Modelling and Analysis of a Steam Generator using UPPAAL

*Kåre J. Kristoffersen** *Paul Petterson* †

Abstract

We present the modelling and analysis of a steam generator using the automatic verification tool UPPAAL. The physical parts of the steam generator as well as the control software are modelled in the language of timed automata. Requirements to the system are stated in a real–time logic allowing for expessing safety properties such as 'the waterlevel in the drum must never go below nor above certain critical values'. Also the logic allows for expressing bounded liveness properties, e.g. 'steam must be produced within a certain timelimit'. The automatic modelchecker in UPPAAL is used to investigate system behaviour with respect to logical properties.

Introduction

In a steam generator water is let into a drum where it gets heated by a burner with the aim of producing steam to be used for example in the production of electric power. The drum and the burner are controlled and monitored by two control programs, whose job is to ensure thet fresh water is let into the drum when needed and to turn the heat on and off when needed. We like steam generators to enjoy certain properties regarding *safety*. For example, the waterlevel in the drum should always be kept in a particular interval to avoid meltdown or explosions. In addition to this we generally expect *liveness* from a steam generator namely that some steam is produced within a fixed time frame [KAS95, Abr95].

To assure ourselves that such properties (and many more) hold we need to build some kind of true model of the system and then *analyze* this model. Here we choose the language of timed automata, or more precisely parallel compositions of such. But why use timed automata? Why use a language with time? Well as it turns out, in a steam generator actions really do take some time. It takes time to let fresh water into the drum, it takes time to heat water to its boiling point and it takes some time to transform a certain amount of water to steam. Also being able to put components in parallel (with the intention of concurrent execution) enables us to nicely capture the distributed nature of a steam generator.

*Dept. of Math. & Comp. Sc., Aalborg University. email: jelling@iesd.auc.dk

†Dept. of Comp. Systems, Uppsala University, Sweden. email: paupet@docs.uu.se

Correctness questions now reduces to to asking wheater some desirable/undesirable states of the system can be *reached*. This task can in principle be carried out using paper and pencil, but this way of doing it will generally take far too long time. Fortunately we are lucky to have automatic tools to carry out this sort of work for us. UPPAAL which is a joint effort between department of Computer Systems, Uppsala University, Sweden and Department of Computer Science, Aalborg University, Denmark is a computerized tool designed to carry out precisely this kind of job. In a graphical interface the user can draw composite systems of timed automata, and by using a verification engine *reachability analysis* can be performed, hereby checking a system for safety and bounded liveness.

This paper documents the modelling and analysis of a steam generator using UPPAAL. The work was carried out in october 1995 when the second author was visiting Aalborg University. This experiment is one in a number of experiments carried out to investigate what kind of systems and what kind of behaviour the formalisms of UPPAAL can capture. In addition to this a major aim has been to test the performance of UPPAAL with respect to speed. [YPD95, YLP95].

Steam Generator

Roughly speaking a steam generator consists of *four* components, namely a *drum* containing some water, underneath the drum a *burner* that heats the water in the drum and additionally two control programs, one for each of the drum and the burner. Let us therefore think of a steam generator as the diagram in Figure 1 below.

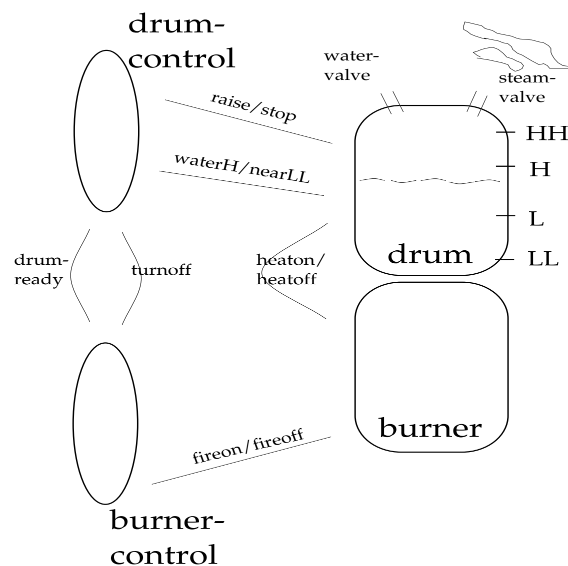

Figure 1: Flowgraph of a steam generator

On the right hand side we have the drum and the burner, which are the physical

units. On the left we have the control software consisting of a controller for the drum and one for the burner. The components are connected by communication wires (links) to be used in synchronization.

The job of the drumcontroller is to monitor the level of water (by reading a sensor) in the drum and prevent the waterlevel to reach LL (very low) causing drum meltdown and HH (very high) causing explosions. The drumcontroller can achieve this goal by regulating the inlet of water in the drum by sending one of the two signals **raise** and **stop**. **raise** means that the valve for waterinlet should be opened and **stop** means that the water–valve should be closed again.

When the drumcontroller reads **waterH** from the sensor in the drum the water-level is appropiate to begin steamproduction and it sends the signal **drumready** to the burnercontroller which now takes action. It turns the fire on in the burner sending the signal **fireon**. This again causes the burner to send the signal **heaton** to tell the drum that heat is on. Now after a while steamproduction begins and after some period of steamproduction the waterlevel has decreased and this is observed by the drumcontroller by reading **nearLL** from the sensor in the drum. The drum controller then tells the burnerconrol to **turnoff** heating the water until some more water has been let into the drum. This is more or less the cyclic behaviour of a steam generator.

Safety

In our case study we have focussed on two safety properties, both concerning the waterlevel in the drum. We have the following two requirements:

Requirement 1 The waterlevel must never be HH

Requirement 2 The waterlevel must never be LL after startup

In the litterature people have considered one additional safety requirement regarding the behaviour of the burner. In a real burner the burner chamber is ventilated for a small period each time the fire is to be turned on. This will remove potential explosive gasses, thus avoiding a destroying explosion in the burnerchamber when fire is put on. Our reason for not treating this requirement is that we have a very simplified modelling of a real burner, that is, we have simply omitted the ventilation phase in the burner, and therefore we can of course not check that it appears.

Liveness

We consider two liveness requirements that both have to do with the production of steam.

Requirement 3 The water must boil at some point in time

Requirement 4 Steam must be produced

Timed Automata

The way to describe systems in UPPAAL is as parallel compositions of timed automata. For the most detailed description of timed automata we refer to [YLP95]. Here we give only the informal description of the formalism.

A single automaton

A timed automaton is a finite state automaton extended with a set of real–valued clocks. As for an ordinary finite state automaton a timed automaton consists of a finite set of control nodes and some transitions (directed archs) connecting these nodes. A transition is labelled with three things: a *guard*, an *action* and a *reset–set*. The guard is an expression over clocks saying *when* the transition is enabled, the action label is just an ordinary action label and the reset–set is a set of clocks to be reset when the transition is taken. Initially all clocks have the value 0. At each point in time one of two things will happen, either time will progress causing all clocks to grow uniformly or a transition will be taken. To illustrate how this works in practice consider the burnercontrol in Figure 2, which is in fact the burnercontrol in the modelling of the steam generator. Notice that this process is a drawing from the graphical userinterface of UPPAAL. The control node labelled **bc1** is the initial node,

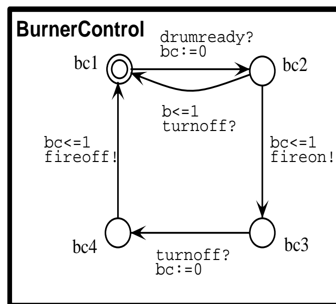

Figure 2: A process to control the burner modelled as a timed automata.

and it has an *unguarded* transition labelled **drumready**to state **bc2**. The unguardedness simply means that the transition can happen at any point in time. When the transition is performed the clock bc is reset (set to 0). In UPPAAL action–transitions happen pairwise by two components performing a handshake. Therefor the **drumready**? transition cannot happen alone, it needs a complementary **drumready**! to be taken by another component. In this situation this transition is performed by the drumcontroller, which we cannot see here of course. However, the burnercontrol will stay in node **bc2** at most 1 time unit. Here it is willing to respond to a **turnoff** signal returning to the initial state and to send the signal **fireon** (to the burner).

Parallel Compositions

Timed automata can be put in parallel and the semantics is as follows. Initially all clocks in the system have the value 0. At each point in time either time will progress causing all clocks to grow uniformly or two processes will perform a handshake.

To illustrate parallel compositions we can consider the burnercontrol and the burner in Figure 3.

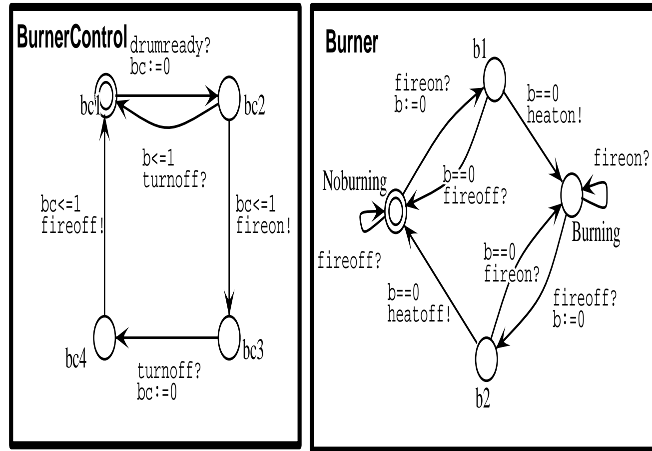

Figure 3: The burner and its controlprogram

For the moment assume that the burnercontrol is in state **bc2** (this is accomplished by synchronizing with the drumcontrol, but we do not bother about this now), so the vaule of clock bc is 0. If the burnercontrol does not get the **turnoff** signal it will synchronize with the burner on the **fireon** action causing the controller to reach state **bc3** and the burner to reach **b1**. The burner is now very eager to tell the drum–process that it is burning using the **heaton** action.

Timed Logic

The requirement language currently implemented in UPPAAL is a very simple language for expressing reachability properies. One can write that a system state (a state vector with one state from each component) can be reached or can never be reached. UPPAAL is then able to check whether this actually holds.

To be precise, questions concerning reachability of certain states can be extended with propositions on clocks. For example we are able to ask 'Is it possible for process p to reach state p3 when clock x is less than or equal to 5?' by asking $\exists\Diamond(\text{p.p3} \wedge x \leq 5)$.

Implementation

Things are now set for a presentation of the actual steam generator implementation. As a matter of fact we have already seen and discussed half of the system, namely the burner and its control program. Let us therefore restrict ourselves to have a look at the drum part and its controlprogram. See Figure 4.

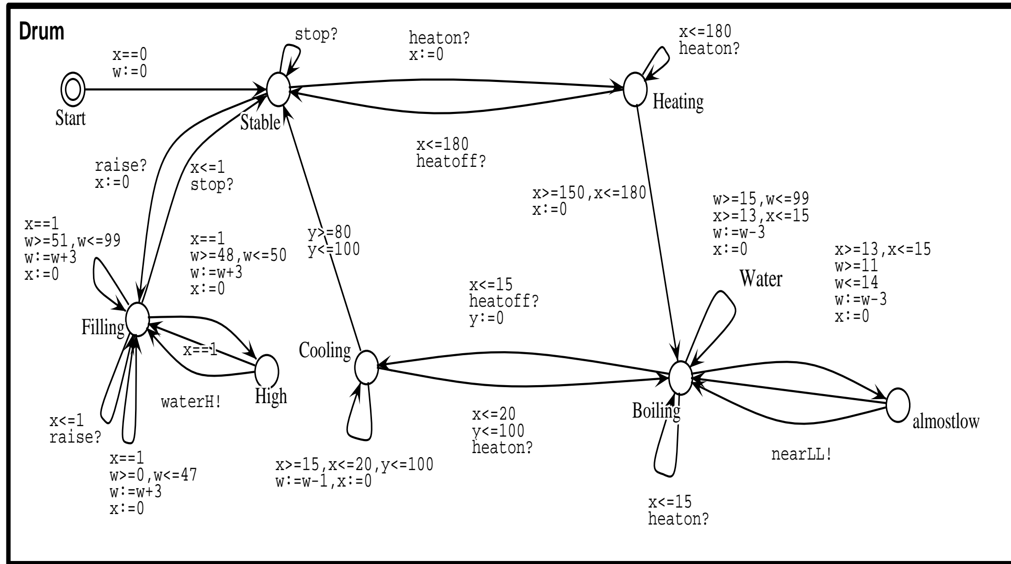

Figure 4: The drum.

In the drum process we have intended to capture the various phases the water goes through when it is in the drum. The drum can be **Stable** meaning that no fresh water is being put into the drum for the moment, and the water is not being heated either. In the **Filling** state the waterlevel raises, in the **Heating** state the temperature grows and oin the **Boiling** state the water is boiling. The **Cooling** state is when heat is taken off and the water is still boiling a bit.

The waterlevel itself is incorporated as the integer w, which is initially 0. Initially the controller sends the **raise** signal to the drum whereby it reaches the **Filling** state. Now w is incremented by 3 for every timeunit (this is taken care of by the guard x==1). When w reaches a value between 48 and 50 the signal **waterH** is sent to the controller which then sends the **stop** signal to prevent inletting more water in the drum.

Next the drum receives the **heaton** signal from the burner process hereby reaching the **Heating** state. After a while here it reaches the **Boiling** state where w is now decreased, modelling that the waterlevel decrease when the water is boiling. Notice that w now decreases much slower than it was increasing in the **Filling** state. When w=13 the drum sends the signal **nearLL** to the drumcontroller, which then tells the burnercontrol to take care of not heat the water for a while.

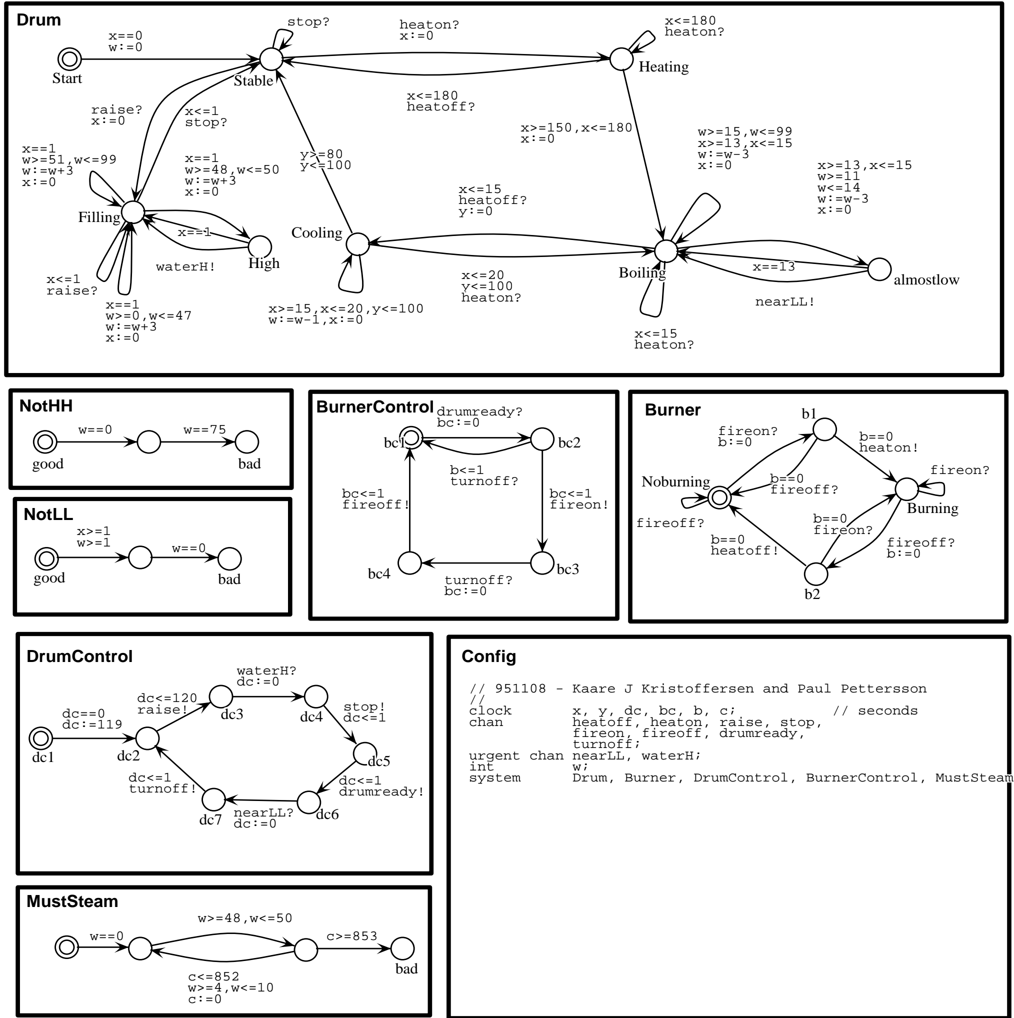

Figure 5: The steam generator.

Verification

To verify the requirement saying that the waterlevel must never be HH we ask a question about the automaton **NotHH**, see Figure 5. We ask UPPAAL whether this automaton can reach the node named **bad**, and notice that this can only happen if w at some point in time is assigned the value 75 which we here shall interpret as very high. So we ask:

$$\forall\Box\neg(\text{NotHH.bad})$$

To verify that the waterlevel not reaches 0 after startup we verify the property $\forall\Box\neg(\text{NotLL.bad})$ in a similar fashion.

To prove that the water can indeed be boiling we prove

$$\exists\Diamond(\text{Drum.Boiling})$$

To prove that steam is repeatedly produced within an interval of time we prove

$$\forall\Box\neg(\text{MustSteam.bad})$$

Conclusion

It is possible to make really good use of UPPAAL when trying to model and verify a steamgenerator.

References

[Abr95] J. R. Abrial. Steam-boiler control specification problem. *Dagstuhl Meeting, Methods for Semantics and Specification*, 1995.

[KAS95] C. H. Kristensen, J. H. Andersen, and A. Skou. Specification and Automatad Verification of Real–Time Behaviour – a Case Study. 1995.

[YLP95] W. Yi, K. G. Larsen, and P. Petterson. Compositional and Symbolic Model Checking of Real–Time Systems. In *Proceedings of RTSS95*, 1995.

[YPD95] W. Yi, P. Petterson, and M. Daniels. Constraint solving... In *lkasjdfkj*, 1995.