parallel model checking to alleviate this problem. On the other hand, another problem with scalability is concerned with how time is dealt with in the model. Currently, we can synthesize solutions than can be carried out during, at most, six hours. This time interval is enough for the type of system we are analyzing. However, in order to extend this time interval, we should reimplement the model under analysis.

Future work will focus on two main tasks: the improvement of the framework/methodology and the development of a tool that is functional and user-friendly. In the first task, it is necessary to improve the performance of analysis to manage more complex systems. To this end we would like to use abstraction to improve the time model by reducing the state space. A second point is the use of a different logic that can express time, in order to automatically generate the optimized never claims presented in Section **??**. In the second task, we have to construct a graphical interface that avoids the manual description of goal automata and the dam model in PROMELA. From our previous applications of SPIN, such as [**?**, **?**, **?**], we concluded that user friendly GUIs are considered desirable by formal method users. Finally, we propose the integration of this functional tool with sensor systems or traditional DSS to obtain a stand-alone controller that can be automatically adapted to different flood scenarios.

REFERENCES

1. Taghirad H, Bakhshi G. Composite-h ∞ controller synthesis for flexible joint robots. *Intelligent Robots and System, 2002. IEEE/RSJ International Conference on*, vol. 3, Saint Louis, USA, 2002; 2067–2072 vol.3.
2. Feng G, Lu G, Zhou S. An approach to h controller synthesis of piecewise linear systems. *Communications in Information and Systems*, vol. 2, 2002; 245–254.
3. Rodonyi G, Gaspar P, Bokor J. Robust lpv controller synthesis with uncertainty modeling. *Control and Automation, 2009. MED '09. 17th Mediterranean Conference on*, Thessaloniki, Greece, 2009; 815–820.
4. Fehnker A. Scheduling a steel plant with timed automata. *RTCSA '99: Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications*, IEEE Computer Society: Washington, DC, USA, 1999; 280.
5. Gomes CP, Smith D, Westfold S. Synthesis of schedulers for planned shutdowns of power plants. *KBSE '96: Proceedings of The 11th Knowledge-Based Software Engineering Conference*, IEEE Computer Society: Washington, DC, USA, 1996; 12.
6. Cassez F, David A, Fleury E, Larsen KG, Lime D. Efficient on-the-fly algorithms for the analysis of timed games. *CONCUR 2005 - Concurrency Theory*, Springer-Verlag: London, UK, 2005; 66–80.
7. Bhansali S, Hoar TJ. Automated software synthesis: An application in mechanical cad. *IEEE Trans. Software Eng.* 1998; **24**(10):848–862.
8. Håkansson J, Mokrushin L, Pettersson P, Yi W. An analysis tool for uml models with spt annotations. Presented at SVERTS2004, http://www-verimag.imag.fr/EVENTS/2004/SVERTS.
9. Tronci E. Optimal finite state supervisory control. *Proceedings of 35th IEEE Conference on Decision and Control*, vol. 12, Kobe, Japan, 1996; 2237–2242.
10. Clarke EM, Emerson EA, Sistla AP. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.* 1986; **8**(2):244–263.
11. Queille JP, Sifakis J. Specification and verification of concurrent systems in cesar. *Proceedings of the 5th Colloquium on International Symposium on Programming*, Springer-Verlag: London, UK, 1982; 337–351.
12. Holzmann GJ. The model checker spin. *IEEE Transactions on Software Engineering* 1997; **23**(5):279–295.

13. Giupponi C, Mysiak J, Fassio A, Cogan V. Mulino-dss: a computer tool for sustainable use of water resources at the catchment scale. *Math. Comput. Simul.* 2004; **64**(1):13–24.
14. Mysiak J, Giupponi C, Rosato P. Towards the development of a decision support system for water resource management. *Environmental Modelling & Software* 2005; **20**(2):203 – 214. Policies and Tools for Sustainable Water Management in the European Union.
15. Koutsoyiannis D, Karavokiros G, Efstratiadis A, Mamassis N, Koukouvinos A, Christofides A. A decision support system for the management of the water resource system of athens. *Physics and Chemistry of the Earth, Parts A/B/C* 2003; **28**(14-15):599 – 609.
16. Hasebe M, Nagayama Y. Reservoir operation using the neural network and fuzzy systems for dam control and operation support. *Adv. Eng. Softw.* 2002; **33**(5):245–260.
17. Karaboga D, Bagis A, Haktanir T. Controlling spillway gates of dams by using fuzzy logic controller with optimum rule number. *Appl. Soft Comput.* 2008; **8**(1):232–238.
18. Holzmann GJ. *The SPIN Model Checker : Primer and Reference Manual*. Addison-Wesley Professional, 2003.
19. Bosnacki D, Dams D. Discrete-time PROMELA and SPIN. *FTRTFT '98: Proceedings of the 5th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, Springer-Verlag, 1998; 307–310.
20. Holzmann GJ, Joshi R. Model-driven software verification. *SPIN*, 2004; 76–91.
21. Holzmann GJ, Joshi R, Groce A. Model driven code checking. *Autom. Softw. Eng.* 2008; **15**(3-4):283–297.
22. Clarke EM, Emerson EA. Design and synthesis of synchronization skeletons using branching time temporal logic 2008; :196–215.
23. Manna Z, Wolper P. Synthesis of communicating processes from temporal logic specifications. *ACM Trans. Program. Lang. Syst.* 1984; **6**(1):68–93.
24. Pasareanu CS, Dwyer MB, Huth M. Assume-guarantee model checking of software: A comparative case study. *SPIN*, London, UK, 1999; 168–183.
25. Andersen HR. Partial model checking. *LICS '95: Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science*, IEEE Computer Society: Washington, DC, USA, 1995; 398.
26. Maler O, Pnueli A, Sifakis J. On the synthesis of discrete controllers for timed systems (an extended abstract). *STACS*, Munich, Germany, 1995; 229–242.
27. Behrmann G, David A, Larsen K. A Tutorial on UPPAAL. *Formal Methods for the Design of Real-Time Systems*, no. 3185 in LNCS, Springer–Verlag, 2004; 200–237.
28. Hune TS, Larsen KG, Pettersson P. Guided synthesis of control programs using UPPAAL. *Nordic Journal of Computing* 2001; **8**(1):43–64.
29. Ruys TC. Optimal scheduling using branch and bound with spin 4.0. *SPIN*, Portlan, OR, USA, 2003; 1–17.
30. Behrmann G, Cougnard A, David A, Fleury E, Larsen KG, Lime D. Uppaal-tiga: Time for playing games! *CAV*, Berlin, Germany, 2007; 121–125.
31. Jessen JJ, Rasmussen JI, Larsen KG, David A. Guided controller synthesis for climate controller using UPPAAL-TIGA. *Proceedings of the 19th International Conference on Formal Modeling and Analysis of Timed Systems*, no. 4763 in LNCS, Springer, 2007; 227–240.
32. Dabney JB, Harman TL. *Mastering Simulink*. Pearson: Upper Saddle River, NJ, 2004.
33. Cassez F, Jessen JJ, Larsen KG, Raskin JF, Reynier PA. Automatic synthesis of robust and optimal controllers — an industrial case study. *HSCC '09: Proceedings of the 12th International Conference on Hybrid Systems: Computation and Control*, Springer-Verlag: Berlin, Heidelberg, 2009; 90–104.
34. Frehse G. Phaver: algorithmic verification of hybrid systems past hytech. *Int. J. Softw. Tools Technol. Transf.* 2008; **10**(3):263–279.
35. Gromyko A, Pistore M, Traverso P. A tool for controller synthesis via symbolic model checking. *Discrete Event Systems, 2006 8th International Workshop on*, IEEE Press: Ann Arbor, USA, 2006; 475–476.
36. Penna GD, Magazzeni D, Tofani A, Intrigila B, Melatti I, Tronci E. Automatic synthesis of robust numerical controllers. *ICAS '07: Proceedings of the Third International Conference on Autonomic and Autonomous Systems*, IEEE Computer Society: Washington, DC, USA, 2007; 4.
37. Morgenstern A, Schneider K. Using model checking to solve supervisor synthesis problems. *Decision and Control, 2005 and 2005 European Control Conference. CDC-ECC '05. 44th IEEE Conference on*, Seville, Spain, 2005; 2757–2762.
38. Lerda F, Kapinski J, Maka H, Clarke E, Krogh B. Model checking in-the-loop: Finding counterexamples by systematic simulation. *American Control Conference*, Seattle, USA, 2008; 2734–2740.
39. Havelund K, Pressburger T. Model checking java programs using java pathfinder. *STTT* 2000; **2**(4):366–381.

40. Gallardo MM, Martínez J, Merino P, Pimentel E. αspin: A tool for abstract model checking. *Int. J. Softw. Tools Technol. Transf.* 2004; **5**(2):165–184.
41. de la Cámara P, Gallardo MM, Merino P. Model extraction for arinc 653 based avionics software. *SPIN*, Berlin, Germany, 2007; 243–262.
42. Gallardo MM, Merino P, Sanán D. Model checking dynamic memory allocation in operating systems. *Journal of Automated Reasoning* 2009; **42**(2):229–264.