Steam Boiler System

Table of Contents

UPPAAL Model 1

UPPAAL Specification 4

Security of Boiler Control and Model-Checking 5

(A) 6

(B) 6

Reliability Of Boiler Control and Model-Checking 7

(A) 7

(B) 7

Critical Evaluation 8

References & Citation: 9

# UPPAAL Model

Uppaal makes use of timed automatons, which can be defined as finite-state machines that extend individual clock like variables. These clock variables are used to evaluate real integers in a dense-time modeling scheme. All of which progress in a synchronous fashion, networked together to create a model. This concurrent system can then be fashioned into a state-transition graph model, where the systems behaviors are compared to temporal logic formulas to analyze the real time system for correctness in both syntax and timing.

The boiler model that was created for this project makes use of several timed automata in parallel; to extend discrete variables, all of which are set to predetermined arithmetic operations that where provided in our assignment briefing. Once clear comprehension of the system and its requirements of the boiler project where made, the development of the template automata where determined and defined. To do this the system requirement where evaluated and processed using the formal description technique, detailing each individual function and probable/possible implementations predefined in the system briefing. The physical entities where broken down into individual FSM’s that remained synchronous to the overall project. Each containing a set of parameters of multiple types, these parameters are substitutions for arguments that are made in the process declarations. These physical entities are as listed: Boiler system and Pump system.

The briefing dictates that water is allowed into the boiler using the pumping system, where it is heated and evaporated to produce steam. The boiler is considered unsafe to operate once the water level in not maintained between a specified minimum (W1) and maximum (W2) levels (the current water level (w)). Because the project required the use of bounded integer variables, where the min and max are used as the lower/upper bound, respectively as guards; as well as being used to check upon verification where violating an upper/lower bound triggers an invalid state. A sensor will be used to monitor these levels, the formula for which is simplified to (W1 <= w <= W2) = SAFE.

The Boiler automation for this project is in parallel with both the systems Sensor and Controller models, it’s responsible for detecting events without changing the observed system. It’s a simple automation that contains three nodes, an Idle start node that traverses along the edge when an emergencyStop? is synchronized to the EMERGENCY\_STOP node that stops the entire system. This is ultimately triggered if a sensor detects a variable is out\_of\_bounds, preventing further damage to the physical system in the case of an emergency. The third node is parallel to the Controller and is used as a manual emergency stop; therefore if pressed by the user (emergencyStopButton?) the node will traverse along the edge to the EMERGENY\_STOP node and stop the entire system.

The Sensor automation is responsible for constantly polling the system to determine if the bounded integer variables for both the water and steam levels. If it is detected that either of these levels are not within the predefined set limits then the error node will traverse through the emergencyStop! channels, activating the trigger to STOP the Boiler. The steam-rate is maintained in a similar fashion to the water-level, whereas the steam-rate must also not exceed a specified maximum rate (S). The current steam-rate is defined by the variable ‘s’ (lowercase), therefore the formula used to check the bounds by the Controller is (s<S).

The Controller automation is a digital system responsible for a variety of important tasks; it contains the primary safety properties, which are set to prevent any system faults. This is done by setting operations invariantly under thresholds as a means of fault prevention. Another task constantly checked by the Controller is to determine whether the sensors are responsive. The water-level and steam-flow sensor are polled and each is expected to reply with its reading within 0.5-second; if no reply is received, another poll is issued. Polling repeats at 0.5-second intervals until a reply is received, if however 5 polls are unanswered the system is deemed faulty and shuts down. When this occurs the channel will issue an emergencyStop! that leads to a SHUT\_DOWN node parallel to the Boilers. The Controller automation also contains an option that allows for the Boiler to be shut down manually if an error occurs. It is imperative that the message transmission system, used to poll the boilers is always prioritized. For this purpose the urgent synchronization channels between the system polling are declared by prefixing the channel declaration with the keyword urgent. This can be seen in the Sensor automation as a means to deter any possibility of delays occurring while a synchronization transition on the channel pollWater? is enabled.

The Pump automation is also a physical entity in this system, it allows for the alternation between pumps (p1 and p2) as well as the supply of water to flow into the boiler at a constant rate. The setup allows for no more than one pump to be active at any given moment; this is so that the secondary pump can be used in emergencies, while maintenance is preformed in the event that failure occurs without turning off the entire boiler. The status of each pump is continually polled for failure, even after failure has occurred and the pump has been declared as broken. The setup of the Pump automation starts on an idle state node that is then synchronized with the choosePump? channel, which has split probability of selecting either p1 or p2. Once a pump has been selected by the Pump, the channel/edge is traversed, providing annotated selections, guards, and synchronizations. Providing water to the boiler (w++), detecting if the pump has become broken (!pump\_Broken) and releasing the pump (releasePump2?). If the pump breaks the pump state changes so pump\_Broken == 1, which causes it to travel back to the unguarded-self loop Idle and to stop both the water flow (w--) and pump (stopPump?).

Alternatively when I began designing this concurrent system I spent an awful lot of time with the Sensor automation, reconfiguring the invariants and changing around the guards to figure out why deadlock was constantly occurring. In my original mock-up the polling of the sensors was on an unguarded self-loop that would increase the poll (poll++) each time it went unanswered. Sadly every 0.25 seconds the system would deadlock because it would fail to poll the system properly due to the fact that the guards set along the poll associated edge where set to y = 0, rather than actually polling an parallel system to determine if the poll sensors where still in-bounds.

# UPPAAL Specification

The model-checker is used to verify the model with respect to the requirement specifications. In the model, the requirement specifications are expressed formally in a well-defined (machine readable) language. These declarations of the system properties can either be global or local, containing the declarations of blocks, bounded integers, channels, arrays, records or types. Below is a table depicting the individual urgent channels, channels, integer variables and Boolean variables in each automation:

|  |  |  |  |
| --- | --- | --- | --- |
| Controller() | | | |
| Urgent chan: | Channels: | Int variables: | Bool variables: |
| emergencyStop! | pollSensors! | polls |  |
| emergencyStopButton! | w |
| pollWater! | W1 |
| waterOK? | W2 |
| ChoosePump! | s |
| releasePump1! | sMAX |
| releasePump2! |
| stopPump! |

|  |  |  |  |
| --- | --- | --- | --- |
| Sensor() | | | |
| Urgent chan: | Channels: | Int variables: | Bool variables: |
| emergencyStop! | pollSensors? | w |  |
| pollWater? | W1 |
| W2 |
| s |
| sMAX |

|  |  |  |  |
| --- | --- | --- | --- |
| Boiler() | | | |
| Urgent chan: | Channels: | Int variables: | Bool variables: |
| emergencyStop? | emergencyStopButton? |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Pump() |  |  |  |
| Urgent chan: | Channels: | Int variables: | Bool variables: |
| choosePump? | w | pump1Broken |
| releasePump1? | Pump2Broken |
| releasePump2? |
| stopPump? |

The verifier allows you to select individual properties in the Overview list to add/remove to the model-checker. Once properties have been added to the list it’s possible to edit the definition and leave comments on the documentation of the program. When editing the property equivalences its possible to check the possibly, invariantly, potentially always, eventually and leads to properties of an individual specification property. Manipulating these equivalences allows us to determine if each property has simple reachability, safety and live-ness properties but most importantly allow us to check for the absence of deadlock.

# Security of Boiler Control and Model-Checking

If an intruder is able to fool the boiler controller into believing that a pump is working correctly when, in fact, it has failed. The redundancies to this happening is that the entire system would shutoff given the level of water will proceed to drop/rise below/above respectfully, the given bounds. My assumption is a hacker in this scenario has set the bool value of pump1Broken == 0 to trick the system into believing the pump is working when it has in fact failed. However given the fact that the PUMP\_1 cycles in a self-Loop, constantly checking for updates of the pump broken status and changes in the state of the system from the Controller automation until the pump is finished. This means the hacker would have to constantly alter the state of the pump and still keep the boiler within the set bounds without causing the emergency shutoff to be triggered.

# (A)

Bellare and Rogaway created the formalization of cryptographic protocols, which are used as a formal model to check against threats to a system from hackers. The variants of the BR cryptographic protocol can be modified and implemented in the specification of almost any program. The secure protocol provided to us in the assignment briefings are as follows:

B 🡪 P: {NB, B}KP

P 🡪 B: {NB, NP}KB

B 🡪 P: {NP}KP

In the protocol above mutual authentication between the two automations Controller (B) and Sensor (P) is achieved. In this protocol a message is transposed between the Controller and Sensor and authentication (via-shared keys) is then achieved through the use of encryption (keys and nounce). The authentication protocol sends a message to the B or P, to accept the message; the previously sent message must contain a nonce that validates the recent usage of the encryption. The use of this protocol should prevent the use of multi-protocol attacks on the boiler system.

# (B)

The protocol can be rewritten for the communications between the boilers Controller and Sensor automations below:

Controller 🡪 Sensor : {NounceController, Controller)Key Sensor

Sensor 🡪 Controller : {NounceController, NounceSensor}Key Controller

Controller 🡪 Sensor : {NounceSensor}Key Sensor

* Authentication of NounceController, Controller (B) and Sensor (P). Nodes B and P share the same value for NounceController, Controller (B) and Sensor (P) both execute the same session of the protocol.
* Confidentiality of Sensor (P), a secure message is shared between P and B.

|  |  |  |
| --- | --- | --- |
| Controller | 🡪 | Sensor  Certificate |
| Sensor key received  Change nounce | 🡨 | Controller key received |
| Controller Message | 🡪 | Change nounce  Finished |
| Controller(ver\_max, cr, rsid, nounce, comp\_methods) | | |
| Sensor(version, sr, sid, nounce, comp\_method) | | |

# Reliability Of Boiler Control and Model-Checking

Model-checking algorithms supply elegant solution to many distributed coordination problems and other real-life systems that are inherently stochastic in nature. This is because they have the ability to include components that are known to be unreliable. The boilers Controller model is the centerpiece of the system, being responsible for polling the sensor data, starting/stopping pumps, and shutting the system in the case of an emergency. This is why it’s important to check the reliability of this system through the model-checker.

# (A)

Assuming that a reply to a request for a sensor value has the probability p of being delayed, for a range of values p. Determine how many retries are required on average, and how long the system can be expected to run before a timeout fault occurs and the system is shutdown. After the initial poll replies as a fault it passes through a urgent synchronization channel between the sensors and error nodes, the use of this urgent channel is to prevent any possibility’s of delays occurring while a synchronization transition on the SENSORS\_CHECKED edge. The following edge contains the guard: x >= 5 (polls++), meaning that the channel is looped through every .5-seconds until a reply is received. However if 4 additional polls are unanswered (.25-second) a timeout error occurs and the system is deemed faulty triggering an error. Allowing for more retries is problematic due to the systems briefing guidelines. My PRISM model of the process simulating the programs sensor is as follows:

*dtmc //the protocol is synchronous with no non-determinism*

*module Sensor //module for sensor*

*s : [0..5] init 0; //state*

*[success] s=0 -> 0.8 : (s’=0) + 0.2 : (s’=1);*

*[fail] s=1 -> 0.8 : (s’=0) + 0.2 : (s’=2);*

*[fail] s=2 -> 0.8 : (s’=0) + 0.2 : (s’=3);*

*[fail] s=3 -> 0.8 : (s’=0) + 0.2 : (s’=4);*

*[fail] s=4 -> 0.8 : (s’=0) + 0.2 : (s’=5);*

*[fail] s=5 -> (s’=5);*

*endmodule*

*rewards “PumpSuccessful”*

*[success] true : 1;*

*endrewards*

*rewards “PumpFail”*

*[fail] true : 1;*

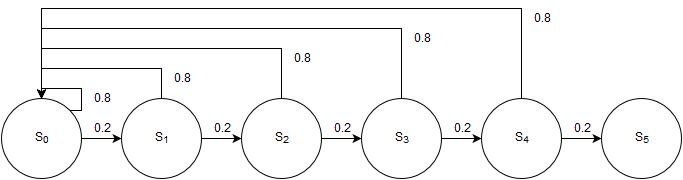
*endrewards*

*rewards “Poll”*

*true : 1;*

*endrewards*

Below is a simulation of the sensor model created in Promela:



Discrete-time Markov chain:

P = {S0, S1, S2, S3, S4, S5}

Pini = S0

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| P{ | 0.8 | 0.2 | 0 | 0 | 0 | 0 |
| 0.8 | 0 | 0.2 | 0 | 0 | 0 |
| 0.8 | 0 | 0 | 0.2 | 0 | 0 |
| 0.8 | 0 | 0 | 0 | 0.2 | 0 |
| 0.8 | 0 | 0 | 0 | 0 | 0.2 |
| 0 | 0 | 0 | 0 | 0 | 0 |

# (B)

Discrete-time Markov chains are a stochastic process, meant to model the evolution over time of a process where randomness is inherent. In this project the use of discrete-time Markov chains is used in the data being received by the sensor 🡪 controller. It denotes the probability of state changes through a step-by-step process, using a transition probability to move through the controller. It’s also used in a less complex way in the Pump automation, where the probability of selecting either p1 or p2 is split by the following formula:

(Where the probability of the next state is entirely dependent on the previous event.)

# Critical Evaluation

PROMELA (Protocol Meta Language) / SPIN (Simple PROMELA Interpreter) are mainly protocol validation model languages. The validation of the model however does not need to describe the critical details of implementation; rather, it focuses on the structure of a model. Based on this model, the checker is able to use the theory of finite state machines to reduce the required amount of language elements and simplify the overall development of our model. An advantage to this is it allows for the description of properties in a process ‘proctype’ and global resources to be used in-between processes as a means of communication.

Uppaal is also an integrated tool, developed with the purpose of modeling and verifying real-time embedded systems. The typical usage of Uppaal deals in areas of real-time controllers and communication protocols, where the aspect of timing individual components becomes crucial in the overall functionality of a model. The model-checker in Uppaal is used to determine the invariants and reachability of properties by tracing the state-space of a given system. This diagnostic trace can provide explanations to why a property is/isn’t satisfied by a system as well as being used as a graphical visualization of the movement through the real-time system.

Many embedded systems are continually reacting to real-time changes in the system’s environment, and must be able to compute certain arithmetic’s in real-time without causing delay. In this program the boilers Controller continually polls the sensors, where the probability of the next state is entirely dependent on the previous senor data. It must compute whether the sensors data in inbounds to both the water and steam limits every .5 second repeatedly; when a delayed computation result occurs (5 times) or a out\_of\_bounds limit is reached the system reports a failure, thus begins maintenance of the system before going into a emergency shutoff. These real-time implementations fulfill the systems functionality simultaneously; therefore the model checker needs to be preformed at each stage to establish optimal design metrics.

The advantage of using Uppaal for this project is it allows us to use a graphical user interface to run a simulator. This simulation can be run in three ways: the system can be run manually, where the transitions are chosen. A random mode can be toggled to let the system run in a ‘free’ mode. Lastly the user can go through a trace to see how certain individual states can be used and reached throughout the automations. When the trace is generated the model-checker finds what properties have been satisfied/unsatisfied. This allows the user to make interpretations based on the constraints of the system, showing all possible combinations of clock variations that can be reached along a path.

Promella is good at analyzing models with the SPIN model-checker, when it comes to implementing a representation of the program through imposing natural restrictions as a method of generating real-time interaction within the system. However it is deficient when it comes to preforming verification under strong constraints and boundaries, due to it being unable to develop solutions and development models that run concurrently with out causing the system to enter deadlock.

# References & Citation:

* 28, 2006 Updated November. "A Tutorial on Uppaal 4.0." *A Tutorial on Uppaal 4.0* (2006): n. pag. *Uppaal*. Gerd Behrmann, Alexandre David, and Kim G. Larsen, 28 Nov. 2006. Web. 10 Dec. 2015.
* "Language Reference." *UP4ALL Inc*. Ed. UP4ALL. International AB, 1 Jan. 2009. Web. 10 Dec. 2015.
* Vaandrager, Frits, MR. "Introduction." *Knowledge First?* (2014): n. pag. *Mbsd*. Institute for Computing and Information Sciences, 1 Jan. 1970. Web. 10 Dec. 2015.
* Alves-Foss, Jim, Center For Secure And Dependable Software, University Of Idaho, and Id 83844-1010 Moscow. "Provably Insecure Mutual Authentication Protocols The Two Party Symmetric Encryption Case." *Provably Insecure Mutual Authentication Protocols: The Two-Party Symmetric-Encryption Case* (2008): n. pag. *Csrc*. Center Jim Alves Foss for Secure and Dep Endable Software. Web. 10 Dec. 2015.
* Llanos Tobarra, Diego Cazorla, Fernando Cuartero, Gregorio D ́ıaz, and Emilia Cambronero. "Model Checking Wireless Sensor Network Security Protocols: TinySec + LEAP." *Citeseerx*. Escuela Polit ́ecnica Superior De Albacete, n.d. Web. 10 Dec. 2015.
* Marta Kwiatkowska, Gethin Norman and David Parker. "Probabilistic Model Checking in Practice: Case Studies with PRISM1." *Probabilistic Model Checking in Practice: Case Studies with PRISM 1* (n.d.): n. pag. *Prismmodelchecker*. School of Computer Science/University of Birmingham. Web. 10 Dec. 2015.