Component-based Modeling and Analysis of Embedded Systems

Davor Slutej

Västerås, Sweden – 2008/06

Mälardalen University
School of Innovation, Design and Engineering

Supervisor Paul Pettersson
Abstract

Development of real-time embedded systems is associated with a set of challenges not usually of concern for developers of desktop applications – such as high demands on safety, functional and temporal correctness and resource efficiency. Embedded systems are also increasing in complexity and the time to market is constantly shrinking. Significant effort is put, both within academia and commercially, into developing methods and processes to meet the aforementioned challenges.

Two such methods are the component-based software engineering and the formal verification through model-checking. The former promises shorter development cycles and complexity management through reusability, interchangeability and encapsulation while the latter primarily addresses the safety and correctness concerns.

In this thesis, we have evaluated parts of a lightweight (low memory and processing overhead) component technology, SaveCCT – developed within the scope of the SAVE project. SaveCCT and the modeling language SaveCCM are specifically designed for real-time embedded applications with particular emphasis on automotive industry, and are suitable for other resource constrained control systems. The evaluation is made by means of a case study in which a previously published turntable industrial control system is modeled using the Save IDE (Integrated Development Environment).

In addition to modeling the control system using the component approach found in SaveCCT – UPPAL Tool was used to model the behavior of the environment. Finally, UPPAAL PORT model-checker is employed to formally verify the system.
Acknowledgements

First of all I would like to thank my supervisor Paul Pettersson for giving me this opportunity as well as excellent guidance and much needed pointers during the course of the thesis.

I would like to thank Aurelien Monot and Florian Noyrit for designing and implementing the initial SaveIDE. Aurelien in particular patiently endured a constant stream of questions during those early days…

…and extra special thanks to Séverine Sentilles and Anders Pettersson for maintaining the SaveIDE.

John Håkansson did excellent job with UPPAAL PORT and the Eclipse plug-in. Special thanks for the analysis of THAT trace and “_urgent”.

Last but not least, I would like to thank people from the SAVE group and related projects whose discussions and input have been inspiring, Christina Seceleanu, Jan Carlsson, Jagadish Suryadevara and Ivica Crnkovic.

Thank you all.
# Table of Contents

Abstract ......................................................................................................................... 2
Acknowledgements ........................................................................................................ 3
Table of Contents ........................................................................................................... 4

1 Introduction .................................................................................................................. 6
  1.1 Real-Time Embedded Systems ............................................................................. 6
  1.2 Software Engineering ......................................................................................... 6
  1.3 Save Project .......................................................................................................... 7
  1.4 Formal verification through model-checking ...................................................... 7
  1.5 The scope of this master thesis .......................................................................... 8

2 Related Work ............................................................................................................... 8

3 Preliminaries ................................................................................................................ 9
  3.1 Model-checking with UPPAAL .......................................................................... 9
    3.1.1 UPPAAL Tool IDE ...................................................................................... 9
    3.1.2 Example: Wolf, Goat and Cabbage Puzzle .................................................. 11
  3.2 SaveCCM ............................................................................................................ 15

4 Modeling Environment ............................................................................................... 17
  4.1 Save-IDE ............................................................................................................. 17
  4.2 UPPAAL PORT .................................................................................................. 21

5 Case study .................................................................................................................... 23
  5.1 Turntable Industrial Production System ............................................................. 23
  5.2 Workflow ............................................................................................................ 24
  5.3 Sensors and actuators ......................................................................................... 25
  5.4 Design Strategy .................................................................................................. 26
  5.5 Simplifications .................................................................................................... 26
  5.6 Architecture Model ............................................................................................. 26
    5.6.1 Clock ............................................................................................................ 29
    5.6.2 Turntable ..................................................................................................... 29
    5.6.3 Loader and Unloader ................................................................................... 35
    5.6.4 Driller .......................................................................................................... 35
    5.6.5 Tester ........................................................................................................... 35
  5.7 Environment Model .............................................................................................. 39
    5.7.1 Global declarations ...................................................................................... 39
    5.7.2 ClampSim .................................................................................................... 40
    5.7.3 Driller_P1 and Driller_P2 ............................................................................ 41
### Table of Contents

5.7.4   DrillerTest ................................................................. 43  
5.8      Linking System and Environment ................................ 44  
5.9      Simulation ............................................................... 45  
5.10     Requirements ............................................................ 47  
6       Results and analysis ..................................................... 49  
7       Conclusion .................................................................. 50  
8       References .................................................................. 52  
9       Appendix – Control System Model .................................. 54  
9.1      Architecture ............................................................... 54  
9.2      Turntable ................................................................. 55  
9.3      Loader .................................................................... 57  
9.4      Driller ..................................................................... 59  
9.5      Tester ..................................................................... 61  
9.6      Unloader ................................................................... 63  
10      Appendix – Environment Model .................................... 65  
10.1     Global Declarations .................................................... 65  
10.2     TesterSim ................................................................. 66  
10.3     TurntableSim .............................................................. 67  
10.4     ClampSim ................................................................. 68  
10.5     Driller_P1 ............................................................... 69  
10.6     Driller_P2 ............................................................... 70  
10.7     DrillerTest ............................................................... 71  
11      Appendix – Properties (turntable.q) ............................... 72  
12      Appendix - Verification results (std output only) ................ 75
1 Introduction

Developing robust, efficient and correct software is a demanding task – in particular if there are deadlines and budgets involved.

While an inefficient application may require a better (more expensive) hardware platform to run on – and thus potentially reduces the economic benefits – a faulty and crash-prone application may, on top of economic loss and property damage, result in human injuries and even loss of life. Examples of such systems are internet banking, vehicular brake systems and artificial pacemakers.

1.1 Real-Time Embedded Systems

Real-time systems are such systems where the correct behavior requires not only functional correctness but also predictable temporal behavior. For instance, a basic function of an artificial pacemaker is to stabilize the heartbeat – if the device occasionally pauses for a minute or so it would be a very bad pacemaker.

Another category of systems that a pacemaker falls into is embedded systems. Embedded is here used to refer to the fact that the function of the pacemaker is completely contained in a small purpose-built unit\(^1\). By having hardware that is specialized for a specific application (or a set of somewhat similar applications) it can be reduced in size, manufactured less expensively or in some other way adapted to the particular environment that it is expected to function in – often significantly easier (cheaper) than compared to a general purpose computing platform such as a PC. Notice, however, that an embedded system is not required to be physically small.

1.2 Software Engineering

Software engineering as a discipline has come up with several methods with the purpose to increase the quality and timeliness of the software development process – including software development models such as classic Waterfall method and modern Agile and iterative methods such as Scrum, Extreme Programming, and the Unified Process [1].

Similarly, programming languages have evolved into arguably higher levels introducing concepts such as object orientated design [2], generic programming such as the templates in C++ [3] and garbage collection in languages such as C#.

Component based development can be considered an extension of the OO (Object-oriented) model although the components are not necessarily implemented using classes and objects.

A component is a part of a specific component framework that defines a contractual obligation – an interface that the component must adhere to in order to fit within that component framework. It is up to the component framework to specify what constitutes a component by specifying the minimal interface that the component needs to implement.

The exact definition of what constitutes a software component is debated in books and articles; see for instance [4], where a whole chapter is devoted to discussion on how other authors are defining (software) components.

---

\(^1\) The shell of the pacemaker device, not the human body.
An example of the usage of the term component is the TComponent\textsuperscript{2} class in the Borland Delphi programming language \cite{5}\cite{6}. The TComponent class defines the super class from which all Delphi components must inherit from. Somewhat simplified, compared to a vanilla class (TObject) in Delphi – the TComponent and its descendants provide the user\textsuperscript{3} with object (instance of a class) identification through object names and a mechanism for parent/child owning relationship between the instances. This in turn allows hierarchical composition that is required for modern GUI (Graphical User Interface) design. A direct descendant of the TComponent class is the TControl class which is the base class for all graphical components.

In Java, similar functionality- i.e. a component framework mostly targeted at design of rich GUIs, is proved by the java.awt (Abstract Windowing Toolkit) package \cite{7} as well as other packages. In both Delphi and Java cases, components are built as classes that inherit from a super-class or through interface implementation and as such are bound to the programming language used.

Another well known component framework is the COM (Component Object Model), a Microsoft technology. As opposed to the two component frameworks mentioned above, COM specifies the contract on the binary level (instead of using programming language constructs). The contractual obligation between the software entity and the COM framework is that the software entity has to correctly bind to at least the IUnknown interface. Here we use software entity to indicate that a component does not need to be an object in a particular language. See \cite{8} for a detailed account on the internals and particulars of the COM.

\section{1.3 Save Project}
SAVE\textsuperscript{4} project is a research project whose aim is to establish a software engineering discipline for safety-critical real-time embedded systems with the main focus on the vehicular industry.

One of the outputs of the Save project is the SaveCCM (SaveComp Component Model) \cite{13} – a lightweight component model suitable for resource constrained applications; in line with the goals of the SAVE project, SaveCCM was primarily designed for automotive applications. In this thesis we will use the SaveCCM as the architectural framework for modeling a case study outside of the automotive domain.

SAVE project provides more than a component model though, the SaveCCT (SaveComp Component Technology) encompasses analysis, verification and synthesis of systems built using the SaveCCM \cite{17}.

In \cite{12}, formal syntax and semantics of SaveCCM have been defined allowing for transformation of SaveCCM component into a network of Timed Automata such as that available in the UPPAAL Tool.

\section{1.4 Formal verification through model-checking}
Once a formal syntax and semantics are established it is possible to perform formal verification of the modeled system. Verification is not be confused with testing. Testing involves selecting a number of test cases, performing the tests associated with each and

\footnote{2 By convention, class names begin with a capital T in Borland Delphi}

\footnote{3 In the sense of “user of the class” which can be regarded as the programmer or the designer of the system, or more precisely the IDE (Integrated Development Environment) that the programmer and designer are using.}

\footnote{4 Component Based Design of SAfety Critical VEhicular Systems, \url{http://www.mrtc.mdh.se/save/}}
examining the results. Even though the tests might by performed by automatic means, for any complex enough system, the coverage offered by the test is not complete. Verification on the other hand, tests that a particular property is completely satisfied by the model at hand. This is not to say that verification replaces testing – testing is in the end performed on real software and real hardware and may detect problems that simply were not part of the models state space.

Model-checking has been successfully used to verify real-time systems and communication protocols and improvements in time and space consumption by better algorithms and data structures have resulted in drastic improvements [14].

1.5 The scope of this master thesis

This thesis is a part of the SAVE Demosstrator project; the project includes subprojects such as the development of the Save IDE – a graphical environment for SaveCCM, the code generation tools, the Save-OS etc.

The scope of this thesis is evaluation of the available tools with regard to architectural design and formal verification – it does not include synthesis for target hardware.

Besides the tools that are a “standard” part of the Save-IDE (i.e. the architecture editor, the TA editor and the UPPAAL PORT model checker), we have also used the UPPAAL Tool for environmental modeling.

The rest of this thesis is organized as follows: Chapter 2 gives a brief overview of the related work. The following two chapters (3 and 4) introduce the preliminaries that are considered necessary for the reader to follow the case study easily.

Chapter 5 is the main part of the thesis – it gives a detailed account of the case study, followed by analysis of results chapters 6.

References are listed in chapter 7.

The rest of the chapters are appendices that provide the complete model for the control model and the environment as well as all verified properties and results.

2 Related Work

The bases of the case study are several papers in which the Turntable production system has been specified and verified using different methods and tools.

In [18] the authors are taking a model specified in the $\chi$ language and developed a translation of it into PROMELA, the input language of the SPIN model-checker [21] which was then used to verify several properties of the model, such as the absence of a deadlock situation.

The same industrial production system was later used in a larger study in [19]. In this extensive paper the $\chi$ model was translated into the specification languages of three model-checkers: CADP, Spin and UPPAAL – comparing both the ease of conversion, the expressiveness of each of the specification languages and the abilities and performances of the respective model-checker.

Even later, in [20] – the Turntable production system was implemented in the COMDES-II component-based software framework, whose goals are similar to that of the SAVE project. Once implemented in COMDES-II, the authors developed a semantic transformation of the COMDESS-II model into UPPAAL Timed Automata allowing for formal verification of a set of properties similar to those in [19].
Of particular interest are the translations into UPPAAL, as UPPAAL is also used in this study. However, whereas the papers mentioned above have translated the models into UPPAAL TAs, in this thesis we are using an extension of UPPAAL (UPPAAL PORT) that has been adapted to “read” (and take advantage of the restrictive semantics) of the SaveComp Component Technology.

3 Preliminaries

Before presenting the case-study, this chapter gives the reader an overview of the methodologies, languages and tools that were used.

3.1 Model-checking with UPPAAL

We start the preliminaries with a brief introduction of the UPPAAL Tool\(^5\). An UPPAAL model will later be used to represent the environment in the case study. While the behavior of the component is captured with another tool, namely the TA editor of the Save-IDE, the general principles are quite similar.

\[
\text{Figure 1 - UPPAAL 4.0 splash screen}
\]

UPPAAL (UPPsala university + AALborg university), see Figure 1, is a (free) tool for modeling, simulation and verification of systems that can be modeled as a network of Timed Automata. TA (Timed Automata) are an extension of the FSM (Finite State Machine). Originally, see [9], FSM where extended with real-valued clock variables. UPPAAL TAs are further extended with integer and Boolean variables (and arrays there of), numerous other features (see [11]) and of special interest to the case study - the specification language allows for a C-style syntax in declarations. In the rest of the thesis, when referring to a Timed Automaton we are referring to the extended Timed Automaton available in UPPAAL.

In this chapter we give a brief informal overview of the UPPAAL Tool, with a focus on an example in a later section. For a more thorough introduction to UPPAAL see [10] and [11].

3.1.1 UPPAAL Tool IDE

The three main parts of the UPPAAL Tool are the editor, the simulator and the verifier. Intuitively, the editor is used to created the system, see Figure 2. In the figure, a TA (a demo system supplied with the UPPAAL distribution) is shown.

The locations of the TA are connected with edges. The firing of an edge is conditioned by several elements.

Firstly, the TA is allowed to stay in the same location for as long as the location invariant is satisfied. It must fire an edge if the location invariant becomes unsatisfied. If there is no

\(^5\) Available online from http://www.uppaal.org/
location invariant specified at all (and given that the urgent or committed location is not used) then the TA is free to delay in the location forever. Forgetting this has caused plenty of headaches during the verification.

Secondly, an edge can be protected by a guard. A guard expresses a condition (using discrete variables or clock) that must be satisfied in order for the edge to be “open”. If there is such an edge, then the TA can fire it. Notice “can” instead of “must”. If there is no forcing condition (such as the location invariant becoming unsatisfied), then the edge can be fired for as long as it is open, or in fact never.

Section 2.3 “Understanding Time” in [11] gives a good, example rich, explanation of how guards, invariants and time interact in UPPAAL.

![Figure 2 - Editor tab of the UPPAAL IDE, showing an example Timed Automaton and highlighting locations, edges, guards etc.](image)

Edge Updates are the assignment(s) that take place when an edge has been fired. In the figure above, the marked assignment $x=0$ is a reset of a real-valued clock $x$. Not shown in the figure, but used in the example below are the synchronization channels. They are used when two automata need to fire their respective edges synchronously – in which case both edges will be annotated by the same synchronization label (if the channel name is $test$, then one TAs edge will be annotated with $test?$ and the other with $test!$). An edge can also have a Selection, which non-deterministically binds a value to a variable from a specified range.

The two other main parts of the UPPAAL Tool, the simulator and the verifier will be used in the example below.

3.1.1.1 Requirement Specification

Once we have a model, because it is described using formal semantics we can ask the verifier “questions” using the query language of UPPAAL. There questions, or specifications of either desirable or undesirable properties can be categorized (we are only considering categories that we will be using in the example and the case study) as follows:
Reachability properties – written $E<> p$. Where $p$ is a state formulae, i.e. an expression that describes a state of a (network of) TA(s). The state formulae can express if the TA is in a particular location, if a clock or a discrete variable has a certain value (such as $\mathsf{P.state} = 100$). Also, inequalities can be expressed ($x < 100$). See UPPAL online help or [11] for a full reference. A reachability property, $E<> p$ will be satisfied if there exists a path in the state space that reaches the given state formulae. In other words, “Is it possible that $p$ happens?” If the verifier finds such a path, it produces a trace that shows the path through the state space. Sometimes, it is desirable that the property is not satisfied, for instance if the state formulae $p$ indicates an error state. This is the only type of property that will be used in the example below.

Safety properties – written $A[] p$. There is a second variant ($E[] p$) which we do not use in this thesis. In this case, the state formulae $p$ must be true in all states. This type of property will be used extensively in the verification of the system in the case study, often in the form $A[] p \implies q$, i.e. it is always true than when $p$ is true, $q$ is also true.

Finally, liveness properties, and in our case particularly the “leads to” property – written $p HT q$. This property expresses that once $p$ has happened, $q$ must eventually happen in all paths leading from $p$.

Once again, note that the types of properties briefly explained here are only the ones that will be used in the example and the case study. For an in depth account, consider section 2.2 “The Query Language” in [11].

3.1.2 Example: Wolf, Goat and Cabbage Puzzle

In this section a classic river-crossing puzzle “Wolf, Goat and Cabbage” (WGC) is modeled using UPPAL.

The WGC puzzle defines the problem as follows: a person needs to transport a wolf, a goat and some cabbage across a river using a boat. However, with him on the boat the person can only transport one item. Additionally, if the wolf and the goat are left unattended the wolf will eat the goat. Likewise the goat and the cabbage cannot be on the same side of the river without the person present.$^6$

Is there a way for the person to safely transport all three items across the river?

A quick way to fail: A man takes the cabbage with him on board the boat and starts crossing the river. The wolf and the goat are left behind so the goat gets eaten by the wolf.

Modeling is an abstraction process in which one should reduce the problem to a model that is sufficient to solve the problem or more often some aspect(s) of the problem.

In this case, we could start with 6 objects: Man (person), Goat, Cabbage, Wolf, Boat and River.

The Boat has two essential properties:

- it can carry a person and possibly another item
- it can cross the river

The River also has two essential properties:

$^6$ Neither the wolf nor the goat will escape if left alone – because they are tied down. But they gain super powers if food is nearby thus allowing them to break loose and eat the food!
- it can only be crossed by a boat (in our simple world)
- it divides the world into two sides

We can reduce the number of objects by stating that the Man, Goat, Cabbage and Wolf have a property that states on which side each of them is. Additionally, only the Man can switch side alone – each of the other objects requires to be accompanied by the Man to cross. This eliminates the need for a Boat and a River without removing the essence of the problem.

Each of the remaining objects will be modeled by a dedicated Timed Automata. The Timed Automata will need to be synchronized in the cases when the Man wants to bring one of the items across the river. Another set of synchronizations is necessary for cases when the Wolf eats the Eat and the Goat eats the Cabbage.

The time it takes to cross the river is not essential to the problem, so let’s assume that the crossing is instantaneous – it takes no time to pass over to the other side. This may seem to be a reasonable reduction as time is not mentioned in the puzzle formulation.

In the model, the banks of the river will be represented numerically by 0 or 1, and by West and East.

Let’s first start with the declarations, shown in Figure 3 and explained below.

```plaintext
1. urgent chan eatC, eatG;
2. chan crossW, crossE;
3. chan priority default < eatG < eatC; // MUST eat if possible
4. int[0,1] wolf = 0;
5. int[0,1] goat = 0;
6. int[0,1] man = 0;
7. int[0,1] cabb = 0;
8. bool wolfCanEat() {
   return (wolf==goat) and (wolf!=man);
}
9. bool goatCanEat() {
   return (goat==cabb) and (goat!=man);
}
```

**Figure 3 - Global declarations**

The first line declares two synchronization channels that are used when the Cabbage or the Goat gets eaten.

The second line declares two additional synchronization channels used when an item is carried by Man to the other side.

Line 4 declares the priorities of the synchronization channels. The need for this line is a consequence of excluding the passing of time in the model! The command states that eating has the highest priority of all channels – meaning that if eating is possible it must not wait (declared urgent) and it must happen before any other transition (such as crossing the river).

To further clarify: because crossing the river takes no time at all, crossing the river twice also takes no time at all. So the Man could take the Cabbage to the other side and return back without the Goat being eaten by the Wolf. Eating the Goat takes no time but neither does crossing the river twice so the model-checker would produce a very short and, given the model, correct but perhaps unexpected result. Introducing a delayed state, such that each crossing take at least certain amount of time solves the problem as well because then there would in fact exists a time interval where the Goat and the Wolf are alone in one side of the river.
Lines 6-9 define integer variables that denote which side of the river the object is at. Finally the function queries wolfCanEat() and goatCanEat() return true if the action is possible given the rules of the puzzle. For wolfCanEat() for example: the wolf and the goat must be on the same side as each other, but the Man must be on the opposite side at the same time.

The first Timed Automata we will consider is the Man TA, shown in Figure 4. Two nodes are defined W and E, for West and East respectively. Consider the start node W, there are two edges leading to node E. Both edges have the assignment statement \( \text{man}=1 \) which is used to set the global variable \( \text{man} \) to correspond to the current side of the river. The difference is in the synchronization action, the upper edge synchronizes with another TA using the crossE channel. Having been declared with a question mark, the edge transition can synchronize with another TA’s edge declared with an exclamation mark. Only two Timed Automata can synchronize on each transition i.e. only one item can cross the river together with the Man.

Edges without the synchronization model the case in which the Man is crossing the river alone.

![Figure 4 - Man TA – only two nodes are needed, W(est) and E(ast).](image)

Figure 4 shows the Timed Automaton that defines the behavior of the Goat. As with the Man TA, there are only two nodes used to model which side of the river the Goat is currently positioned at. There is now only one way (edge) to cross from one side to the other and the edges both require synchronization channels – effectively prohibiting the Goat to cross the river without the Man doing so at the same time.

Two additional nodes have been added, both with synchronization channels. Node “stuffed” signifies that the Goat has eaten the Cabbage. There are no edges leaving this node - because if this node is entered the puzzle is not solved. The fact that the Goat is still on one side of the river and could in fact become eaten by the Wolf is no longer of interest.

The edges leading to “stuffed” are protected by guards. A result of true from the function goatCanEat() is required for the edge transition to be enabled.

As a side note: The global variables used to denote which side the object is at, are only used in the goatCanEat() and wolfCanEat() functions.

Node “dead_goat” is entered when the Wolf is able to start consuming the Goat. Similarly to the “stuffed” node – there are no edges leading out from this node. These edges are not protected by guards – instead the guards are placed on relevant edges in the Wolf TA shown in Figure 7.

![Figure 5](image)
Figure 5 - Goat TA – two additional nodes are need. In “stuffed” the Goat has eaten the Cabbage and in “dead_goat” the Goat has been eaten.

The Timed Automata for the Cabbage and the Wolf, shown in Figure 6 and Figure 7 respectively complete the network of Timed Automata needed to analyze the puzzle.

Figure 6 - Cabbage TA. As with other “items” the Cabbage can only cross the river by synchronizing with another (Man) Ta.

Figure 7 - Wolf TA.

The solution to the puzzle is gained by first defining the property for model checking. The puzzle is solved if all Timed Automata are in the location E at the same time. This is expressed easily within the UPPAAL requirement specification language – the requirement is shown in Figure 8.

\[ E \leftrightarrow (\text{Wolf.E and Goat.E and Cabbage.E and Man.E}) \]

Figure 8 – Verification; Does a state exists in which all four timed automata are in their respective "east-side" state?

The model-checker will find the property satisfied and if a diagnostic trace is enabled (shortest trace is suitable) then we can see which sequence of events creates a successful crossing of the river.

A screenshot of such a trace, giving the shortest solution to the puzzle is shown in Figure 9. For the interested reader, there is a similar but more complex puzzle available in the UPPAAL distribution – the Bridge puzzle also involves time as an essential component.
In this section we give a brief overview of SaveCCM. Indeed, only parts of the SaveCCM that are used in the case study will be explained. For more details on SaveCCM, we kindly refer to [13].

SaveCCM, SaveComp Component Model, is a lightweight component model specifically designed for real-time embedded applications. The design goals are to allow analysis and predictability – not to create a general purpose component model.

A SaveCCM component, shown in Figure 10, is an architectural element that encapsulates a behavior of a part of a system. The interface of a component is defined by a set of ports, see Figure 11. In SaveCCM, ports are divided into data ports and trigger ports.

Data ports carry information and have a data type associated with them. Trigger ports control the activation of components. A port can be either an input or an output port. For data ports, this classification depends on if the components reads or writes the data to it.
A component can have any number of input trigger ports. When all of the trigger ports have become active, the component – which is initially passive – is activated. When the component is activated, the values of the input data ports are read, whereupon the function associated with the component is executed. Once the component has finished execution, it once again becomes passive and the values of the output data ports are written. Finally, any output trigger ports are activated. These output trigger ports can in turn be connected to another component or set of components. The read-execute-write semantics of the component execution means that once the component has read its input ports the results of the execution, eventually written to the components output ports, are not affected by any concurrent activity.

Ports have name associated with them and for data ports also a type. Connections between the ports are restricted so that:

- Trigger ports are only allowed to be connected to trigger ports
- Data ports are only allowed to be connected to other data ports of the same data type
- Input ports are only allowed to be connected to output ports (and vice versa)

SaveCCM defines a special type of component called a Clock which only has an output trigger port and provides a periodic triggering signal.

Aperiodic triggering is provided by external triggering sources which form a part of the I/O interface as defined by the run-time environment. This is analogous to having components triggered by either periodic task activation due to the Operating System or external hardware interrupts.

Composition of components is made by connecting their ports; data ports for data communication and triggering ports to explicitly specify the execution precedence.
4 Modeling Environment

While the previous chapter describes the modeling and requirement specification languages this chapter focuses on the graphical development environment that is used by the end-user. This includes the modeling and verification tools and the interaction thereof. In particular – the chapter explains which editors and tools are available, the files that the tools generate and the necessary integration configuration.

4.1 Save-IDE

The main modeling environment is the Save-IDE. Save-IDE is a collection of plug-ins for Eclipse\(^7\) IDE. Eclipse IDE has become a popular development platform, in particular within the open source or free software community, and although a vanilla Eclipse installation is synonymous with Java development – users have provided plug-ins for a wide range of languages and software development processes.

Besides Eclipse, Save-IDE builds upon another set of Eclipse plug-ins namely EMF (Eclipse Modeling Framework), GMF (Graphical Modeling Framework), GEF (Graphical Editing Framework), Acceleo and other third party plug-ins. The details of Save-IDE implementation are beyond the scope of this document but interested readers are kindly referred to the “Save-IDE – Developer Documentation” available from the Subversion repository of the Save-IDE project on SourceForge.net\(^8\).

Save-IDE provides\(^9\) a number of editors, which are summarized in Figure 13 together with the UPPAAL Tool introduced in a previous chapter.

---


\(^8\) Save-IDE project at Sourceforge.net: [http://sourceforge.net/projects/save-ide/](http://sourceforge.net/projects/save-ide/)

\(^9\) At the time that this thesis was conducted, only the architectural and TA-based verification parts were implemented.
Figure 13 - Editors and Tools. The single arrow indicates the sole direct dependency – meaning that all other editors and tools can be used independently (they communicate via files). The UPPAAL Port plug-in is used for simulation.

The SaveCCM editor is used to create the components and interconnections among them. See Figure 14. The figure shows two components with a single connection in between them. This particular connection is between the Combined Data/Trigger ports. The Headers section of the components refers to the entry functions that would have been used for synthesis on a target system.

For analysis purposes however, we instead assign a model to each of the component. SaveComp is designed so that a component can be delivered with different models to facilitate analysis of different aspects of the system and to support several analysis tools.
Figure 14 - SaveIDE, showing the architectural (SaveCCM) editor. Notice the highlighted palette to the right of the main design area. The two arrows indicate that each component has been assigned a model (for analysis). The type of the model is TA and the names are sender and receiver respectively.

In this case we assigned a Timed Automata model to the components.

Editing the TA model is done using the TA editor, see Figure 15.

The Timed Automaton shown (sender) has two locations, start and final. This is somewhat misleading though because the actual TA representing the component has several more locations that are not directly accessible to the user. These locations are for instance the IDLE location, in which the TA is delaying while the component is in the passive state. Also, the read and write (of the data ports) is modeled implicitly outside of the scope directly visible to the user. Notice however that the passive state, as well as the reading and writing phase are dictated by the semantics of the SaveCCM and as such should not be allowed to be changed by the user. What is accessible to the user is the modeling of the execution phase.
SaveCCM TAs differ from UPPAAL TAs in certain ways. The start and final nodes have what an entry and exit assignment respectively. These assignments replace what would otherwise have been assignments on edges leading to and from the IDLE location. Locations have location invariants ($x <= 1$), just as in UPPAAL TAs. Further, edges have assignments ($i = j$, $x = 0$), guards ($x >= 1$), and selections ($j : int [0, 1]$).

Synchronization channels are not allowed in SaveCCM TA though – which makes sense because the components are only allowed to communicate through ports and the values of the ports are read and written outside of the scope of the user definable behavior.

To bind TA variables to ports of the component, a mapping file needs to be created using the mapping editor. Examples of this are available in the next chapter where we present the case study.

Once the design of has been made in the SaveIDE, each of the TAs used in the component needs to be “compiled” into an intermediate XML file. Also, the architectural design is compiled into a separate intermediate XML file. These files are then merged together (in order to connect TA variables with component ports) using the mapping file(s) and a final XML file is created. The specifications of the XML file format are available in [15].

Figure 16 summarizes the files created using the SaveIDE editors and code generation tools.

We use SaveIDE to design the control system. In order to verify the system, we also need to model the environment. For this, the UPPAAL TA is more suitable – because we do not want the environment to be triggered periodically.
For verification, we are using the UPPAL PORT Tool. The Tool can load both SaveIDE generated models and those generated using UPPAL Tool.

UPPAAL PORT is an extension of UPPAL in which a Partial Order Reduction Technique (hence PORT) has been implemented in order to use the extra information provided by the semantics of the component framework to reduce the time and space requirements for model-checking [16].

UPPAAL PORT, in its current\(^{10}\) distribution, consists of two parts – the Eclipse plug-in and the server executables. The former provides the user interface while the latter is the run-time for simulation and verification. The plug-in is not necessary for verification – instead the server executable can be used directly via the command-line interface (and files created manually or generated by the Save-IDE).

The linking of the control model and the environmental model is done via the .port file, as shown in Figure 17.

In the next chapter we introduce the case study and explain the design choices.

---

\(^{10}\) March 2008, v0.48
Figure 17 - Linking Save-IDE architectural and behavioral models with UPPAAL-based environment model. The rounded boxes at the top signify the two design environment with their final outputs shown in the boxes below. The .port file, containing pointers to both the .save file and the .xml file is then loaded into the UPPAAL PORT.
5 Case study

The bulk of this thesis is a case study in which we specify and implement (for simulation and verification only) a simplified industrial production system.

The goal of the study is to design the system using the SaveCCM and implement the design by using the available tools. The focus is on the tool evaluation and formal verification of safety and liveness properties; the system is not going to be implemented on real hardware within the time-frame of this thesis. This will allow for a somewhat simpler design as will be explained further down.

5.1 Turntable Industrial Production System

Mechanically, the system consists of a turntable (a rotary disc), which has slots that can hold some kind of product. In this particular case, there are four equal and equally spaced product slots. See Figure 18 for an overview of the system.

![Figure 18 - Top view of the turntable system – shown is the rotating disc itself, the four product slots and the processes associated with each stationary position. The dotted circular arrow indicates the direction of the rotation.](image)

The turntable is driven by a motor that enables rotation and each rotation iteration is set to 90 degrees.

The product enters the production cycle in position 0. Products are drilled in position 1 and the results of the drilling are tested in position 2. The results of the test are then used to determine if the product should be removed (unloaded) from the turntable or if it should stay on the turntable and go through the drill and test cycle again. A more detailed description is presented in the next section.
All slots are allowed to be occupied at the same time and all processes are allowed to operate in parallel.

5.2 Workflow

Initially, the turntable is stationary and all product slots are assumed empty. The product slots are aligned with the tools.

The system sends the order to the loading tool and waits for it to signal that it has finished loading the product. The exact details of how the loading tool performs the loading are not specified.

In the description below, we assume that the product slots have been filled and thus the turntable has rotated, although this is not explicitly stated until the end of the section.

In the next slot, the product is drilled. When the product is in place, the drilling process is started by clamping the product to disallow movement during the drilling. The drill is then started and lowered until a sensor signals that the drill has reached its low position. The drill is then raised until the top position sensor is signaled. The drill is stopped and the product is unclamped. Drilling requires significantly more interaction between the controlling component and the environment, compared to the Loading process.

In the next position, the results of the drilling process are being tested. This is performed by lowering a mechanical tester into the hole. A sensor will signal once the tester has reached the bottom – if this happens within a certain time than the drilled hole is proper. If the tester does not reach the bottom within the time interval (or not at all) – the whole is considered faulty and the product needs to be drilled again. The Tester, together with the Driller is the processes that have to perform non-trivial interactions with the environment. The Tester also includes timing information.

In the final position, the product is either removed from the turntable or not – depending on the result of the test in the previous slot.

Once the Tools have been signaled to start working – the Turntable waits for the Tools to signal that they have finished working. The Turntable then starts the motor and waits for a signal that signifies that a 90 degrees rotation has been completed after which the cycle repeats.
5.3 Sensors and actuators

This section summarizes the sensors and actuators used in the system.

<table>
<thead>
<tr>
<th>Turntable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sensors</td>
<td></td>
</tr>
<tr>
<td>sCompleted</td>
<td>Signaled when the turntable has completed a 90 degrees rotation</td>
</tr>
<tr>
<td>Actuators</td>
<td></td>
</tr>
<tr>
<td>aRotate</td>
<td>Power the turntable motor</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Loader</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sensors</td>
<td></td>
</tr>
<tr>
<td>sLoaded</td>
<td></td>
</tr>
<tr>
<td>Actuators</td>
<td></td>
</tr>
<tr>
<td>aLoad</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Driller</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sensors</td>
<td></td>
</tr>
<tr>
<td>sLocked</td>
<td>Signaled when the clamping order has completed, i.e. the product is held firmly in place</td>
</tr>
<tr>
<td>sUnlocked</td>
<td>Signaled when the clamp has completely released the product</td>
</tr>
<tr>
<td>sDrillDown</td>
<td>The drilling has reached the lowest position</td>
</tr>
<tr>
<td>sDrillUp</td>
<td>The drill is in the topmost position</td>
</tr>
<tr>
<td>Actuators</td>
<td></td>
</tr>
<tr>
<td>aClamp</td>
<td>Order clamping (or unclamping if set to false)</td>
</tr>
<tr>
<td>aDrillOnOff</td>
<td>Switch drill on or off (depending on the value)</td>
</tr>
<tr>
<td>aDrillMoveDown</td>
<td>Move the drill down</td>
</tr>
<tr>
<td>aDrillMoveUp</td>
<td>Move the drill up</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tester</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sensors</td>
<td></td>
</tr>
<tr>
<td>sTesterDown</td>
<td>The tester has reached the bottom</td>
</tr>
<tr>
<td>sTesterUp</td>
<td>The tester is in the topmost position</td>
</tr>
<tr>
<td>Actuators</td>
<td></td>
</tr>
<tr>
<td>aTesterMoveDown</td>
<td>Order the tester to move down</td>
</tr>
<tr>
<td>aTesterMoveUp</td>
<td>Order the tester to move up</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Unloader</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sensors</td>
<td></td>
</tr>
<tr>
<td>sUnloaded</td>
<td></td>
</tr>
<tr>
<td>Actuators</td>
<td></td>
</tr>
<tr>
<td>aUnload</td>
<td></td>
</tr>
</tbody>
</table>
5.4 Design Strategy

Our overall design strategy is simplicity. The system should be easy to understand and maintain and at the same time be resource efficient. The design is foremost intended for evaluation of the SaveCCM framework, the Save IDE and the verification tool-chain.

A central idea is that the specific operations performed by the tools in the case study are just a particular version of the system. There could be other versions where the drilling and testing (even loading and unloading) have been replaced by some other functions – still the tools’ interface should only differ towards the environment – the interface between the tool and the control hub (in this case the turntable) should be the same.

Also, there could be a version of the turntable that features another number of product slots – perhaps the product is painted before it leaves the turntable. The tools do not need to be aware of this number as long as the proper alignment with the product slots is maintained by some other part of the system (in this case the turntable component).

Similarly, the same tools could be used in another arrangement without the specific layout enforced by the turntable – any kind of assembly line.

The turntable and the tools are considered reusable and, more importantly, replaceable components.

5.5 Simplifications

The system is overly optimistic in that it does not anticipate for events such as broken tools (for instance a continuous stream of negative test results could indicate that either the drill is broken or that there is something wrong with the tester). Also, there are no sensors to actually indicate that the product is in place (or has been removed) – this information is provided by the loader and unloader tool and not by the sensors in product slots.

There are no deadlines detected or handled. Here by deadline we mean how long it should maximally take for a tool to signal that it had finished operating. Likewise, the 90 degrees rotation can take infinitely long without the system detecting it as a failure.

There is no emergency stop button. In fact, the system starts automatically and it cannot be stopped by any means other than pulling out the plug.

As there is not target hardware, the schedulability of the system is not analyzed. Further, because the system design shown in the following sections, the system is naturally synthesized into a single task so schedulability analysis become trivial even for real hardware. The simplification in the model for verification is that the execution time is not modeled at all, thus all components execute in zero time (we still have periodic triggering by means of a SaveCCM clock component and the events and sequencing is time dependant – but the execution time of each component invocation is zero).

5.6 Architecture Model

This is the step where the Turntable System is broken down into subsystems. Subsystems are then mapped to SaveCCM components (a more complex system than the case study presented would further benefit from composite components and assemblies) – and the interconnections are defined.

In this case, there is a convenient and obvious mapping possibility: assign one SaveCCM component per controllable physical device (i.e. the turntable and the four tools). Given this mapping, the role of each component is to by reading sensor and directing the actuators control its associated part of the system.
This results in five components: Turntable, Loader, Driller, Tester and Unloader.

Other implementations of the system, such as the one used in the [19], include an additional Main Controller component – whose main purpose is to coordinate other components and ensure sequencing of the events.

![Diagram showing component interconnections and dependencies]

**Figure 19 - Initial breakdown into subsystems. The subsystems can be directly mapped to components. Solid lines represent component interconnections, while the dashed lines represent dependencies due to shared data.**

Closely related to the subsystem breakdown is the definition of the interfaces between the components. An interface is here defined as the set of interconnections between the ports of the components. Recalling the design strategy this is an important part because we strive to define the same interface between the turntable and all components. In the case of the system at hand, it is simple – we start by stating the requirements:

- The tools need to receive the information about the product in the slot that they are operating on
- The tools need to return the information about how they have affected the product
- The tools need to “know” when to start operating on product
- The tools inform the turntable when they have finished operating so that the turntable can initiate a rotation

This calls for four ports per connected tool-component. For now, let’s call these ports “SlotStatus” (integer), “SlotResult” (integer), “StartWork” (boolean), “WorkFinished” (boolean) – assuming proper direction, these ports directly satisfy the demands listed above. This group of ports defines a “Common Interface” for connection between a turntable (or indeed any similar assembly line or tool controller) and a single tool. Note however that SaveCCM does not have any semantics for grouping of ports – the mentioned relationship between the ports exists only conceptually.

Next decision is on the triggering of the components. We assume that all components are running on the same hardware thus there exists on parallelism\(^{11}\). The triggering sequence starts with the clock. The clock triggers the Turntable component which in turns triggers the Loader component. The triggering chain continues until the Unloader component is triggered. The system is then idle until the next iteration triggered once again by the clock.

The considerations so far lead to a design shown in Figure 20. Notice that this is not presented in proper SaveCCM graphical notation but features a simplified layout.

---

\(^{11}\) When considering the control system model only – there is parallelism when the environment model is augmented.
Figure 20 - Turntable design, simplified layout. The thicker blue lines, starting and ending with triangles, show the triggering path, starting with the clock generator.

At this point the required components, their interconnections as well as the external ports (i.e. signals and actuators) are known. In the next section we will look at the implementation of the individual components before continuing with the environment, the simulation and the verification. For reference, Figure 21 shows the actual design using SaveCCM notation taken from the SaveIDE. Notice the extra connection (dtStatus, dtStart, dtResult and dtFinished) – these are used by the Test Timed Automata as explained in a later section. A bigger version of the figure is available in Chapter 9.

Figure 21 - Turntable model, actual screenshot from SaveIDE.
5.6.1 Clock

The Clock is generating periodic triggering. The properties of the Clock are set to:

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Period</td>
<td>1.0</td>
</tr>
<tr>
<td>Jitter</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Thus we are assuming a perfect clock with a periodicity of one time unit.

5.6.2 Turntable

The Turntable is the first component that executes in the chain of components. Because this is the first component that we will explain it will receive the most attention. The implementation strategy explained here is followed by other components as well.

The Turntable component has two essential functions: it controls the rotation of the (mechanical) turntable, and is also a hub to which all of the Tool components are connected. With four tools connected we need four instances of the “Common Interface” that was discussed in the beginning of this chapter. This translates into 4 sets of 4 ports – which are shown together with the other ports needed in Table 2.

<table>
<thead>
<tr>
<th>Port</th>
<th>Type</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TrigIn</td>
<td>N/A</td>
<td>Trigger In</td>
<td>Activates this component</td>
</tr>
<tr>
<td>TrigOut</td>
<td>N/A</td>
<td>Trigger Out</td>
<td>Activates the next component</td>
</tr>
<tr>
<td>aRotate</td>
<td>bool</td>
<td>Data Out (I/O)</td>
<td>Actuator that starts the rotation of the turntable</td>
</tr>
<tr>
<td>sCompleted</td>
<td>bool</td>
<td>Data In(I/O)</td>
<td>Sensor that signals that a rotation has been completed</td>
</tr>
<tr>
<td>SLOT 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S0_Status</td>
<td>integer</td>
<td>Data Out</td>
<td>Current known status of the product in position 0</td>
</tr>
<tr>
<td>S0_Result</td>
<td>integer</td>
<td>Data In</td>
<td>The result after the associated tool has finished working</td>
</tr>
<tr>
<td>S0_Start</td>
<td>bool</td>
<td>Data Out</td>
<td>Signal the associated tool to start operating</td>
</tr>
<tr>
<td>S0_Finished</td>
<td>bool</td>
<td>Data In</td>
<td>Signaled when the associated tool has finished operating</td>
</tr>
<tr>
<td>SLOT 1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Note from the table above that Sn_Status and Sn_Result ports are of the integer type. These integers encode the status of the product in the respective slot\(^{12}\). The Turntable Component does not actually need to know what the codes mean (except of the code 0 – empty slot that is used during initialization) because it never operates on the products itself – only the connected components need to agree on the meaning of the numbers. Nevertheless, we list the slot codes here, in Table 3.

<table>
<thead>
<tr>
<th>Slot Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Empty Slot</td>
</tr>
<tr>
<td>1</td>
<td>New Product</td>
</tr>
<tr>
<td>2</td>
<td>Drilled Product</td>
</tr>
<tr>
<td>3</td>
<td>Product Tested Bad</td>
</tr>
<tr>
<td>4</td>
<td>Product Tested Good</td>
</tr>
</tbody>
</table>

**Table 3 - Slot Codes**

SaveCCM allows several different models to be assigned to the Component – in order to verify different aspects of the model. We assigned a TA type model that can be used for verification using UPPAAL Port.

Figure 22 shows the (SaveCCM) Timed Automaton that encapsulates the behavior of the Turntable Component.

---

\(^{12}\) To be more precise, “slot n” is here used to mean the “product slot currently aligned with the stationary tool position n”
The Timed Automaton uses parameters as a means of mapping its internal variables to the port of the Component that it is associated to. The parameters are, together with other properties, defined in the Turntable TA Properties list, as shown in Table 4.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declarations</td>
<td>See Listing 1</td>
</tr>
<tr>
<td>Parameter</td>
<td>slot0,slot1,slot2,slot3, trnStart,trnFinished, changeslot0,changeslot1,changeslot2,changeslot3, run0,run1,run2,run3, fin0,fin1,fin2,fin3</td>
</tr>
<tr>
<td>TA name</td>
<td>Turntable</td>
</tr>
</tbody>
</table>

Table 4 - Turntable TA properties

The names of the properties do not need to be equal to the names of the ports of the associated component – the mapping is handled by the mapping dialog shown in Figure 23. The mapping file (an xml file called turntable.tamap) is generated by the Save-IDE and is used in code generation. Code generation tool use the name of the Timed Automaton to decide which mapping file to use – this means that, if the same behavior (TA) model is assigned to multiple components then the names of the ports must be same in both component.

Next, we will look into the behavior of the component.

On each invocation, caused by the trigger input port, the Turntable TA enters the Entry node. All edges in the Timed Automaton are red which indicates that the edges are urgent – the transition defined by the edge must be, if it is possible according to the guards, made without delaying. If this were not the case, the Timed Automaton could delay in a location – perhaps indefinitely if nothing else forces a transition. A reminder: in a Timed Automaton and given the semantics use by UPPAAL Timed Automata – if the transition is not explicitly forced (by location invariants for instance) the Timed Automaton – being non-deterministic, can perform a transition at any point in time allowed by the guards or indeed never. The “never” part is
problematic because if we do not explicitly force progress we could not verify “always leads to” properties (such as Event A always leads to Event B).

The properties of the Entry node are:

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entry Assignment</td>
<td>evaluate()</td>
</tr>
<tr>
<td>Invariant</td>
<td></td>
</tr>
<tr>
<td>Location Name</td>
<td>Entry</td>
</tr>
</tbody>
</table>

**Table 5 - Turntable TA - Entry node properties**

Immediately after entering the Entry node, the Timed Automaton calls the evaluate() function. The evaluate() function performs all control logic. As the edges leading from the Entry node are urgent, immediately after calling the evaluate() function the automaton will transition to either the Exit location directly or transition to Exit location via an intermediate node Turning. The path taken depends on the **mode** variable (explained shortly) as can be seen in Figure 22 (**mode==S_TURNING** is an edge guard). The two possible paths (or rather their guards) cover all possible values of the **mode** variable; if all values were not covered, the Timed Automaton would block in the Entry node.

The edge leading from the Turning location to the Exit location has no guard and is also urgent – meaning that the Turning location will be left immediately.

The transitions thus appear to make no sense! In fact, the only reason we are doing this (in particular going via the intermediary Turning location) is because the names of the locations are visible in the simulation trace.

Let’s return to the evaluate() function. This function, together with other declarations and code, is found in the Turntable TA – Declarations which is part of the Turntable TA – Properties. The declarations (the code of component) are shown in Listing 1. We will insert the explanation directly into the code.

```
Variable to track how much time has passed. Not essential to the function of the component.

clock GLOBALTIME;

The following constants represent the possible states of the component. The behaviour of the component is going to be capture by a finite state machine coded in the C-style language supported by the UPPAAL Port.

const int S_START=2;
const int S_TURNING=1;
const int S_IDLE=0;

The mode variable holds the current state of the FSM that defines the behaviour of the component. The [ ] brackets indicate the range of the variable. This is not standard C – it is used by the model-checker. Notice however that the code would be just as functional without the range specification. All components are declared so that mode variable is equal to 0 (zero) when the component is in the S_IDLE state. We are going to exploit this in the properties for verification.
```
The following constants are the slot codes as specified in Table 3.

```c
const int SL_EMPTY=0;
const int SL_LOADED=1;
const int SL_DRILLED=2;
const int SL_TESTBAD=3;
const int SL_TESTGOOD=4;
```

The $ sign in the variable names indicate that they are in fact parameters that will be mapped to the ports of the component. Parameters must be present as variables in the code.

```c
int $slot0=SL_EMPTY;
int $slot1=SL_EMPTY;
int $slot2=SL_EMPTY;
int $slot3=SL_EMPTY;
```

Case 2, the Turntable is in the S_TURNING state. It will continue to be in this state until the $trnFinished is signalled – indicating that the 90 degrees rotation has been completed. The resetting of the $trnStart variable creates a falling edge on the aRotate actuator, completing the pulse. This implies that the aRotate actuator is used to start the rotation, not to power the motor. Had the later been the case (so that the actuator needed to stay high during the whole rotation), the
$trnStart=false statement would have been moved inside the next if statement block.

If a $trnFinished is true, i.e. the environment has signalled that the 90 degrees rotation has finished, the Turntable Component sends a signal to all four Tools to start working and changes it’s own mode to S_IDLE.

```java
else if (mode==S_TURNING)
{
    $trnStart=false;
    if ($trnFinished) {
        $run0 = true;
        $run1 = true;
        $run2 = true;
        $run3 = true;
        mode=S_IDLE;
    }
}
```

In the S_IDLE mode, the $run parameters (port) are reset (the Tools should have read this information and started to act accordingly in the previous invocation). The code then checks if all Tools are reporting that they have finished operating in which case their results are read back and the turntable is once again ordered to rotate. Notice that the call to rotateSlots() is placed here. It could have been made at the end of the rotation as well – because the reading of slot status during a rotation is undefined behaviour. (A more robust version would for instance set the slot status to a special value indicating ongoing rotation.)

```java
else if (mode==S_IDLE)
{
    $run0 = false;
    $run1 = false;
    $run2 = false;
    $run3 = false;
    if ($fin0 and $fin1 and $fin2 and $fin3) {
        $slot0 = $changeslot0;
        $slot1 = $changeslot1;
        $slot2 = $changeslot2;
        $slot3 = $changeslot3;
        rotateSlots();
        $trnStart=true;
        mode=S_TURNING;
    }
}
```

Listing 1 - Turntable TA - Declarations

The reader could here pose a question, why the control logic is not expressed graphically using locations, edges and guards. While this is certainly possible, it is not convenient. The Timed Automaton will, on each triggering of the Component, start in the Entry location and must always exit – it cannot stay in an intermediate node because if it does the Component never finishes executing. We cannot hack around this by triggering each component only once and staying within the Timed Automaton either, because the reading and writing to the ports is only performed on entering and leaving the Timed Automaton, respectively. In order to do the control logic graphically, we would need to store the last location prior to exiting (via the Exit location) and on Entry recall the last location and make a transition to it. This introduces a lot of visible “plumbing”.
The implementation of the Turntable Component, as can be seen, is easily expressed using C-style code “embedded” in a Timed Automaton. Expanding the system to take into account, for example, the retardation of the rotation of motors would mean introducing new states, reacting to more sensors, actuators and more complicated control logic but the structure remains.

5.6.3 Loader and Unloader

As the components follow the general structure laid out in the previous section, we refer the reader to appendix chapters 9.3 and 9.6 for implementation details of these two components.

5.6.4 Driller

As with the previous two components we defer the implementation details to appendix chapter 9.4.

The appendix has all graphical design and code for all components.

5.6.5 Tester

All of the Tools components interact with the Turntable in similar fashion so we will only show in detail how this interaction works for one component, namely the Tester.

Table 6 lists the ports used by the component. The first two ports are the triggering ports. The Tester component is triggered by the Driller component, and in turn triggers the Unloader component. Because the Tool components are not interacting with each other directly the order of triggering past the Turntable component is not important. The particular order selected mimics the passing of a product through the product slots. The triggering ports exist in this pair for all components except of the Unloader component which has no TrigOut port, although it could have had a TrigOut but without a connection.

<table>
<thead>
<tr>
<th>Port</th>
<th>Type</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TrigIn</td>
<td>N/A</td>
<td>Trigger In</td>
<td>Activates this component</td>
</tr>
<tr>
<td>TrigOut</td>
<td>N/A</td>
<td>Trigger Out</td>
<td>Activates the next component</td>
</tr>
<tr>
<td>sTesterUp</td>
<td>Bool</td>
<td>Data In (I/O)</td>
<td>Signals that the tester is in the topmost position</td>
</tr>
<tr>
<td>sTesterDown</td>
<td>Bool</td>
<td>Data In (I/O)</td>
<td>Signals that the tester is in the down most position</td>
</tr>
<tr>
<td>aTesterMoveDown</td>
<td>Bool</td>
<td>Data Out (I/O)</td>
<td>Order the tester to move down</td>
</tr>
<tr>
<td>aTesterMoveUp</td>
<td>Bool</td>
<td>Data Out (I/O)</td>
<td>Order the tester to move up</td>
</tr>
<tr>
<td>TURNTABLE INTERFACE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Input</td>
<td>integer</td>
<td>Data In</td>
<td>Slot code of the associated product slot</td>
</tr>
<tr>
<td>Output</td>
<td>integer</td>
<td>Data Out</td>
<td>Result after operating on the</td>
</tr>
<tr>
<td>Start</td>
<td>bool</td>
<td>Data In</td>
<td>Order (from the Turntable) to start operating on the product</td>
</tr>
<tr>
<td>---------------</td>
<td>---------</td>
<td>---------</td>
<td>---------------------------------------------------------------</td>
</tr>
<tr>
<td>Finished</td>
<td>Bool</td>
<td>Data In</td>
<td>Signaled by the Tester when it has finished operating on the product</td>
</tr>
</tbody>
</table>

| Table 6 - Ports used by the Tester Component |

The external ports (I/O) used by this component features two sensors and two actuators as expected given the system description.

The section in the table marked with “Turntable Interface” defines the interface with the Turntable component. This is a mirror of the “SLOT 2” section in Table 2.

Figure 24 shows the layout of the Tester Timed Automaton. As with the Turntable TA shown in previous section, all edges are urgent and the intermediate locations “MovingUp” and “MovingDown” are employed only as visual feedback in the simulation trace.

The Entry location has the evaluate() function as the Entry Assignment property. In the Tester TA we make a small variation, as compared to the other Timed Automata. The Exit location has a property called Exit Assignment. For that we specify function doSetupOutPorts(). As a component’s only means of communication with other components and the environment is through ports, and the ports are only written to at the end of the activation (when the TA leaves the Exit location) – it does not really matter if we call function doSetupOutPorts() in the Entry location or in the Exit location. If the intermediate nodes did some processing that depended on the intermediate port values, it would have mattered.

Listing 2 shows the code that defines the behavior of the Tester component. The structure of the code is similar to that of the Turntable Component so we will only focus on the differences.

---

13 Internally, the component “sees” the changes to port-mapped variables.
The lead of the declarations is same as for the Turntable TA, except that the modes of the Tester TA are different.

```c
const int S_REA
DY=0;
const int S_MOVING_DOWN=1;
const int S_MOVING_UP=2;
int[S_POINTS, S_MOVING_UP] mode=S_POINTS;

const int SL_EMPTY=0;
const int SL_LOADED=1;
const int SL_DRILLED=2;
const int SL_TESTBAD=3;
const int SL_TESTGOOD=4;
```

Parameters are defined next.

```c
bool $sensorTop, $sensorDown;
bool $moveUp, $moveDown;
bool $active, $finished;
int $slotstatus, $slotresult;
```

`ticks` is a local variable that is used to keep track of time. In a real system, calls to a real-time OS would be used instead.

```c
int ticks;
```

The `evaluate()` function that performs the actual control logic. Once again it is a simple implementation of a Finite State Machine with the `mode` variable used to indicate the current state.

Note how the `ticks` variable is incremented while the tester is moving down. According to the specification, the test has failed if the sensor indicating that the tester is in the down position has failed to signal within a certain time period (in this case it has to be signalled within two ticks).

```c
void evaluate() {
    if (mode==S_POINTS) {
        if ($active) {
            $slotresult=$slotstatus;
            if ($slotstatus==SL_DRILLED) {
                $finished=false;
                mode=S_MOVING_DOWN;
                ticks=0;
            } else {
                $finished=true;
                mode=S_POINTS;
            }
        }
    } else if (mode==S_MOVING_DOWN) {
        ticks++;
        if (ticks<=2) {
            if ($sensorDown) {
                $slotresult=SL_TESTGOOD;
                mode=S_MOVING_UP;
            }
        } else {
            ...
        }
    }
    ...
```
The doSetupOutPort() function, called from the Exit location. In this example, the two actuators are directly depending on the current state of the Finite State Machine.

As mentioned, this function could have been called from the (end of the) evaluate() function without changing the behaviour of the system.

```c
void doSetupOutPorts() {
    $moveDown=mode==S_MOVING_DOWN;
    $moveUp=mode==S_MOVING_UP;
}
```

**Listing 2 - Tester TA – declarations**

Next section deals with the environment model.
5.7 Environment Model

Before we can start the verification of the control system, we need to model the behavior of the environment\textsuperscript{14}. As with the control system model, the environment is going to be a simplification of the real world. Modeling every aspect of the machinery is not feasible and, depending on the focus of the verification, might not be overly useful either.

For instance, it takes some time for the drill to stop rotating after it has received the order to stop. We are not going to model this because it is, in our case study, not essential to the proper function of the system. We will stop the drill in the topmost position of the drill – the drill bit is not in contact with an object and restarting the drill before it has completely halted has no negative consequences.

When the drill receives an order to start, it takes a small amount of time before it reaches full rotational speed. This will not be modeled either because this time is expected to be very short, much shorter than the time it takes to lower the drill until it reaches the object.

As we stated previously, the control system model is not anticipating any extraordinary events to happen (such as the drill bit getting broken). We intend to verify the correctness of the system under “good conditions” – so also the environment model can be restricted to “good conditions”.

The environment is modeled using UPPAAL Tool. The model that is generated, is however not a “legal” UPPAAL model and cannot be simulated nor verified using UPPAAL Tool.

In an early version of the case study, the environment was modeled as SaveCCM components; however we soon realized that the passive nature of SaveCCM components is not suitable for environment modeling. Specifically, we want the environment to immediately react to actuators (or if we need a delay, we would like to model it explicitly) instead of it waiting to be triggered by a clock (or another component). Triggering introduces discretization of time in the environment model which is undesired. Another problem is the growth of state space if the discrete time steps are shortened in order to reduce the latency due to discretization.

5.7.1 Global declarations

The environment model needs to communicate with the control system model. This link will be established later outside of the UPPAAL Tool - the details of which are explained in section 5.8. From within the UPPAAL Tool, global variables are used. These variables, that are accessible from all Timed Automata (as well as the SaveCCM system defined using SaveIDE), are defined in the global declarations part of the UPPAAL model, see Figure 25.

\begin{verbatim}
clock GlobalClock;
/**\ 
 * -----------------
 * \ 
 * \ TURNTABLE \ 
 * \ 
 * \ 
 * // INPUT 
 * bool aRotate = false; // Start rotation
 * // OUTPUT
 * bool sCompleted = false; // Completed rotation
 * /**\ 
 * \ 
 * \ 
 * // INPUT
 * // false = Unclamp, true = Clamp
\end{verbatim}

\textsuperscript{14} Testing, as opposed to formal verification, would involve actual environment.
bool aClamp = false;

// OUTPUT
bool sLocked = false;
bool sUnlocked = true;
/**\-------------------------------
/**\ ---------------- DRILL --------- \**/

// INPUT
bool aDrillOnOff = false;
bool aDrillMoveDown = false;
bool aDrillMoveUp = false;

// OUTPUT
bool sDrillDown = false;
bool sDrillUp = true;
/**\-------------------------------
/**\ ---------------- DRILL TEST ---- \**/

int dtStatus = 0;
int dtResult = 0;
bool dtStart = 0;
bool dtFinished = 0;
/**\-------------------------------
/**\ ---------------- TESTER ------ \**/

// INPUT
bool aTesterMoveDown = false;
bool aTesterMoveUp = false;

// OUTPUT
bool sTesterUp = true;
bool sTesterDown = false;

Figure 25 - UPPAAL global declarations - the variables specified as global are used as interface between the control system model and the environment

The variables are given initial values; these values are set up so that the Timed Automata do not make an initial transition before the values from the Control System model are received. In the following section we will show the details of the Driller-related Timed Automata, for a complete design of the environment, please refer to the Appendix chapter 10.

The Driller is modeled using three Timed Automata, representing the three controllable parts of the Driller process. These Timed Automata do not intercommunicate but are controlled by the same SaveCCM component.

5.7.2 ClampSim

Figure 26 shows the Timed Automaton that defines the behavior of the clamp. The function of the clamp is to lock the product in place so that the drilling can be carried out.

The Timed Automaton is initially in the Unlocked\textsuperscript{15} location. If the actuator aClamp goes high (==1), the Clamp TA transitions to the Locking location – as specified by the edge guard. The edge also resets the clock claClock and resets the sUnlocked signal.

The TA will delay in the Locking location before transitioning to the Locked location. The delay time is controlled by the Locking location invariant (claClock<=CLAMP\_TIME) – which means that the transition must be made no later than after CLAMP\_TIME has passed, and the edge guard claClock>=CLAMP\_TIME. Effectively, this means that the time it takes to lock the product in place is determined to be exactly CLAMP\_TIME. A similar procedure is modeled for the unlocking operation.

\textsuperscript{15} The location-coloring (Unlocked = green, Locked = red) is user defined.
The use of synchronization on the special _urgent channel guarantees that the transition takes place as soon as possible. As with other Timed Automata, if this was not enforced – by the semantics of the UPPAAL Timed Automata, the transition could take place anywhere between immediately and never (which breaks the verification). Further, if there were no location invariants assigned to the Locking and Unlocking locations – the Timed Automaton could delay in those locations forever, even after a transition was made possible by the edge guards.

If we wanted to model that the time it takes to lock (or unlock) the product is not exactly equal to CLAMP_TIME, we would use two constants such as MIN_CLAMP_TIME and MAX_CLAMP_TIME – and use the MAX_CLAMP_TIME for the location invariant and the MIN_CLAMP_TIME as the edge guard (assuming of course that MIN_CLAMP_TIME < MAX_CLAMP_TIME). An example of this is shown in the next section.

5.7.3 Driller_P1 and Driller_P2

The two automata depicted in Figure 27 and Figure 28 are used to model the behavior of the drill.

Automata Driller_P1 is a simple one that toggles between the driller being on and off, controlled by the actuator aDrillOnOff. The mode toggles are considered instantaneous which is good enough for our purposes as is explained earlier.

---

16 _urgent is understood by Uppaal Port, but not by the standard Uppaal – hence the red text (which indicates an error in the Uppaal model). It is an urgent channel defined internally in Uppaal Port.
Figure 27 - Driller_P1 – reacts to aDrillOn actuator to turn the drill on and off

The second Timed Automata used to model the Driller is the Driller_P2. It is used to model the movement of the drill up and down. The initial position is in the location Driller_UP. When the actuator aDrillMoveDown is triggered, the clock x is reset and the drill is modeled to be moving down (location Driller_MOVING_DOWN). The lowest position is reached within \( \text{MIN\_DOWN\_TIME} < x < \text{MAX\_DOWN\_TIME} \). The \( \text{MIN\_DOWN\_TIME} \) and \( \text{MAX\_DOWN\_TIME} \) are local constants supplied for the Driller_P2. See Figure 29.

Figure 28 - Driller_P2 – moves the drill up and down

Moving the Driller upwards is triggered by the actuator aDrillMoveUp and a similar setup of location transitions as for the downward motion. Notice that, while our implementation of the case study provides two separate actuators for upwards- and downward movements – this could be reduced to a single actuator triggering both movements as in the original paper.

$$\text{clock } x;$$

$$\text{const int } \text{MIN\_UP\_TIME} = 2;$$

$$\text{const int } \text{MAX\_UP\_TIME} = 2;$$

Figure 29 - Driller_P2 - Local variables (clock) and constants
If the location invariants (purple colored) such as \( x \leq \text{MAX\_DOWN\_TIME} \) for the Driller\_MOVING\_DOWN location were not present, the environment could very well stay in that location forever as no progress is enforced.

### 5.7.4 DrillerTest

DrillerTest, shown in Figure 30, is a Timed Automaton that differs from the previous Timed Automata in that it does not model the behavior of the environment – instead it is used to check that the component controlling the Driller finishes the drilling process within a single work cycle (one work cycle being the period between two consecutive rotations of the turntable). The global variables that control the transitions of this Timed Automaton are logically connected to the internal ports of the control system model (i.e. it “listens in” on the interface between the components) as opposed to the external ports (signals and actuators).

![Figure 30 - DrillerTest - a Test Timed Automaton](image)

Remember that the Driller component (as any other tool component) starts its work cycle upon receiving the “start work” signal – depicted as “dtStart” in the figure. The guard “\( \text{dtStatus}==1 \)” is Driller specific – if there isn’t any product in the relevant product slot there is no need to start the test.

Once in the “started” location, the Timed Automaton waits for a falling edge, meaning that the turntable component has reset the “start work signal”. Once this is detected the DrillerTest is in the “working” location (this happens on the second invocation, or activation, of the Turntable component).

For a sequence of events to be correct the Driller component must produce a correct result (\( \text{dtResult}==2 \)) and signal to the Turntable that it has finished working. This must happen before a second “start work” signal has been sent by the Turntable.

If a second “start work” signal is received or if the result is not correct – the DrillerTest Timed Automaton goes into an “error” location.

Non-existence of a path through the state space that leads to the DrillerTest Timed Automaton entering the “error” location is then the property that needs to be satisfied.

This type of Test Timed Automata allows checking of complex sequence of events, without the need for verification related flags or other non-functional constructs in the control system model. The sole intrusion is on the architecture level because of the need for delegation between the “probe points” and the environmental model.

In the next section we discuss the connection between the control system model and the environment model.
5.8 Linking System and Environment

The architecture model and the associated behavior of the control system do not need to change for production code generation and verification by model-checking. The connection between the control system model and the environment is made by utilizing external I/O ports that act as sensors or actuators depending on the direction of the delegation.

In our case, the environment is not connected to actual hardware but instead to a simulation of the environment as described in the previous section. For a production system, the I/O ports would have been connected to drivers for external sensors or actuators. A SaveCCM component does not “see” any difference thought.

UPPAAL PORT is loaded with a .port\(^{17}\) file that points to the application and environment. Figure 31 shows the file used in the case study.

```xml
<!-- Loads ENVIRONMENT and SYSTEM models in UPPAAL PORT -->
<PORT>
  <ENVIRONMENT filename="Turntable_Env.xml"/>
  <APPLICATION filename="Turntable.save"/>
</PORT>
```

Figure 31 - .port used to load environment and control system in UPPAAL Port. Turntable_Env.xml is the UPPAAL environment model generated by UPPAAL Tool. Turntable.save is the architectural and behavioral model generated by Save-IDE.

Sensors and actuators are defined as external I/O ports in SaveCCM. In the UPPAAL environment model, the concept of sensors and actuators is represented by global variables in the global declarations section of the model.

When both the SaveCCM model and the UPPAAL model are loaded, UPPAAL PORT will match I/O ports with global variables in UPPAAL model based on the names of ports and variables respectively. This simply means that if there is an outgoing I/O port with the same name as a global variable in UPPAAL model – writing to this port will change the value of the global variable. The notion of a port and delegation direction only exists in the SaveCCM world, so the UPPAAL TA could also change the value of a global variable; this is not explored in the case study but could be used for instance to simulate random bit flips caused by disturbances on the communication link.

UPPAAL PORT will signal an exception if there are external I/O ports defined in the SaveCCM model but there is no global variable to connect to, see Figure 32 for an example. Similarly, the exception is signaled if the global variable is actually declared as a constant using the const modifier (mentioned here because the message is exactly the same – which could be confusing).

However, this check is only limited to name, if the data types are not compatible no exception is signaled and the verification may produce unexpected results.

```xml
<PORT>
  <ENVIRONMENT filename="Turntable_Env.xml"/>
  <APPLICATION filename="Turntable.save"/>
</PORT>

EXCEPTION: Unknown identifier: bRotate
```

Figure 32 - Error output from verifyta - I/O port bRotate does not have a corresponding global variable

\(^{17}\) The automatically generated .port file (at the time of writing this) is not actually used, because it always refers to “env.xml” for environment model whereas we want our environment model to be “turntable_env.xml”. Also, because the SaveIDE (currently) does not provide an “Open in UPPAAL Port” function for .port files, but it does so for .save files, we have changed the file extension to .save.
5.9 Simulation

With both the Control System model and the Environment model ready and linked – we can simulate the system using the UPPAAL PORT plug-in\(^\text{18}\) for the Eclipse IDE, by selecting the linking file described in the previous section. This presents us with a dialog similar to the one shown in Figure 33.

The simulator group box in the upper left corner of the dialog allows us to step forward and backwards in the state space while the Transitions group box has a list box that allows a selection of possible transitions for the next step. In the figure below, there is only one transition possible for the next step, the transition from the Entry to the Exit location in the Driller component.

The State group box lists all the variables associated with the currently selected component (the Driller component in this case). By clicking on the components boxes in the main window another component’s variables can be investigated.

\[\text{Figure 33 - Example simulation trace using the UPPAAL PORT plug-in. Data Transfer means that exchange of data between the two components is taking place via their respective ports. The direction of data transfer is indicated by the arrow.}\]

\(^{18}\) The simulation can be performed using command-line tools as well
The trace shown in the figure above shows the Turntable (C1) component going through the rotation phase while the tool components are idling. This can be seen because the Turntable TA is going through the intermediate location “Turning”. Remember from previous chapters that this is in fact one reason why we opted to use the intermediate locations – there is no functional benefit from doing so. This is further demonstrated in Figure 34 which shows the trace after the first product has been loaded, the turntable rotated and the next cycle of tools working is initiated. The Tester and Unloader component are idle, because they do not yet have a product to operate on. The Driller however is initiating the Clamping mode and the Loader is loading a new product in the product slot 0. This particular trace shows an older version of the model where the Turntable is going through the intermediate location Working while the tools are in operation.

Figure 34 - Simulator trace. The horizontal red line indicates the switch between the end of turntable rotation and the beginning of tools working cycle.

The simulator is particularly useful during the initial design and during fault/tracing because of its graphical nature and ability to single step forwards and backwards in the state space. Additionally, the Check function of the simulator can be used for model-checking a single property.

In the next section we will look at the properties and their verification.
5.10 Requirements

In this chapter, we will analyze a selection of properties used to verify the safety and liveness of the control system. A complete set of properties can be found in appendix chapter 11. The results of model-checking are found in appendix chapter 12.

When specifying the properties, we must use the UPPAAL PORT assigned component identifiers, not the component names as specified in the SaveCCM architecture model. Listing 3 shows the identifiers needed. These numbers are visible in the Simulator and may unfortunately change if the system architecture is changed.

Listing 3 - UPPAAL PORT assigned component identifiers

The first property that we want to have satisfied is the absence of a deadlock situation in the state-space. A deadlock is a situation in which the system cannot progress further. In a real-time system this is often caused by two tasks mutually excluding each other from acquiring a lock on a resource. While we, by design, cannot have a deadlock situation (the SaveCCM does not allow resource locks) – a deadlock could also be caused by a fault in the environment model.

The absence of deadlock is checked by a property shown in Listing 4. Translated into English this property reads: “Is it true that there exists no state in which a deadlock situation is present?”

Listing 4 - Property: No deadlock exists

The absence of a deadlock in the state-space does not mean that the system is guaranteed to progress. In particular, the control system could be continuing with the component trigger without the components progressing through their respective finite state machines. We know that the Turntable (as all components we have designed) have a variable mode which indicates the current mode of operation (the state in the Finite State Machine). By convention, this variable is equal to zero while the component is idle (in the sense that it has finished operating and waits for the next cycle). For the Turntable component mode==0 means that it is waiting for the Tool components to finish operating on the product. Further, mode==1 means that the Turntable is rotating and waiting for the signal that the rotation has completed. A system is thus progressing, if it continuously alternates between these two modes. We formalize this in Listing 5.

Listing 5 – A set of properties that guarantee that the control system is progressing

The first two properties check for existence of states where the C1 (Turntable) is in the two modes we are interested in. The 3rd and 4th properties are the leads-to properties. They are satisfied if each path in the state-space that contains the left-side condition has the right-hand side condition afterwards. In other words, the 3rd and 4th properties, in English: “Is it true, that every time the Turntable is in the turning state, it will eventually go into idle state”, and “Is it
true that, that every time the Turntable is in the idle state, it will eventually start turning”. If both properties are satisfied, the Turntable will continuously alternate between the two states and it can only do it if all involved components are finishing what they are starting (according to the design).

We can ask the model-checker this specifically as well. Listing 6 shows such a property for the Loader component. For the rest of the components, check the appendix. Here we are basing the check on the value of the ports as opposed to internal component variables as was the case in the previous example.

Listing 6 - Does the Loader component, once started - eventually signals that it has finished working?

An obvious safety property, which we want to be satisfied, is that either the turntable should be turning or the tools should be working but not both at the same time. There are several ways to express this; here we will explain the one in Listing 7. C1 is the Turntable component; C2, C3, C4 and C5 are the Tool components. As mentioned previously, all components have a mode variable that is controlling the internal Finite State Machine of each component. By convention, a value of zero means that the Component (and in effect the part of the machine that the component is controlling) is in idle mode.

Listing 7 - Either turning or operating on products - but never both!

The property above demonstrates the expressiveness of the query language. Notice that what we desire is that the expression is always equal to zero. Because we cannot have (by design) a component with a negative value of the mode variable – the right parenthesis will be greater than zero if ANY of the tools is working. For the product to be zero – the C1 (Turntable) component must be idle. Likewise, if the Turntable is greater than zero (i.e. rotating) then ALL tools must be idling (for the property to be satisfied). With this query we also allow ALL components to be idle at the same time.

Next, state correspondence checks – meaning that the internal state of the component corresponds to what is happening in with the environment. An example of such a property is shown in Listing 8.

Listing 8 - The (environmental model) of the turntable should NOT be turning while the Turntable component "thinks" that it is not

For the driller and tester, we have two actuators each for moving the tool up or down. What if both actuators are active, should the – lets say driller – be moving up or down? We can regard this as undefined behavior instead and create a property that checks that no conflicting orders are (ever) sent to the environment by the control system. See Listing 9 for an example: the property is satisfied if there does not exist a state where both actuators are active.

Listing 9 - No conflicting orders to the driller

There are a total of 32 properties specified in the appendix, using the techniques described above.
6 Results and analysis

The properties were analyzed using a laptop PC with an Intel T2600 processor running at 2.16GHz\(^{19}\) and equipped with 2GB of RAM.

The verification is performed at command line using the verifyta\(^{20}\)(.exe) application provided with the UPPAAL PORT. For this purpose a .bat file was used, shown in Listing 10.

```
verifyta.exe -C -s -u -s0 -no -t1 -Y "!! LOAD ME !!.save" Turntable.q 2>verifyta_error.txt
```

**Listing 10 - verifyta.bat - used for verification**

The “!! LOAD ME !!.save” is the name of what would normally be a .port file. The contents are shown in Figure 31 in a previous chapter. “Turntable.q” is a file containing all properties that we want to check; this file is listed in the appendix chapter Appendix – Properties (turntable.q). “2>verifyta_error.txt” is a redirection of error output to the specified (verifyta_error.txt) file. The error output contains a trace for those properties where it is applicable.

The selected options are summarized in Table 7

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-C</td>
<td>Disable most memory reduction techniques. <strong>UPPAAL PORT, and thus verifyta.exe, is under development at the time of writing this thesis so this option is selected to disable certain experimental reduction techniques.</strong></td>
</tr>
<tr>
<td>-s</td>
<td>Do not display the progress indicator. <strong>Cosmetic, doesn’t clutter the command-line window and also allows the redirection of standard output to a file.</strong></td>
</tr>
<tr>
<td>-u</td>
<td>Show summary after verification (incorrect for liveness properties) <strong>Shows how many states where stores and explored in addition to property being satisfied or not</strong></td>
</tr>
<tr>
<td>-s0</td>
<td>Optimize space consumption = none. <strong>Similar to –C, as we are using beta version of model-checker.</strong></td>
</tr>
</tbody>
</table>

\(^{19}\) Running on AC power with Power options set not to allow the CPU to slow-down

\(^{20}\) An unofficial version of the tool was used. Model-checking this particular system will not work with UPPAAL PORT v0.48. At the time of writing this, UPPAAL PORT v0.49 is not yet officially released but that version should have necessary features implemented.
Table 7 - verifyta.exe options used during verification

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>n0</code></td>
<td>Select extrapolation operator = automatic. <em>Using automatic, fine-tuning beyond the scope of this thesis.</em></td>
</tr>
<tr>
<td><code>-t1</code></td>
<td>Generate diagnostic information on stderr = shortest trace (disables reuse) <em>Indicates that the diagnostic traces (such as when a property is not satisfied) should generate a shortest possible trace.</em></td>
</tr>
<tr>
<td><code>-Y</code></td>
<td>Display traces symbolically (pre- and post-stable) <em>Does not affect verification, only the generated traces. This option made it easier to debug the traces.</em></td>
</tr>
</tbody>
</table>

With these options, verifying all 32 properties takes less than 45 seconds on the specified hardware – this is less than two seconds per property.

The model-checker needs to explore a maximum of 38166 states. The maximum is reached when verifying properties where the model-checker needs to explore the whole state space (such as the non-existence of a deadlock situation). Property 28 (“is there a state where the turntable component is in the rotate-mode”), on the other hand, takes only 6 states to verify.

The verification used less than 18MB of memory, at peak when all 32 properties are checked in a single-run.

### 7 Conclusion

In this thesis we have used a toolset consisting of the SaveIDE component based modeling environment that facilitates the SaveCCM language for system composition and modeling, UPPAAL Tool for environment modeling and finally the UPPAAL PORT tool for simulation and verification.

As the thesis was progressing, work concentrated around a few focus areas. Firstly, the component approach; what it means and how to design the system accordingly within the frame of the available language. The overall design of the system had changed as we gained better understanding of how this can be applied to the available toolset and, more importantly, as our focus shifted from considering components as reusable building objects to emphasizing their replaceability; as explained in the case study. The final design, although proven successful in verification, does point out some possible weaknesses with SaveCCM – such as the limited possibility to specify easily replaceable component interfaces using only ports (this is exemplified by the turntable tools interface).

Once the system design was starting to take form, the focus shifted towards the interaction between the system and the environment including the verification properties. The strengths of each of the two behavioral modeling options (the SaveCCM TA and the UPPAAL TA)

---

21 Areas that took more work than other areas
weren’t apparent from the get go because of what, at first – before proper knowledge was gained, appeared to be subtle differences were in fact most important in modeling the control system and the environment respectively.

Finally, some of the parts of the toolset were still in development during this thesis so considerable effort was put into toolset integration, model-linking and providing feedback to the developers. A sizeable contribution of this thesis is thus in providing a complete and verifiable model, designed within the toolset.

To evaluate the component technology and the formal verification tools we have opted to model a production system, previously implemented using other modeling languages and technologies. While the goal of this thesis was not to compare different technologies, this approach still allows interested users a somewhat easier way to compare the related component technologies with SaveCCT.
8 References


9 Appendix – Control System Model

This chapter shows the complete design of the Control System, including the parts previously explained in detail.

9.1 Architecture

Figure 35 - Control System, architecture
9.2 Turntable

Turntable TA Parameters:
slot0, slot1, slot2, slot3, trnStart, trnFinished,
changeslot0, changeslot1, changeslot2, changeslot3,
run0, run1, run2, run3, fin0, fin1, fin2, fin3

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entry Assignment</td>
<td>evaluate()</td>
</tr>
</tbody>
</table>

Table 8 - Turntable TA - location properties

clock GLOBALTIME;
const int S_START=2;
const int S_TURNING=1;
const int S_IDLE=0;
int[S_IDLE, S_START] mode=S_START;

const int SL_EMPTY=0;
const int SL_LOADED=1;
const int SL_DRILLED=2;
const int SL_TESTBAD=3;
const int SL_TESTGOOD=4;

int $slot0=SL_EMPTY;
int $slot1=SL_EMPTY;
int $slot2=SL_EMPTY;
int $slot3=SL_EMPTY;

int $changeslot0, $changeslot1, $changeslot2, $changeslot3;
bool $run0, $run1, $run2, $run3;
bool $fin0, $fin1, $fin2, $fin3;
bool $trnStart, $trnFinished;
```c
void rotateSlots() {
    int temp = $slot0;
    $slot0 = $slot3;
    $slot3 = $slot2;
    $slot2 = $slot1;
    $slot1 = temp;
}

void evaluate() {
    if (mode==S_START) {
        rotateSlots();
        $trnStart=true;
        mode=S_TURNING;
    }
    else if (mode==S_TURNING) {
        $trnStart=false;
        if (!$trnFinished) {
            $run0 = true;
            $run1 = true;
            $run2 = true;
            $run3 = true;
            mode=S_IDLE;
        }
    }
    else if (mode==S_IDLE) {
        $run0 = false;
        $run1 = false;
        $run2 = false;
        $run3 = false;
        if (!$fin0 and $fin1 and $fin2 and $fin3) {
            $slot0 = $changeslot0;
            $slot1 = $changeslot1;
            $slot2 = $changeslot2;
            $slot3 = $changeslot3;
            rotateSlots();
            $trnStart=true;
            mode=S_TURNING;
        }
    }
}
```

Listing 11 - Turntable TA - declarations

```xml
<MAP var="slot0" port="S0_Status"/>
<MAP var="slot1" port="S1_Status"/>
<MAP var="slot2" port="S2_Status"/>
<MAP var="slot3" port="S3_Status"/>
<MAP var="trnStart" port="aRotate"/>
<MAP var="trnFinished" port="sCompleted"/>
<MAP var="changeslot0" port="S0_Result"/>
<MAP var="changeslot1" port="S1_Result"/>
<MAP var="changeslot2" port="S2_Result"/>
<MAP var="changeslot3" port="S3_Result"/>
<MAP var="run0" port="S0_Start"/>
<MAP var="run1" port="S1_Start"/>
<MAP var="run2" port="S2_Start"/>
<MAP var="run3" port="S3_Start"/>
<MAP var="fin0" port="S0_Finished"/>
<MAP var="fin1" port="S1_Finished"/>
<MAP var="fin2" port="S2_Finished"/>
<MAP var="fin3" port="S3_Finished"/>
</MAPPING>
```

Listing 12 - Turntable TA - mapping file
9.3 Loader

Loader TA parameters:
run, finished, slotstatus, slotresult

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Entry</strong></td>
<td></td>
</tr>
<tr>
<td>Entry Assignment</td>
<td>evaluate()</td>
</tr>
<tr>
<td><strong>Loading</strong></td>
<td></td>
</tr>
<tr>
<td>Invariant</td>
<td></td>
</tr>
<tr>
<td><strong>Exit</strong></td>
<td></td>
</tr>
<tr>
<td>Exit Assignment</td>
<td></td>
</tr>
</tbody>
</table>

Table 9 - Loader TA - location properties

```c
const int S_READY=0;
const int S_LOADING=1;
int [S_READY, S_LOADING] mode=S_READY;
bool $run, $finished;
int $slotstatus, $slotresult;
const int SL_EMPTY=0;
const int SL_LOADED=1;
const int SL_DRILLED=2;
const int SL_TESTBAD=3;
const int SL_TESTGOOD=4;
void evaluate() {
    if (mode==S_READY) {
        if ($run) {
            $slotresult=$slotstatus;
            if ($slotstatus==SL_EMPTY) {
                $finished=false;
                mode=S_LOADING;
            } else {
                $finished=true;
                mode=S_READY;
            }
        }
    }
}
```

Figure 37 - Loader TA
Listing 13 - Loader TA - declarations

```c
else if (mode==S_LOADING)
{
    $slotresult=SL_LOADED;
    $finished=true;
    mode=S_READY;
}
```

Listing 14 - Loader TA - mapping file

```xml
<MAPPING xstamodule="loader.xsta">
    <MAP var="run" port="Start"/>
    <MAP var="finished" port="Finished"/>
    <MAP var="slotstatus" port="Input"/>
    <MAP var="slotresult" port="Output"/>
</MAPPING>
```
9.4 Driller

Driller TA parameters:
run, finished, slotstatus, slotresult, lock, isLocked, isUnlocked, aDrillOnOff, aDrillMoveDown, aDrillMoveUp, sDrillDown, sDrillUp

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Entry</strong></td>
<td></td>
</tr>
<tr>
<td>Entry Assignment</td>
<td>evaluate()</td>
</tr>
<tr>
<td><strong>Clamping</strong></td>
<td></td>
</tr>
<tr>
<td>Invariant</td>
<td></td>
</tr>
<tr>
<td><strong>DrillingDown</strong></td>
<td></td>
</tr>
<tr>
<td>Invariant</td>
<td></td>
</tr>
<tr>
<td><strong>DrillingUp</strong></td>
<td></td>
</tr>
<tr>
<td>Invariant</td>
<td></td>
</tr>
<tr>
<td><strong>Unclamping</strong></td>
<td></td>
</tr>
<tr>
<td>Invariant</td>
<td></td>
</tr>
<tr>
<td><strong>Exit</strong></td>
<td></td>
</tr>
<tr>
<td>Exit Assignment</td>
<td></td>
</tr>
</tbody>
</table>

Table 10 - Driller TA - location properties
const int S_READY=0;
const int S_CLAMPING=1;
const int S_DRILLING_DOWN=2;
const int S_DRILLING_UP=3;
const int S_UNCLAMPING=4;

int[S_READY, S_UNCLAMPING] mode=S_READY;

bool $lock, $isLocked, $isUnlocked;
bool $aDrillOnOff, $aDrillMoveDown, $aDrillMoveUp;
bool $sDrillDown, $sDrillUp;
bool $run, $finished;
int $slotstatus, $slotresult;

const int SL_EMPTY=0;
const int SL_LOADED=1;
const int SL_DRILLED=2;
const int SL_TESTBAD=3;
const int SL_TESTGOOD=4;

void evaluate() {
    if (mode==S_READY) {
        if ($run) {
            $slotresult=$slotstatus;
            if ($slotstatus!=SL_EMPTY) {
                $finished=false;
                $lock=true;
                mode=S_CLAMPING;
            } else {
                $finished=true;
                mode=S_READY;
            }
        }
    } else if (mode==S_CLAMPING) {
        if ($isLocked) {
            $aDrillOnOff = true;
            $aDrillMoveDown= true;
            mode=S_DRILLING_DOWN;
        }
    } else if (mode==S_DRILLING_DOWN) {
        if ($sDrillDown) {
            $aDrillMoveDown= false;
            $aDrillMoveUp = true;
            mode=S_DRILLING_UP;
        }
    } else if (mode==S_DRILLING_UP) {
        if ($sDrillUp) {
            $aDrillOnOff = true;
            $aDrillMoveUp = false;
            $lock=false;
            mode=S_UNCLAMPING;
        }
    } else if (mode==S_UNCLAMPING) {
        if ( $isUnlocked) {
            $finished=true;
            $slotresult=SL_DRILLED;
            mode=S_READY;
        }
    }
}

Listing 15 - Driller TA – declarations
Listing 16 - Driller TA - mapping file

### 9.5 Tester

Tester TA parameters:
moveUp, moveDown, active, sensorTop, sensorDown, finished, slotstatus, slotresult

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Entry</strong></td>
<td></td>
</tr>
<tr>
<td>Entry Assignment</td>
<td>evaluate()</td>
</tr>
<tr>
<td><strong>Moving Up</strong></td>
<td></td>
</tr>
<tr>
<td>Invariant</td>
<td></td>
</tr>
<tr>
<td><strong>Moving Down</strong></td>
<td></td>
</tr>
<tr>
<td>Invariant</td>
<td></td>
</tr>
<tr>
<td><strong>Exit</strong></td>
<td></td>
</tr>
<tr>
<td>Exit Assignment</td>
<td>doSetupOutPort()</td>
</tr>
</tbody>
</table>

Table 11 - Tester TA - location properties
const int S_READY=0;
const int S_MOVING_DOWN=1;
const int S_MOVING_UP=2;
int[S_READY, S_MOVING_UP] mode=S_READY;

const int SL_EMPTY=0;
const int SL_LOADED=1;
const int SL_DRILLED=2;
const int SL_TESTBAD=3;
const int SL_TESTGOOD=4;

bool $sensorTop, $sensorDown;
bool $moveUp, $moveDown;
bool $active, $finished;
int $slotstatus, $slotresult;
int ticks;

void evaluate() {
    if (mode==S_READY) {
        if($active) {
            $slotresult=$slotstatus;
            if ($slotstatus==SL_DRILLED) {
                $finished=false;
                mode=S_MOVING_DOWN;
                ticks=0;
            }
            else {
                $finished=true;
                mode=S_READY;
            }
        }
    }
    else if (mode==S_MOVING_DOWN) {
        ticks++;
        if (ticks<=2) {
            if ($sensorDown) {
                $slotresult=SL_TESTGOOD;
                mode=S_MOVING_UP;
            }
            else {
                $slotresult=SL_TESTBAD;
                mode=S_MOVING_UP;
            }
        }
    }
    else if (mode==S_MOVING_UP) {
        if ($sensorTop) {
            $finished=true;
            mode=S_READY;
        }
    }
}

void dosetupOutPorts() {
    $moveDown=mode==S_MOVING_DOWN;
    $moveUp=mode==S_MOVING_UP;
}
Listing 18 - Tester TA - mapping file

9.6 Unloader

Unloader TA parameters:
active, finished, slotstatus, slotresult

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entry</td>
<td></td>
</tr>
<tr>
<td>Entry Assignment</td>
<td>evaluate()</td>
</tr>
<tr>
<td>Unloading</td>
<td></td>
</tr>
<tr>
<td>Exit</td>
<td></td>
</tr>
<tr>
<td>Exit Assignment</td>
<td></td>
</tr>
</tbody>
</table>

Table 12 - Unloader TA - location properties

```c
const int S_READY=0;
const int S_UNLOADING=1;
int[S_READY, S_UNLOADING] mode=S_READY;

bool $active, $finished;
int $slotstatus, $slotresult;

const int SL_EMPTY=0;
const int SL_LOADED=1;
const int SL_DRILLED=2;
const int SL_TESTBAD=3;
const int SL_TESTGOOD=4;

void evaluate() {
```
if (mode==S_READY) {
    if ($active) {
        $slotresult=$slotstatus;
        if ($slotstatus==SL_TESTGOOD) {
            $finished=false;
            mode=S_UNLOADING;
        } else {
            $finished=true;
            mode=S_READY;
        }
    }
} else if (mode==S_UNLOADING) {
    $slotresult=SL_EMPTY;
    $finished=true;
    mode=S_READY;
}

Listing 19 - Unloader TA – declarations

Listing 20 - Unloader TA - mapping file
10 Appendix – Environment Model

In this appendix chapter, we present the complete environment model – this includes the parts that were presented in detail in the main part of the thesis.

10.1 Global Declarations

```c
#include <stdio.h>

int main() {
    // Global declarations
    clock GlobalClock;
    /**  
     * TURNTABLE 
     */
    bool aRotate = false; // Start rotation
    bool sCompleted = false; // Completed rotation

    /**  
     * CLAMP 
     */
    bool aClamp = false;
    bool sLocked = false;
    bool sUnlocked = true;

    /**  
     * DRILL 
     */
    bool aDrillOnOff = false;
    bool aDrillMoveDown = false;
    bool aDrillMoveUp = false;

    /**  
     * DRILL TEST 
     */
    int dtStatus = 0;
    int dtResult = 0;
    bool dtStart = 0;
    bool dtFinished = 0;

    /**  
     * TESTER 
     */
    bool aTesterMoveDown = false;
    bool aTesterMoveUp = false;

    // Output
    bool sDrillDown = false;
    bool sDrillUp = true;
    bool sTesterUp = true;
    bool sTesterDown = false;
}
```

Listing 21 – Environment model: Global declarations
10.2 TesterSim

**Figure 41 – Environment model: TesterSim TA**

```c
int MIN_TIME = 1;
int MAX_TIME = 2;
int UP_TIME = 1;
int BREAK_TIME = 2;
```

**Listing 22 – Environment model: TesterSim TA declarations**
10.3 TurntableSim

Figure 42 - Environment model: TurntableSim TA

```
clock trnClock;
const int TURN_TIME=2;
```

Listing 23 - Environment model: TurntableSim TA declarations
10.4 ClampSim

Figure 43 - Environment model: ClampSim TA

Listing 24 - Environment model: ClampSim TA declarations

clock claClock;
const int CLAMP_TIME=1;
10.5 Driller_P1

This Timed Automaton has no local declarations.
10.6 Driller_P2

Figure 45 - Environment model: Driller_P2 TA

Listing 25 - Environment model: Driller_P2 TA declarations
10.7 *DrillerTest*

![Timed Automaton Diagram](image)

**Figure 46 - Environment model: DrillerTest TA**

This Timed Automaton has no local variables.
11 Appendix – Properties (turntable.q)

This appendix chapter lists all properties that have been checked. The properties marked with red are not satisfied – this is intentional.

// SL_EMPTY = 0;
// SL_LOADED = 1;
// SL_DRILLED = 2;
// SL_TESTBAD = 3;
// SL_TESTGOOD = 4;
// C1 = Turntable
// C2 = Loader
// C3 = Tester
// C4 = Unloader
// C5 = Driller
// C6 = Clock

Property 1:

// -------------------------------
// No deadlock?
A[] not deadlock
// -------------------------------

Properties 2, 3, 4, 5:

// -------------------------------
// (5) no drilling, testing or unloading is attempted from an empty slot
// No loading takes if the slot is already occupied
// -------------------------------
A[] C2.Input#43!=0 imply not C2.Loading
A[] C3.Input#55==0 imply not (C3.MovingUp or C3.MovingDown)
A[] C4.Input#62==0 imply not (C4.Unloading)
A[] C5.Input#68==0 imply not (C5.Clamping or C5.DrillingDown or C5.Unclamping)

Properties 6, 7, 8, 9:

// -------------------------------
// If any of the tools has received a start signal, it eventually signals
// that it has finished working
C2.Start#42 --> C2.Finished#44
C3.Start#51 --> C3.Finished#54
C4.Start#64 --> C4.Finished#65
C5.Start#70 --> C5.Finished#71
// -------------------------------

Properties 10, 11, 12:

// -------------------------------
// Either turning or working, but not both
// Idea: For C1-C5, mode=0 means IDLE. So either both parentheses must be
// 0 or at least on of them must be 0 (IDLE)...
A[] ((C1.mode)*(C2.mode+C3.mode+C4.mode+C5.mode))==0
A[] (C2.mode==0 or C3.mode==0 or C4.mode==0 or C5.mode==0) imply C1.mode==0
A[] C1.mode!=0 imply (C2.mode==0 and C3.mode==0 or C4.mode==0 or C5.mode==0)
// -------------------------------
Properties 13, 14, 15:

```
// Correspondence internal state <-> external model
// If C1 (Turntable) is in idle mode, the environment mode should not be
// in a "turning" state
// Will report NOT SATISFIED, if TURN_TIME=1 (in TurntableSim UPPAAL model)
// For the C5 query, the first one will not be satisfied. This is because
// the Driller_P1 is very simple and the model does not wait for the
// drilling to stop. However, the second query is more "correct". It does
// not matter if the driller is still on as long as it is in the UP position
A[] C1.mode==0 imply not TurntableSim.turning
A[] C5.mode==0 imply not (ClampSim.Locking or ClampSim.Locked or
ClampSim.Unlocking or (Driller_P1.Driller_ON and Driller_P2.Driller_DOWN) or
Driller_P2.Driller_DOWN)
A[] C5.mode==0 imply not (ClampSim.Locking or ClampSim.Locked or
ClampSim.Unlocking or Driller_P1.Driller_ON or Driller_P2.Driller_DOWN)
```

Property 16:

```
// Empty slot, leads to loaded slot
C1.S0_Status#23==0 --> C1.S0_Result#30==1
```

Properties 17, 18, 19, 20, 21, 22:

```
// Products are drilled, if and only if there is a product in the slot
E<> C1.S1_Status#24==1
E<> C1.S1_Result#32==2
C1.S1_Status#24==1 --> C1.S1_Result#32==2
E<> DrillerTest.working
A[] not DrillerTest.error
A[] Driller_P2.Driller_DOWN imply ClampSim.Locked
```

Property 23:

```
// Products are tested and the results are given
C1.S2_Status#25==2 --> (C1.S2_Result#31==3 or C1.S2_Result#31==4)
```

Properties 24, 25:

```
// No conflicting orders are sent to the environment
A[] not (C3.aTesterMoveUp#49 and C3.aTesterMoveDown#50)
A[] not (C5.aDrillMoveDown#77 and C5.aDrillMoveUp#80)
```
Appendix – Properties (turntable.q)

Property 26:

// Good products are removed from the system
C1.S3_Status#26==4 -> C1.S3_Result#33==0

Property 27:

// Bad products are not removed from the system
// Idea: If the slot status is Bad Test, the C4 (Unloader) never goes through the unloading state

Property 28, 29, 30, 31:

// The system keeps on going
// Idea: check that the turntable keeps on alternating between the idle and turning state
// This is a stronger verification than NO DEADLOCK (which only checks for deadlocks in the model, not in the model behaviour)
E<> C1.mode==1
E<> C1.mode==0
C1.mode==1 -> C1.mode==0
C1.mode==0 -> C1.mode==1

Property 32:

// This query will not be satisfied; this is because the component will not be executing at the same time and the state of the SaveCCM TA is not remembered after the triggered execution has finished. To test for this type of property, an internal component variable (such as mode) must be used instead of the state name.
E<> (C5.Clamping and C2.Loading)

12 Appendix - Verification results (std output only)

This appendix chapter shows the command line used to execute the batch verification. The property numbers match the numbers in the previous chapter.

Notice that the error output is redirected to verifyta_error.txt file. This file contains traces for E<> type of properties as well as the properties that have not been satisfied. Using the specified set of properties, this file is almost 7MBs in size and is thus not presented here.

```
verifyta.exe -C -s -u -S0 -n0 -t1 -Y "!! LOAD ME !!.save" Turntable.q
2>verifyta_error.txt
```

Options for the verification:
- Generating shortest trace
- Search order is breadth first
- Using no space optimisation
- Seed is 1211554146
- State space representation uses difference bound matrices

Verifying property 1 at line 21
-- Property is satisfied.
-- States stored : 38166 states
-- States explored : 0 states

Verifying property 2 at line 35
-- Property is satisfied.
-- States stored : 38166 states
-- States explored : 0 states

Verifying property 3 at line 36
-- Property is satisfied.
-- States stored : 38166 states
-- States explored : 0 states

Verifying property 4 at line 37
-- Property is satisfied.
-- States stored : 38166 states
-- States explored : 0 states

Verifying property 5 at line 38
-- Property is satisfied.
-- States stored : 38166 states
-- States explored : 0 states

Verifying property 6 at line 48
-- Property is satisfied.
-- States stored : 38166 states
-- States explored : 0 states

Verifying property 7 at line 49
-- Property is satisfied.
-- States stored : 38166 states
-- States explored : 0 states

Verifying property 8 at line 50
-- Property is satisfied.
-- States stored : 38166 states
-- States explored : 0 states

Verifying property 9 at line 51
-- Property is satisfied.
-- States stored : 38166 states
-- States explored : 0 states

Verifying property 10 at line 63
-- Property is satisfied.
-- States stored : 38166 states
-- States explored : 0 states

Verifying property 11 at line 64
-- Property is satisfied.
-- States stored : 38166 states
-- States explored : 0 states

Verifying property 12 at line 65
-- Property is satisfied.
-- States stored : 38166 states
-- States explored : 0 states
Verifying property 13 at line 81
--- Property is satisfied.
-- States stored: 38166 states
-- States explored: 0 states

Verifying property 14 at line 82
--- Property is satisfied.
-- States stored: 38166 states
-- States explored: 0 states

Verifying property 15 at line 83
--- Property is NOT satisfied.
-- States stored: 1194 states
-- States explored: 0 states
Showing counter example.

Verifying property 16 at line 92
--- Property is satisfied.
-- States stored: 38166 states
-- States explored: 0 states

Verifying property 17 at line 102
--- Property is satisfied.
-- States stored: 243 states
-- States explored: 0 states
Showing example trace.

Verifying property 18 at line 103
--- Property is satisfied.
-- States stored: 1271 states
-- States explored: 0 states
Showing example trace.

Verifying property 19 at line 104
--- Property is satisfied.
-- States stored: 38166 states
-- States explored: 0 states

Verifying property 20 at line 106
--- Property is satisfied.
-- States stored: 523 states
-- States explored: 0 states
Showing example trace.

Verifying property 21 at line 108
--- Property is satisfied.
-- States stored: 38166 states
-- States explored: 0 states

Verifying property 22 at line 109
--- Property is satisfied.
-- States stored: 38166 states
-- States explored: 0 states

Verifying property 23 at line 119
--- Property is satisfied.
-- States stored: 38166 states
-- States explored: 0 states

Verifying property 24 at line 129
--- Property is satisfied.
-- States stored: 38166 states
-- States explored: 0 states

Verifying property 25 at line 130
--- Property is satisfied.
-- States stored: 38166 states
-- States explored: 0 states

Verifying property 26 at line 141
--- Property is satisfied.
-- States stored: 38166 states
-- States explored: 0 states

Verifying property 27 at line 152
--- Property is satisfied.
-- States stored: 38166 states
-- States explored: 0 states

Verifying property 28 at line 167
--- Property is satisfied.
-- States stored: 6 states
-- States explored: 0 states
Showing example trace.
Verifying property 29 at line 168
-- Property is satisfied.
-- States stored : 110 states
-- States explored : 0 states
Showing example trace.

Verifying property 30 at line 169
-- Property is satisfied.
-- States stored : 38166 states
-- States explored : 0 states

Verifying property 31 at line 170
-- Property is satisfied.
-- States stored : 38166 states
-- States explored : 0 states

Verifying property 32 at line 183
-- Property is NOT satisfied.
-- States stored : 38166 states
-- States explored : 0 states