Capturing Assumptions while Designing a Verification Model for Embedded Systems

J. Marincic[1], A. Mader[2], and R. Wieringa

Department of Computer Science, University of Twente, The Netherlands,
P.O.Box 217, 7500 AE Enschede, The Netherlands
email:{j.marincic, mader, roelw}@ewi.utwente.nl

January, 2007.

[1]Supported by the Netherlands Organisation for Scientific Research (NWO), project 600 065 120 241420, Modelling Control Aspects of Embedded Systems
[2]Supported by the Netherlands Organisation for Scientific Research (NWO), project 632 001202, Methods of Modelling Embedded Systems

Abstract

A formal proof of a system correctness typically holds under a number of assumptions. Leaving them implicit raises the chance of using the system in a context that violates some assumptions, which in return may invalidate the correctness proof. The goal of this paper is to show how combining informal and formal techniques in the process of modelling and formal verification helps capturing these assumptions. As we focus on embedded systems, the assumptions are about the control software, the system on which the software is running and the system's environment. We present them as a list written in natural language that supplements the formally verified embedded system model. These two together are a better argument for system correctness than each of these given separately.

Contents

1 Introduction

Formal verification is a way to prove that certain properties are true or false of a model of a system. The proof tells us something about the system, only if the model is an adequate representation of the system. Figure 1 describes activities that must be performed when designing a verification model [15].

In order to show that a system behaves as expected we do the following steps. We start with a system and accompanying documentation, blueprints and stakeholders wishes. We have to find out what the desired behaviour of the system is, i.e. the system requirements that we are going to verify. We then model (or we can say, we describe) the system with aim to include the system parts, aspects and behaviours that influence these requirements. This is represented with a left vertical arrow in the diagram in Fig.1. Requirements are also formally described, as the right vertical arrow in the same diagram shows.

We want to know whether our *system* behaves as required. However, we prove a formalization of the system requirements against the system *model* (upper horizontal arrow in Fig.1). The question now is how much confidence we gained in the behaviour of the system (lower horizontal arrow in Fig.1), if the proof performed was successful. Part of the answer to this question is determined by which assumptions we made about the system and its environment in our formalization.

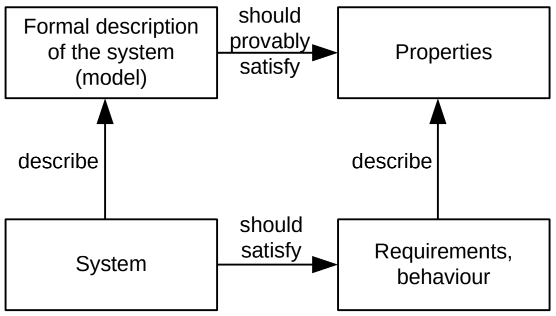


Figure 1: The model is a formal description of the system. It can be formally verified or checked it has the formalized properties.

While building the system model, we make various assumptions. These are the assumptions about the parts of the system, about its behaviour and its environment. Examples of such assumptions are: "We assume that the motors work properly", or "The maximum velocity of the launcher is smaller than the maximum value that can be stored in the register responsible for storing its value". Usually, these assumptions are left implicit. Some people think that they are too obvious to be explicitly stated, or they simply forget about them because they do not document them in the moment when they were made. Some of them are made subconsciously. But, it is important to be aware of the following: *Our formal model that is verified for the desired properties comes* *together with certain assumptions about the system. If the assumptions are false, we cannot guarantee that the system will do what we proved.*

In this paper the focus will be on capturing and documenting the assumptions during design of the verification model. We start from the hypothesis that designing a formal model directly from nonformal descriptions leaves out a lot of important decisions and assumptions implicit which then leaves the space for errors and incorrectness. We will show how combining informal and formal methods helps making the assumptions explicit.

In the next section, Sect.2, we will present the basic concepts from which we start. Sect.3 introduces our illustrating example. In Sect.4 we will show what steps we followed in the system modelling and will elaborate them on the example. We will compare our work with related work in Sect.5. Discussion and the future work, as well as the conclusion, are part of Sect.6.

2 Terminology and Basic Concepts

An **embedded system** consists of a controller and a controlled, physical part. The controller itself consists of the control hardware, an operating system and the control software. The control hardware can be a PC, but in most cases it is a special purpose computer or a microprocessor embedded in a device, machine or a plant. Note that, in contrast to our terminology, sometimes the processor board, its hardware and the software running on it are referred to as an embedded system.

Further on, we will use the term **system** for the embedded system. We will use the term **plant** for the controlled, physical part, and reserve the term **environment** for everything outside the embedded system. Figure 2 shows an embedded system and its parts, as we view them.

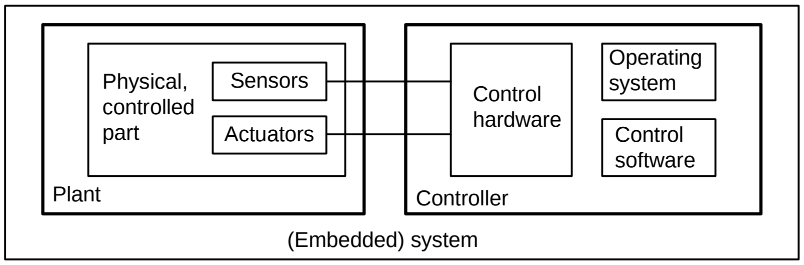


Figure 2: Embedded system

We define **assumptions** as propositions about the control, plant or the environment that must be true for a correctness property to hold. For example, “If the temperature is between -10 and +50 °C”, (assumption) “then the device works as specified”(correctness property).

Our approach is to focus on *both* the controller and the plant, as only their interaction can produce the required behaviour of the embedded system. Therefore we provide models for both the plant and the control and verify their composition against the required behaviour. This is different to other approaches of embedded system verification, where the control software is modelled and the plant and its behaviour are represented with a number of assumptions. In those approaches, events in the plant are usually not related (even if this is the case in reality), but they are represented as nondeterministic events that may occur at the control interface. The focus in those approaches is on software verification.

Our reasons to also model the plant are: (1) Requirements are about the embedded system as a whole (not about the control only) and, (2) effects of control faults are observable in the plant behaviour. Having a plant model helps to trace back observable behaviour of the plant to control fragments responsible for this behaviour.

In our approach we (1) start from the requirements for the embedded system, (2) we model both the plant and the control and (3) we verify the composed model of the plant and the control. In all these activities, we will make assumptions about the control, plant and environment.

3 The Lego Sorter

We demonstrate our approach with a Lego sorter, a small PLC-controlled plant made of Lego bricks, DC motors, angle sensors and a colour scanner. The requirement is:

R_0: The Lego sorter should sort yellow and blue bricks according to their colour.

At this point it is not stated where the bricks will come from. This information will be added when we refine the requirement later on.

Figure 3 shows a top view of the plant and Fig.4 a side view (a short film showing the plant at work is available on the project website [1]).

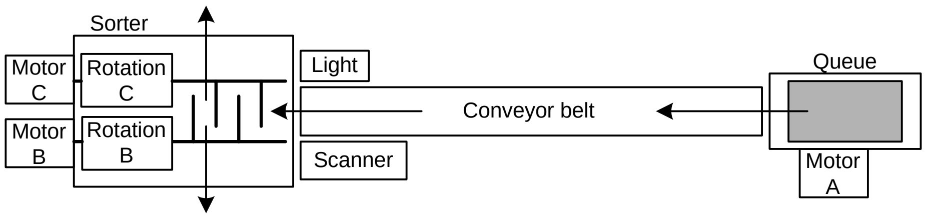


Figure 3: Top view on the Lego plant

Bricks are stored in a queue. Two wheels at the bottom of the queue move the bottom brick to the conveyor belt. Bricks are transported by the conveyor belt to the scanner and further on to the sorter. The scanner senses the colour of a brick. The sorter consists of two fork-like arms. Each arm can rotate a brick to one of the sides of the plant.

Bricks enter the belt one after another and it is possible to have more than one brick on the belt.

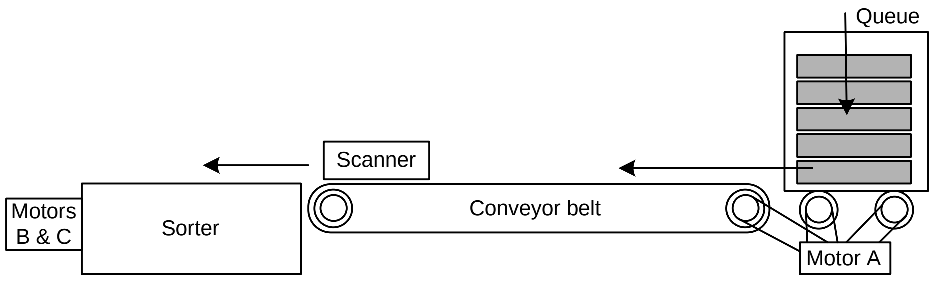


Figure 4: Side view on the Lego plant

The wheels of the queue and belt are coupled - this means that one motor is moving them. The scanner is a sensor that can distinguish a yellow, blue or no brick in front of it. Putting a brick of another colour in front of it would cause the scanner to enter into an unknown state. Each sorter arm is controlled by its own motor and has its own rotation sensor that senses the angle of the arm. The starting angle is 0, and as the arm rotates it changes to 360 degrees.

4 Modelling Steps

In our example, the plant and the technical documentation of its parts are given to us, as well as a natural language specification of the control software. We have to verify system correctness, i.e. that an embedded system consisting of the plant and a controller that implements the specification, satisfies the requirement R_0. The list below contains the steps that we performed while modelling and verifying the system, noting relevant assumptions along the way.

(1) *Specification of the requirement.* Part of the modelling process is to formalise the intuitive understanding of the desired behaviour as formal properties. We have to describe the initial requirement informally, as good as possible.

(2) *Definition of a fragment to be modelled.* First of all, the desired requirement possibly does not concern the overall embedded system, but is realised by its fragment. Furthermore, not all aspects of a systems are relevant to the requirement. Finally, we abstract from some parts as, for example the operating system, as we assume that it works properly.

(3) *Decomposition of the plant and of the requirements.* Depending on the requirements for the plant behaviour, we identify its relevant structures. We can, for example, identify the plant through the processes realised in it. Another way is to decompose to physical parts (or, as we call them,

instruments). The structure that we identify (processes or instruments or something else) implies a decomposition of the system. When decomposing a system, we simultaneously decompose the requirement, where each sub-requirement should be satisfied by a system-component, and all sub-requirements together should imply the original requirement.

(4) *Description of the relevant behaviors of the plant.* For each of the components identified in the previous step we need a description of its behaviour.

(5) *Definition of the representation of the plant within the controller.* The control observes the plant through sensors and change its behaviour via actuators. Additionally, a history of data may be necessary to reconstruct the relevant part of the real world in the control.

(6) *Description of the control behaviour.* In the context of our example we assume that control specification is given.

(7) *Formal description of the system.* We formally describe both the plant and the control behaviour. As we have chosen a model checking technique, this means that we describe the system with automata. We also describe formally the properties with a property specification language, as required by the model checker used (in the model checker we chose it was a temporal logic).

(8) *Model verification.* There are two things here. First we perform experiments with the model to test its adequacy. Then we check if the property holds for the model. Proving a property in case of model checking means checking whether a certain state in the automata is reachable and possibly whether a certain variable will have a certain value once that the state is reached.

Although we present steps in a certain order, some steps revealed errors in previous ones, so in fact this has been an iterative process. In each step we also describe the system behaviour in informal language.

Problem Frames Technique

For realization of the first six steps we also use ideas of the *problem frames* technique [10, 11] to classify the problem and to link the informal and formal descriptions. Framing is a way to structure a problem. In Jackson's approach, a problem frame is a collection of domains, where each domain is a coherent set of phenomena. Domains interfere through shared phenomena. The software to be developed is called *Machine* by Jacskon and it is one of the domains in the problem frame. The other domains are parts of the software environment, like for example parts of a plant. A *problem diagram* shows all domains of a problem, their interfaces and requirements to be satisfied. We give an example of a problem diagram later.

Verification Requirement

In earlier work, one of the authors developed a method for the derivation of a correctness theorem [12, 16, 9]. There, a verification requirement is stepwise extended with details about the system until a correctness theorem can be proved. A *Verification requirement* contains descriptions of the plant, the control and the system requirement. We formally state it as: $A \implies (P \wedge C \implies R)$, where P is the description of the plant behaviour, C is the description of the controller, R is the description of the requirement, and A are the assumptions that have to be satisfied. This method of proof is very similar to the reference model of requirements engineering [7], but it was developed independently. We have to prove formally the verification requirement in the formula.

Even if we do not write down instances of the verification requirement explicitly, it is possible to do so in every step. "In every modelling activity there is a mathematical theorem that we have in the back of our mind", shows [9]. The same holds for the verification requirement. We can show that, at every of the steps, we can state the verification requirement at the corresponding level of formality. This can be too time-consuming to do in practice, but it helps to be aware of this as we move from informal descriptions of the system to formal, mathematical description.

Statecharts

Statecharts are proposed to be used as unofficial language to be used by engineers describing real, reactive systems [8]. There are many proposed versions of their semantics [14]and one of them is the Statemate semantics of statecharts. The role of the statecharts in our method is to (1) describe behaviour of both plant and the control, (2) be the starting point for using some model-checking tool and to (3) contribute to the communication between formal verification expert and a software engineer.

Uppaal

Uppaal [2] is a tool for model checking of timed automata and we use it for the model checking. It is free and it is used in academical circles, as well as for industrial applications. Both statecharts and Uppaal are techniques that we chose for our example. However, in the steps that we propose, they are not the only possible choice.

In the remainder of this section we present the steps for modelling implemented on the Lego sorter.

4.1 Step (1) Specification of the Requirement

In the Lego sorter example the requirement for the plant is informally stated as

R_0: "Eventually, the Lego sorter will sort the yellow and blue bricks according to their colour".

4.2 Step (2) Defininition of a Fragment to Be Modelled

We will model the whole Lego plant because all of its parts are relevant for the requirement. But, not all the system aspects are relevant. We will not, for example, describe the speed of the belt, because the requirement does not say anything about the time to sort the bricks.

4.3 Step (3) Decomposition of the Plant and of the Requirements

Figure 5 documents the first decomposition step, where R_0 is a requirement and a is the description of the interface between the plant P and the controller C. The interface description a, describes the phenomena shared by the plant and the controller.

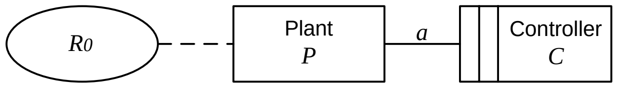


Figure 5: Problem diagram of the Lego plant and the Controller

Referring only to the domains modelled in the problem diagram, the verification requirement in its first version is:

$$P \wedge C \implies R_0.$$

In further steps we decompose the *Plant* to domains and we repeat the process until we get the components that are object to the observable behaviour addressed by the requirement and/or are objects of control observations and actions. Plant and control are decomposed separately here. We can identify physical parts and we define this decomposition as *instrumental decomposition.*

The second decomposition step of the system and the requirement is described in Fig.6

The requirement R_0 is refined to

R_1: "Eventually all the bricks from the queue will be moved by the sorter to the side corresponding to their colour".

R_1 is actually a conjunction of requirements about the queue and about the sorter, $R_1 = R_{11} \wedge R_{12}$, where

R_{11} = "Eventually all bricks of the Queue are gone" and

R_{12} = "Eventually all bricks are on the correct sides of the Sorter".

At this point we make the following assumptions:

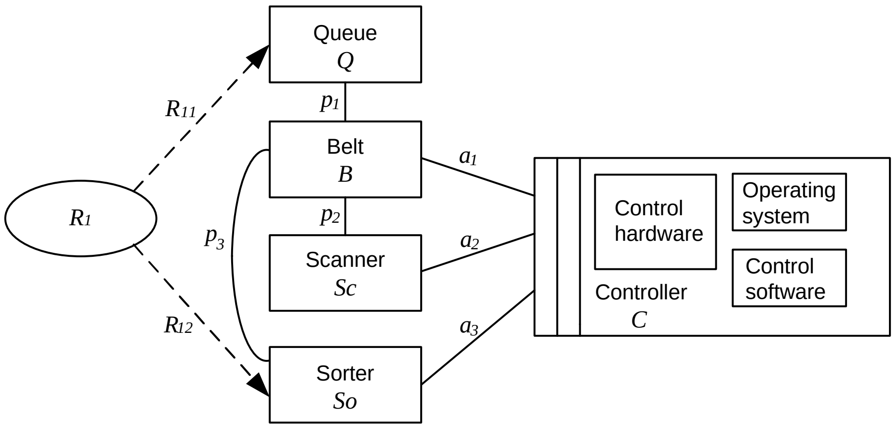


Figure 6: Refined problem diagram of the Lego plant and the Controller

A_1: We suppose that the computer hardware is working properly,

A_2: The operating system supports the software that we design,

A_3: The sampling period of the control is such that it can observe rotation angle with sufficient granularity

A_4: The PLC controller supports the desired sampling frequency

A_5: An operator will put the bricks in the queue.

According to the first four assumptions, we ignore both controller hardware and the operating system in the formal verification. Only the control software needs to be verified. The last assumption refers to the system environment and it expresses a requirement about a user.

Part of this step is the description of each domain and each interface. We describe the domains in natural language first. Also, problem diagrams are supported with a notation that allows us to describe interfaces as sets of phenomena. Sets are grouped by the interface they are part of, and the domain that can initiate or change the phenomena in the set. We found these descriptions useful because later on, the phenomena on the interfaces will be mapped on events in the statecharts.

We continue with the descriptions of each domain and interface.

Domains

Queue: The queue is a container that can hold bricks (of any color). If there are bricks in the queue, one brick rests on the wheels coupled with belt.

Belt: The Belt can move, thereby transporting a brick from the queue to the scanner and further on. The length of the belt and bricks is such that there can be a maximum of two bricks on the belt.

Scanner: The scanner can recognize the color of a brick that passes in front of it. It distinguishes blue and yellow bricks, and the absence of bricks.

If there is a brick of a different colour present in front of the scanner, it will recognize it as either a yellow or a blue brick. There is a minimal distance between the bricks, so that the scanner recognize them as separate.

SorterThe sorter can rotate its arms, but the control should ensure that they are not rotated both at the same time. Which arm is rotated depends on which motor is switched on.

Control software: The controller can turn on and off the belt and the sorter arms, it can observe the sorter arm angle and it receives data about the brick's presence and colour in front of the scanner. We are starting with the following natural language control specification.

The controller is continuously receiving one of the three possible values from the scanner (*blue, yellow* or *nothing*). The controller switches the belt on, as soon as there is no brick at the scanner. The controller switches the belt on if the sorter is idle. It switches the belt off, if there is a brick in the sorter and a brick at the scanner. The controller causes the sorter arms to move according to the colour of the brick in the sorter. After it receives a full rotation angle from the rotation sensor, it switches off the sorter arm.

Interfaces

Interface a_1: The controller switches the belt on and off.

Interface a_2: The scanner sends one out of the 3 values to the controller - blue, yellow or nothing.

Interface a_3: The sorter sends the angle positions of the arms to the controller; the controller switches each sorter arm on and off.

Interface p_1: The first brick (yellow or blue) in the queue moves to the belt, or nothing moves to the belt.

Interface p_2: The scanner detects whether there is either nothing or a yellow brick or a blue one at the scanner position.

Interface p_3: If a brick is at the end of the belt and the belt is moving (and the sorter is idle), then the brick moves to the Sorter.

In the notation of problem diagrams, we can describe this as:

a_1**:** *Controller!* {belt_start, belt_stop}
This means that controller can cause events belt_start and belt_stop. These will be events in the statecharts later.

a_2**:** *Sc!* {nothing_at_scanner, yellow_brick_at_scanner, blue_brick_at_scanner}

a_3**:**
Controller! {yellow_arm_start, yellow_arm_stop, blue_arm_start, blue_motor_stop}
So! {yellow_sorter_angle, blue_sorter_angle}

The sorter can cause the values of yellow_sorter_angle etc. to change. These will be variables in the statecharts later on.

p_1**:** *B!*{blue_brick_moves_to_belt, yellow_brick_moves_to_belt, nothing_moves_to_belt}

p_2: *B!* {blue_brick_is_at_scanner, yellow_brick_is_at_scanner, nothing_is_at_scanner}
p_3: *B!* {blue_brick_moves_to_sorter, yellow_brick_moves_to_sorter}

At this point there are more assumptions that we made:

A_6: The rotation sensors and motors that move the sorter arms and the belt are working properly and transmitting signals instantaneously.

A_7: Bricks are standard Lego bricks (50mm x 15mm x 7mm) that fit in the Queue.

A_8: The scanner observes the colour all the time (level-based, not edge-based)

A_9: There are only blue and yellow bricks in the Queue.

A_{10}: The plant has to be put on a flat horizontal surface, in order not to have gravity force moving bricks.

The consequence of the assumption A_6 is that we do not need to decompose further the domains on the plant side and to describe sensors and actuators. They are assumed to be perfect (i.e. no latency, no broken parts).

The verification requirement at this stage can be elaborated into:

$$A \implies (Q \land B \land Sc \land So \land C \implies \mathcal{R}_1),$$

where *A, Q, B, Sc, So* are formal descriptions of the assumptions, queue, belt, scanner and sorter respectively (yet to be found). Compared to the previous version of the verification requirement, we added assumptions and decomposed the plant into four domains. Note that in the verification requirement we do not have explicit interfaces. The composition is done by the logical $\land$, and all phenomena taking place on the interfaces should be part of the domain descriptions sharing the interface.

4.4 Step (4) Description of the relevant behaviors of the plant

The uncontrolled plant has the potential for a great variety of behaviour, most of which is undesired. The task of the control is to keep the plant within the desired behaviour. When we try to prove this property of the control, we have to consider the plant desired behaviour, but also undesired plant behaviour (and show that the plant will not do it). Usually we have hundreds of pages of technical documentation that contain informal descriptions of all behaviours of plant components. Modelling all undesired behaviour is an enormous amount of work and moreover, not necessary for our verification problem.

We do not consider the possibility that any part of the system can be broken or malfunctioning, like it is done in fault tolerant systems design. Considering malfunctioning devices in the plant would be part of another problem, to be modelled separately [11]. This means we only need to consider undesirable

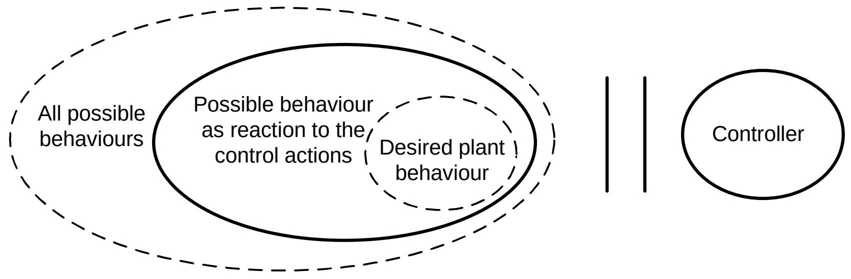


Figure 7: Diagram of the descriptions of the plant and control behaviour.

plant behaviour that results from actions by the control software - the only part of the controller we are not assuming to be correct (see also Fig.7).

In the Lego sorter example, the controller does not change the speed of the belt or sorter arms, it turns them on and off.

Queue

Figure 8 shows the state diagram describing the queue behaviour. It describes events of a brick entering out from the queue. A brick can be yellow or blue.

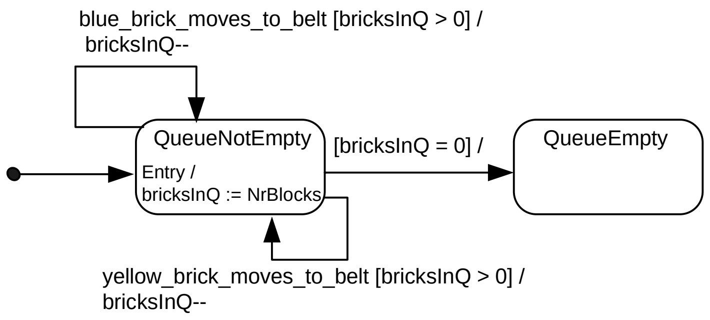

blue_brick_moves_to_belt [bricksInQ > 0] / bricksInQ--
QueueNotEmpty
Entry / bricksInQ := NrBlocks
[bricksInQ = 0] /
QueueEmpty
yellow_brick_moves_to_belt [bricksInQ > 0] / bricksInQ--

Figure 8: State diagram of the queue behaviour

Belt

Figures 9, 10, 11 and 12 are statecharts describing the belt behaviour. It can be seen that belt does the three different things simultanelously - it takes bricks from the queue, it delivers each brick to the scanner and it delivers it further to the sorter. The first state diagram shows the possibility of the belt being turned on and off at any time, which is what we define by 'all possible' behaviours. The other statescharts show its desirable behaviour.

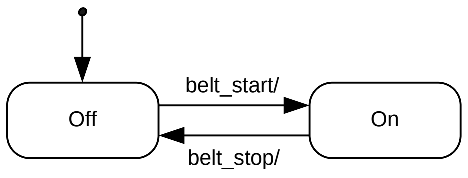
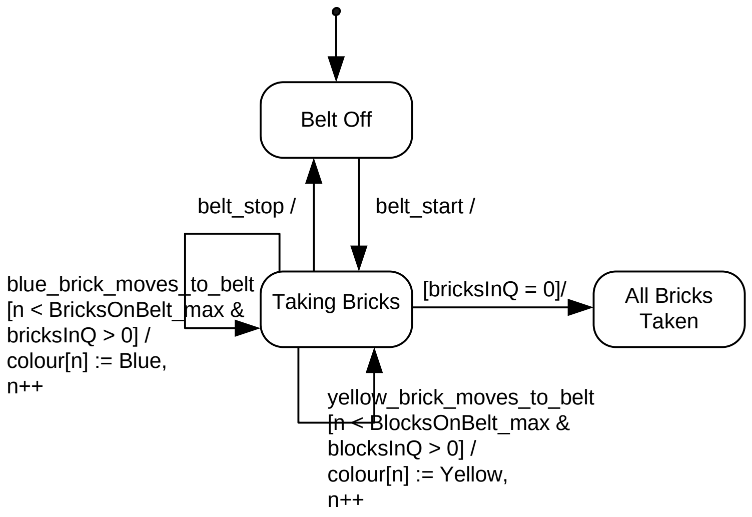


Figure 9: State diagram of the Belt (01)



Belt Off
belt_stop /
belt_start /
Taking Bricks
[bricksInQ = 0]/
All Bricks Taken
blue_brick_moves_to_belt [n < BricksOnBelt_max & bricksInQ > 0] / colour[n] := Blue, n++
yellow_brick_moves_to_belt [n < BlocksOnBelt_max & blocksInQ > 0] / colour[n] := Yellow, n++

Figure 10: State diagram of the Belt (02)

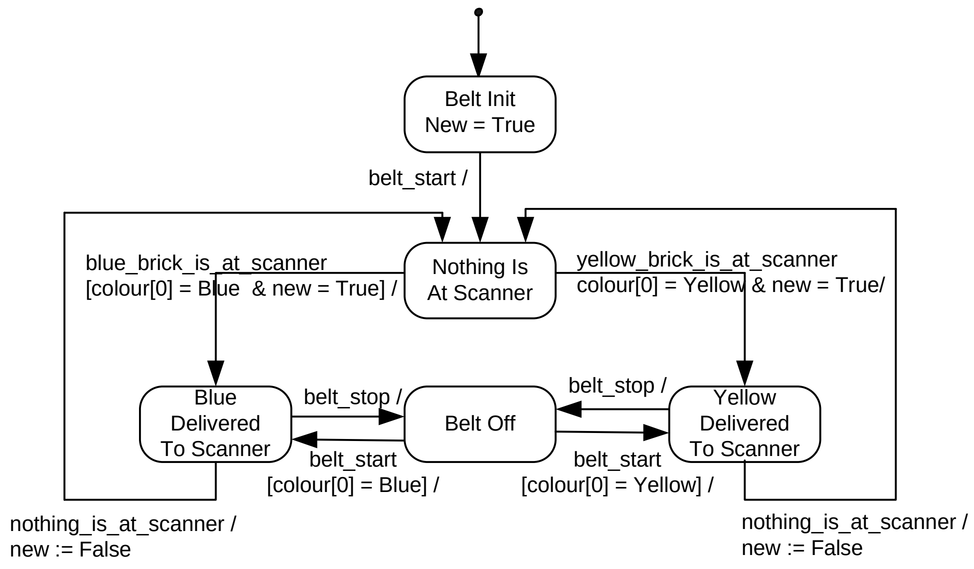


Figure 11: State diagram of the Belt (03)

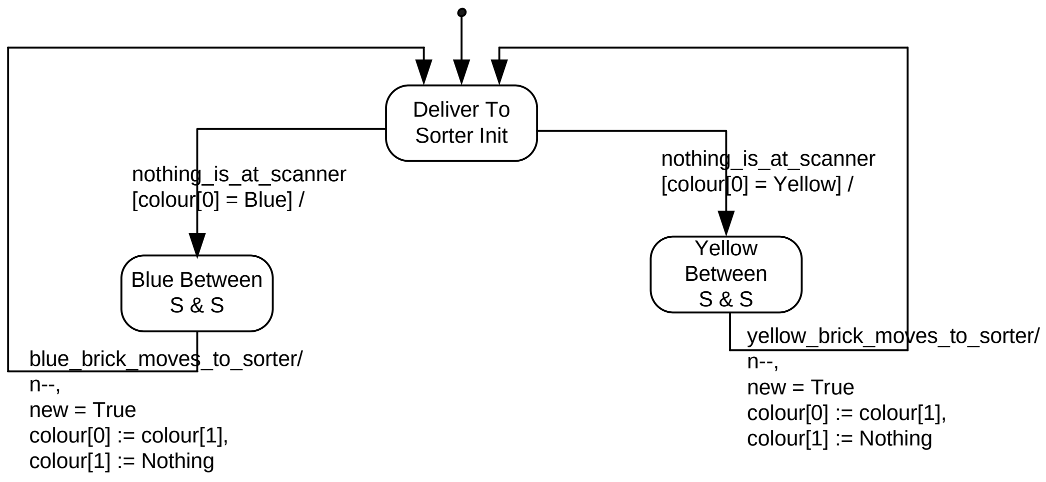


Figure 12: State diagram of the Belt (04)

Scanner

The state diagram in Fig. 13 shows the behaviour of the scanner - as soon as the leading edge of a yellow (blue) brick is in front of it (events blue_brick_is_at_scanner and yellow_brick_is_at_scanner), it will change the value that corresponds to yellow (blue) colour (the change is described by actions blue_brick_at_scanner and yellow_brick_at_scanner). This value is read by the controller. When there is nothing in front of the scanner, the value will correspond to this state (event nothing_is_at_scanner and action nothing_at_scanner).

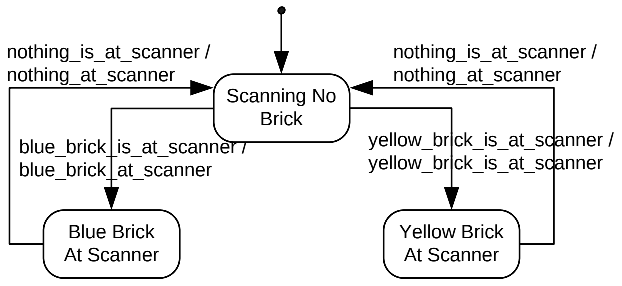


Figure 13: Scanner state diagram

Sorter

The behaviour of the sorter is described in Fig.14. This is the behaviour that an observer standing besides the plant can observe. We start with a sorter that is empty and neither of its arms are rotating. The belt brings a block to the sorter and later on, the controller turns on a corresponding arm. After it makes a full rotation, the controller turns off the arm. Starting one of the arms too early or rotating them both at the same time is an undesired behaviour described by the 'Wrong Sorting' state and events that leads to it.

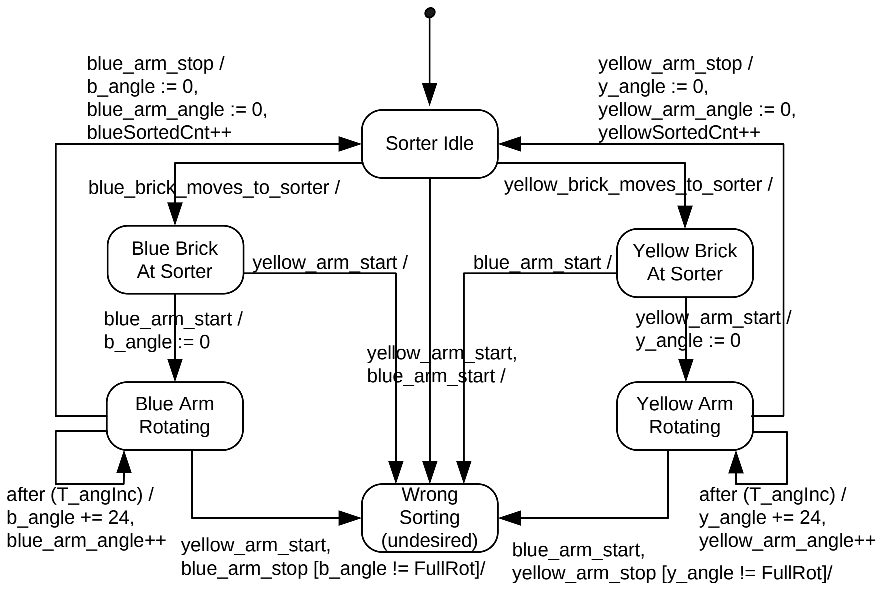


Figure 14: Sorter state diagram

4.5 Step (5) Definition of the representation of the plant within the controller

This step is intertwined with the next one, the control software description. Our controller will have to keep track of total number of bricks in front of the scanner and in the sorter and their colour. We did not represent a brick in the model, but when it arrives at the sorter, the information about its colour is not sensed any more by the scanner. The scanner is now sensing no brick or another brick, so it is necessary to store the colour in a variable.

4.6 Step (6) Description of Control Software Behaviour

In this step we implemented the control specification written in natural language in our model. There is no one possible translation of the natural language specification to the statecharts or any formal description. We chose to implement it to subcontrols - processes controlling parts of the plant (this is object oriented control decomposition). These subcontrols are not independent since they share data. Diagram in Fig.15 shows which processes share data.

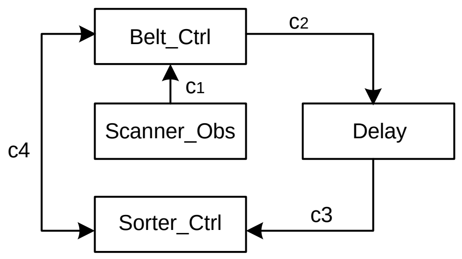


Figure 15: Control decomposed

State diagrams on figures 16, 17, 18 and 19 shows the controller behaviour.

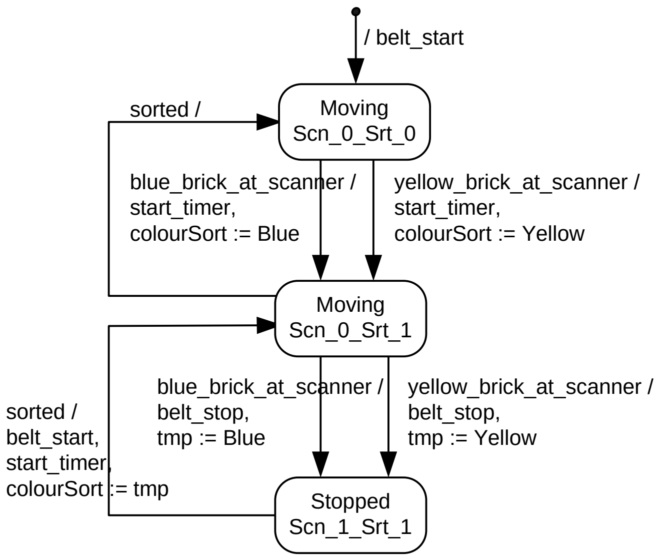


Figure 16: Belt control process

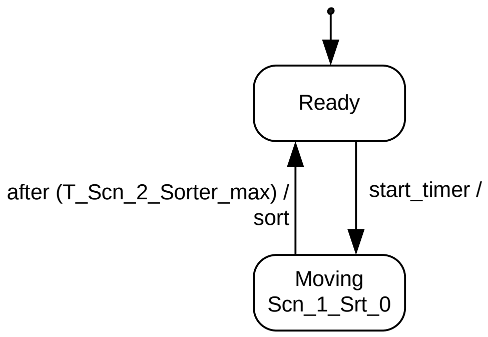
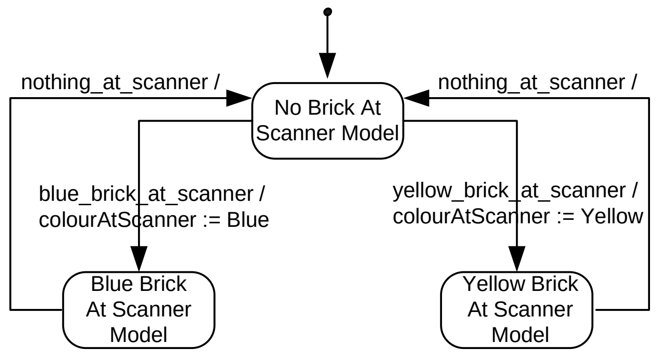


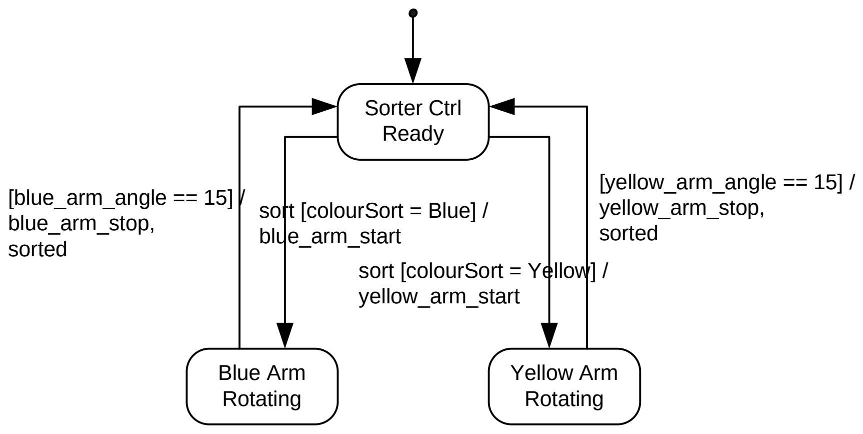
Figure 17: Delay process



nothing_at_scanner /
No Brick At Scanner Model
nothing_at_scanner /
blue_brick_at_scanner / colourAtScanner := Blue
yellow_brick_at_scanner / colourAtScanner := Yellow
Blue Brick At Scanner Model
Yellow Brick At Scanner Model

Figure 18: Scanner control process

The state diagram in Fig.19 shows the sorter control. It shows that the control starts rotating one of the sorter arms after an event sort occurs. We do not have a brick sensor in the sorter, but we previously measured the maximal time for a brick to arrive from the scanner to the sorter (for this the Delay process is responsible). On the sort action), the sorter starts rotating. After the arm has made a full rotation, it waits for another brick to arrive (another sort action).



Sorter Ctrl Ready
[blue_arm_angle == 15] / blue_arm_stop, sorted
sort [colourSort = Blue] / blue_arm_start
sort [colourSort = Yellow] / yellow_arm_start
[yellow_arm_angle == 15] / yellow_arm_stop, sorted
Blue Arm Rotating
Yellow Arm Rotating

Figure 19: Sorter control process

Assumptions made while describing the control are:

A_{11}: The sorter should start with the proper initial position, so that its arms do not block the brick arriving from the belt.

A_{12}: There is a minimal distance between bricks so that

$A_{A12.1}$: There is always "nothing_at_scanner" observed by Scanner before the new brick is observed and

$A_{A12.2}$: There will be no new brick in front of the Scanner before a previous brick moves to the Sorter.

A_{13}: The order of events of the leading edge of a brick observed and a new brick entirely on the belt is not deterministic. It can happen that a brick arrives in front of the scanner before previous brick is sorted

A_{14}: The plant is started with a brick laying neither on the Belt nor in the Sorter.

These assumptions are related to the design decisions for the control. The assumption A_{11} is made because with the rotation sensor and sorter as they are given to us, it is possible only to measure the angle change relative to the initial arm position. (It is not possible to measure whether the sorter arms are in the horizontal position.) Figure 20 shows irregular starting position of the sorter, which prevents a brick from entering.

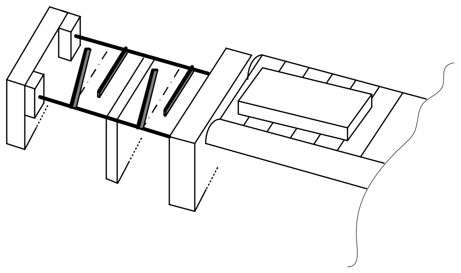


Figure 20: Irregular position of a sorter arm that prevents a brick from entering

The assumptions $A_{12.1}$ and $A_{12.2}$ can be dropped if we have decided to design a control in a different way. For example, we could model the speed of the belt and time needed for a brick to pass along the scanner, which would enable distinguishing bricks attached to each other even if they are of a same colour.

The assumption A_{13} results from observing the plant.

4.7 Step (7): Formal Description of the System: Timed Automata Description

When we are deriving the Uppaal model from the statecharts, there are rules applied straightforward and there are modelling decisions related to the tool itself. We translated states of the statecharts into Uppaal locations and events to Uppaal channels. But, there were additional things that we have to decide for ourselves how to describe in Uppaal. For example, in Uppaal time is described explicitly and, although we are not interested in time that a brick needs to come from the queue to the sorter, we have to state it explicitly there.

Environment

Figures 21 and 22 show the Uppaal implementation of the Queue statechart. The auxilliary automaton is one of the 'tricks' we need when modelling in Uppaal in order to enable transition (channel) x.

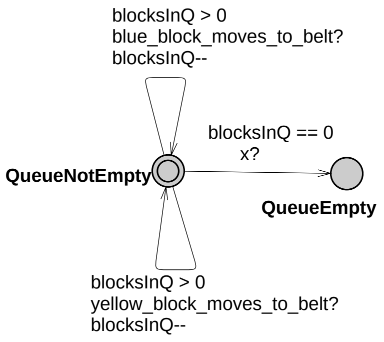


Figure 21: Uppaal model of the Queue

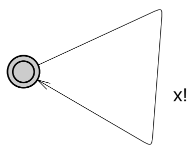


Figure 22: Auxiliary automaton

Figures 23, 24, 25 and 26 represent the timed-automata descriptions of the belt and its all possible responses to events on its interfaces.

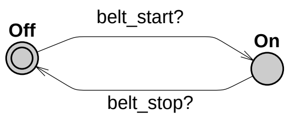


Figure 23: Uppaal model of the belt moving or being stopped

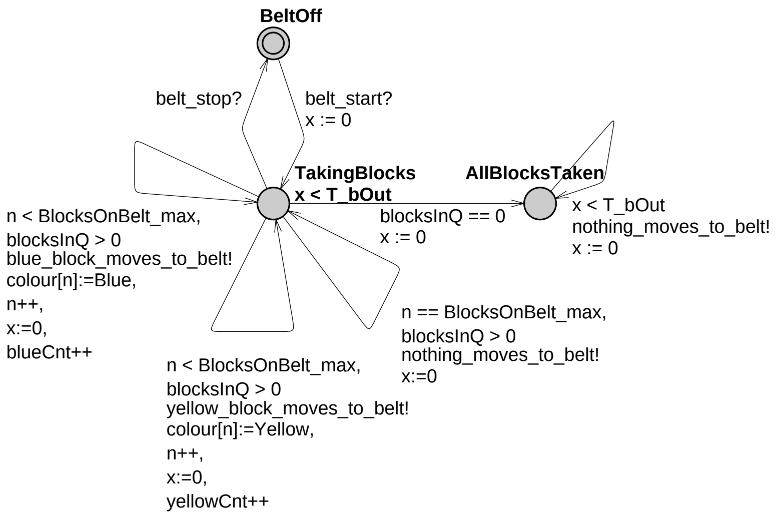


Figure 24: Uppaal model of the belt taking bricks from the queue

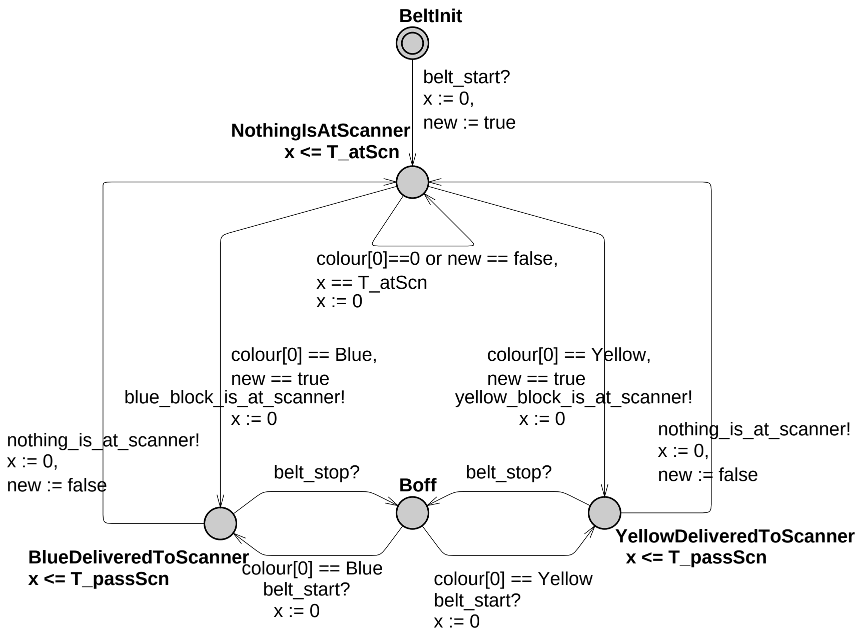


Figure 25: Uppaal model of the belt transporting brick to the scanner

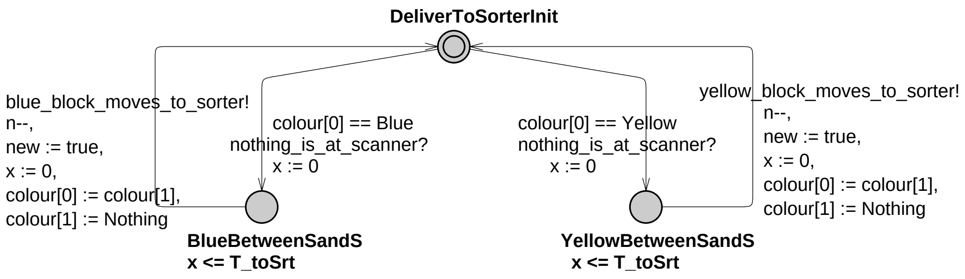


Figure 26: Uppaal model of the belt transporting brick to the sorter

The Uppaal model of the Scanner (figure 27) has a committed location because we assumed that event of scanner signalling a brick arrival to the Machine happens in infinitely short time after actual brick arriving at the Scanner.

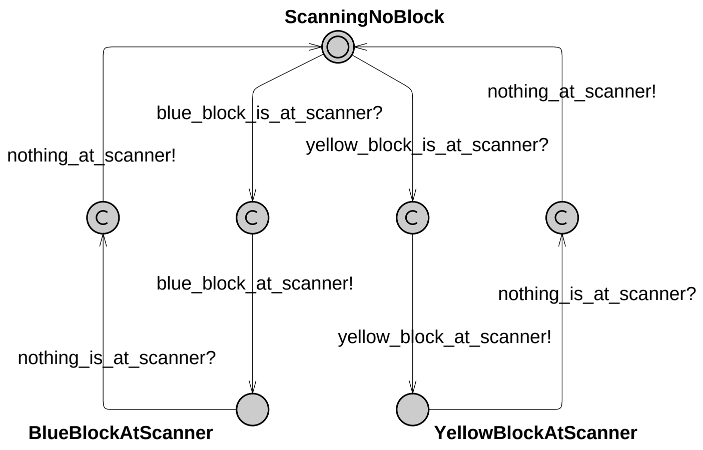


Figure 27: Uppaal model of the scanner

On the figure 28 the Uppaal model of the Sorter is shown.

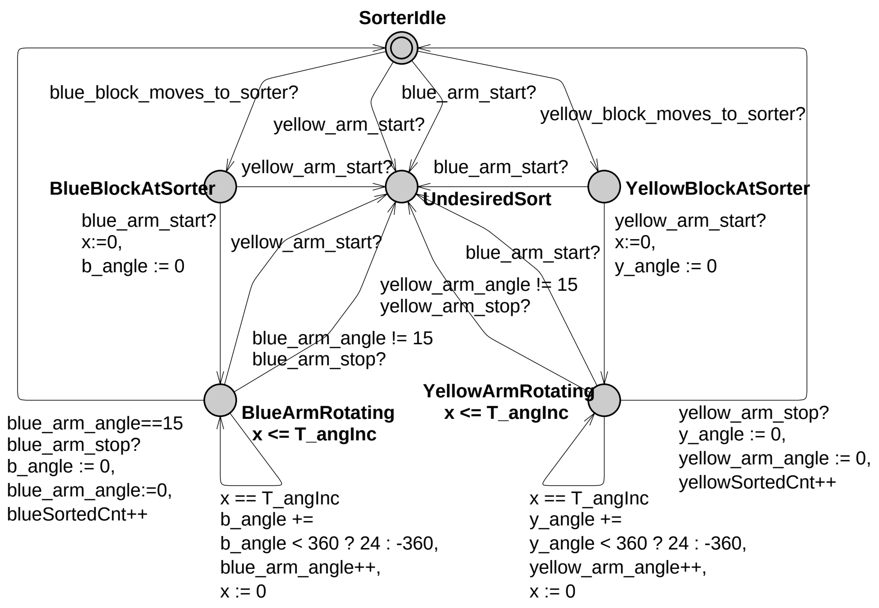


Figure 28: Uppaal model of the sorter

Control

The model of the process controlling the belt (figure 29) has the same structure as the statechart, but there are again committed locations that describe actions (channels) that occur at the same time (that are atomic).

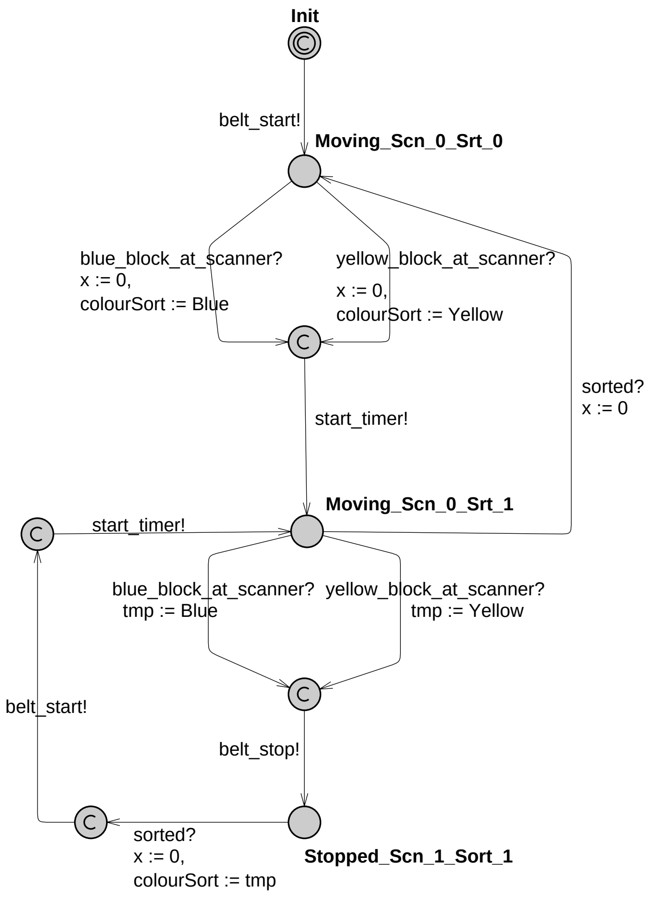


Figure 29: Uppaal model of the Belt control process

The process that triggers the sorter to start sorting is the Delay process, shown on the figure 30.

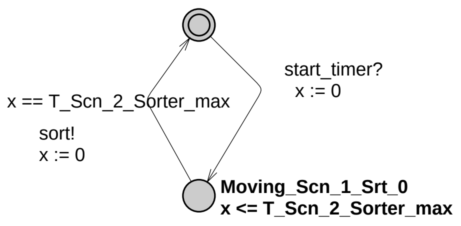


Figure 30: Uppaal model of the Delay process

It has variables that keep the information about the colour observed; these are so called model variables - they represent the Scanner in the control.

Finally, figures 31 and 32 show the Uppaal model of the scanner and sorter control processes.

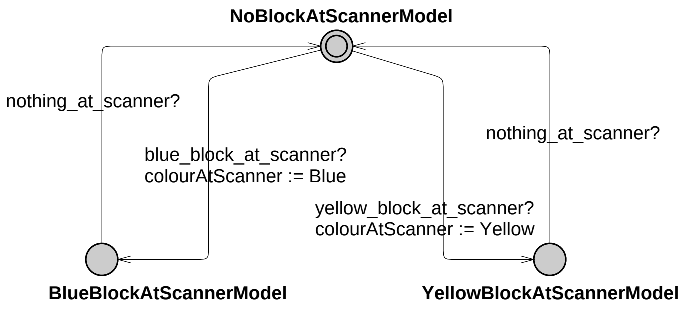


Figure 31: Uppaal model of Scanner control process

When the timer measuring the time needed for a brick to arrive from the scanner to the sorter expires, the controller starts one of the sorter arms. Variable colourSort represents the brick in the sorter with the information of its colour. Upon turning the arm on, its angle is sensed and, when the arm makes a full rotation, the arm is stopped. After the arm is stopped, the sorter is ready to accept a new brick.

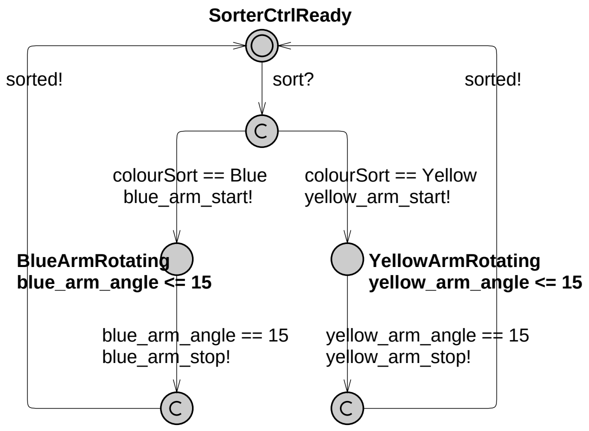


Figure 32: Uppaal model of Sorter control process

4.8 Step (9): Model Verification

Part of the verification is model testing. Is the model an adequate representation of the real system? There are some system properties that we represented by using time in Uppaal and we check whether we have done this correctly by

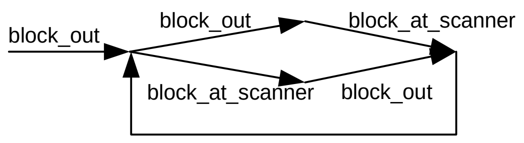


Figure 33: Two possible scenarios describing brick progression from the queue to the scanner

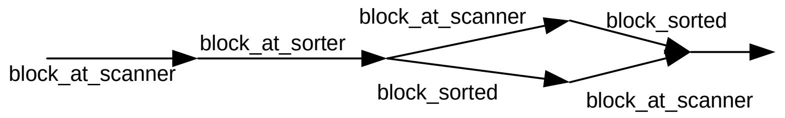


Figure 34: Two possible scenarios describing brick progression from the Scanner to the Sorter

model checking. For example, modelling progression of a brick was not easy thing to do in Uppaal. As we mentioned before, our requirement is functional, not a timing property, so we did not put accurate times between events of brick getting out of the queue, passing the scanner and moving into the sorter. However, we wanted to include all possible scenarios and exclude those that are not possible.

We tuned times in different automata in order to show the scenarios shown in figures 33 and 34. By model checking we checked whether the scenarios are indeed shown in our model.

The requirement R_1 is translated into the property of bricks being sorted in appropriate sides. Since we have automata-based model, we are checking if the system model will reach the state where all bricks are sorted.

For all paths it will eventually hold the following:
(1) The number of the blue (yellow) bricks that were moved out from the Queue is the same as the number of the blue (yellow) bricks on the side of the sorter that sorts blue (yellow) bricks and
(2) The sum of the sorted blue and yellow bricks is the same as the initial number of the bricks put in the Queue.
The second statement prevents the query to be true at the beginning when counters are zero.

In the Uppaal we write this in linear temporal logic:

A<> BeltFromQueue.blueCnt == Sorter.blueSortedCnt and
BeltFromQueue.yellowCnt == Sorter.yellowSortedCnt and
BeltFromQueue.blueCnt + BeltFromQueue.yellowCnt == NrBricks

Uppaal verified both statements to be true of the model. If we look back at the verification requirement that we started from, $A \implies (P \wedge C \implies R)$, we see that with Uppaal tool we proved that $(P \wedge C \implies R)$. This is how far we can go with the formal proof. The assumptions are the informal part that comes together with the formal part of our proof.

5 Related Work

The need to combine, intertwine, interleave or bring closer informal and formal methods has been already stressed by several authors. Some of these are general discussions, like the panel discussion about integration of the methods, where participants concluded that the main object of integration is "...to make formal methods easier to apply and to make informal methods more rigorous." [3]. Note that this is not our goal; we want to increase confidence in the applicability of a formal proof to a particular system by making explicit the assumptions made by the formal proof about the system.

The value of informal language as part of the process of formal specification was stressed by Le Chariler and Flener [4] in one of a series of papers debating how formal methods should be used in practice and what are the myths surrounding them. Meyer showed actually the reverse, namely how formal specification leads to a better natural language description [13] by revisiting well known example of text-processing problem.

There are also papers exploring the combination of different formal and informal techniques. Yeung and Leung [17] argue for intertwining formal and informal methods in the initial software specification stage, and use JSD structure diagrams and CSP language for this purpose. Fraser et. al. [6] described combining informal, structured analysis (SA) and formal Vienna Development Method (VDM) techniques for elicitation and assurance of requirements. Again, our purpose is not to combine formal and informal methods, but to increase confidence in a formal proof by identifying assumptions.

There is another group of authors working on formalization of the UML. Among them, Choppy and Reggio combined this approach with the problem frames technique [5].

An important difference between all these approaches and our work is that we would like to stay independent of the tools and techniques and that our focus is slightly different. We are not formalizing a specific informal technique to a particular formal language. Strictly speaking, formalizing an informal technique results in a formal technique; it does *not* result in the combination of formal and informal technique. We are focused on all informal aspects of the formal verification that should be brought from implicit presence to explicit, documented libraries and documents. Also, we believe that although modelling is a creative process, there are steps that can be described in guidelines specialized for particular classes of problems. These guidelines should capture best practices in the design of a verification model for a certain class of problems. Another difference is that our work is focused on both software and its environment and we, in the end, prove behaviour of the plant, not of the software.

6 Discussion and Conclusion

We presented the steps we followed while modelling an embedded system. In the different stages we captured different assumptions. During the first stage, we considered such things as the operating system and hardware of the computer on which the control is implemented, processor speed, deviations from the sampling times. As we progressed with describing the plant behaviour, we have put constraints (made assumptions) to its parts and the way the plant will be used by other people. While we were describing the control, the initialization conditions were raised. These are the conditions that have to be satisfied before starting on the plant and before starting the controller. The fact that we describe the embedded system behaviour and requirements imposed to its parts, and not only the software was helpful to identify the assumptions about the environment.

Assumptions. Why is it useful to capture the assumptions? If we look at the system as a box that we are delivering with confidence that it behaves as required, then the list of assumptions is a label that should be put on this box, e.g. "This lift will work as described if the total weight of passengers is less than 300 kilos." Although written informally, these assumptions come together with the formal verification proof and they increase our confidence in the applicability of this proof on a particular system.

However, we can never be sure that we captured all important assumptions. The more experience we have with similar systems, the more we can be sure that we did not forget essential assumptions. If we could have a library of subproblems and corresponding modelling patterns, we should also provide a library of assumptions that are usually made in these subproblems and faults that occur if we make wrong assumptions. For example, the model of our Lego sorter could possibly be (re)used for systems with conveyor belts.

For some of the assumptions we (subjectively) decided that they are too obvious to be put in the list with the assumptions. For example, we did not put the possibility of turning on the motors in the wrong direction. Our controller cannot observe this and cannot do anything about it when it occurs.

"Why not formalize these assumptions and add them to the model?", one may ask. Formalization would be too difficult, maybe even impossible. An example is the difficulty to formally describe the assumption of the plant being placed on a flat horizontal surface. Other assumptions may be formally described more easily, but in our experience this results in the model that is too big for model checking and too complex to be understood by other people. (And we want the model to be understandable for other colleagues and used as means of communication with them, and small enough to be handled by model checking tools.)

What happens if we want to drop out one of the assumptions? We can distinguish two kinds of assumptions related to this question. First are the assumptions that, if they were dropped and if we still wanted to have a correct system, we would have to design more sophisticated control. For example, if we want to have a plant that can start with a brick already present in front of the scanner, we could change our control in a way that it deals with this. But there are the assumptions that, if dropped, cannot be compensated for by making the control more complex. For example, putting sorter arms in the initial position is necessary, because the rotation sensor that we have can sense only a relative change of the angle. It cannot observe absolute angles.

If we had an experienced modeller, would he write down all these assumptions if he started making the Uppaal model immediately, without making problem diagrams and statechart models first? This is an empirical question, but a priori something can be said: a formal verification expert may not be a domain expert and she needs a means of communication with a domain expert. Each of the steps has a different level of abstraction of the system and problem diagrams and statecharts can serve as a means of communication with other people that are not model checking experts.

Problem Frames. The problem frames technique is useful in a phase of identifying relevant parts of the plant and relating them to both the requirements and the software. Problem diagrams are similar to context diagrams, but one important difference is that they contain requirements as well. One of the critiques addressed to the problem frames technique however is that it does not give an answer how to recompose the solutions for the subproblems identified in the system decomposition. We plan to look at this in future work.

Statecharts. We used statecharts as a tool that could be understood by a software engineer, who does not necessary have to be a formal verification expert. They can be used for a communication between these two roles.

Also, we had the idea of having statecharts as a common system model from which we can continue using different model checking languages and tools. But we found that there is no such thing as a single statechart model that abstracts from all behaviour modelling model checkers. Certain properties are modelled in one way in one model checker and in another way in the other model checker, and they have no shared statecharts abstraction. Also, statecharts semantics almost certainly differs from a model checker semantics.

So, the price that we have to pay for having them for the communication between software engineer and formal verification expert is that this puts us in danger of having to make additional diagrams that may not represent correctly the more detailed model checking automata.

Uppaal Uppaal is a useful and powerful tool for proving timing properties. In our example we needed time explicitly to describe block progress from the scanner to the sorter. Moreover, although our requirement is functional, not timing, we could imagine that it would be necessary to prove also that sorting will be finished in reasonable time. In our future work, we plan to use other model checking tools and compare them.

Bibliography

[1] MOCA project - ongoing work. *http://moca.ewi.utwente.nl/WORK.html/.*

[2] UPPAAL home page. *http://www.uppaal.com.*

[3] J-M. Bruel. Integrating informal and formal specification techniques. why? how? (panel session). In *Second IEEE Workshop on Industrial Strength Formal Specification Techniques, WIFT'98, Boca Raton/FL, USA*. IEEE CS Press, 1998.

[4] B. Le Charlier and P. Flener. Specifications are necessarily informal or: some more myths of formal methods. *J. Syst. Softw.*, 40(3):275–296, 1998.

[5] C. Choppy and G. Reggio. Using uml for problem frame oriented software development, 2004.

[6] M. D. Fraser, K. Kumar, and V. K. Vaishnavi. Informal and formal requirements specification languages: Bridging the gap. *IEEE Trans. Softw. Eng.*, 17(5):454–466, 1991.

[7] C.A. Gunter, E.L. Gunter, M.A. Jackson, and P. Zave. A reference model for requirements and specifications. *IEEE Software*, 17(3):37–43, May/June 2000.

[8] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[9] H.Wupper. Design as the discovery of a mathematical theorem - what designers hould know about the art of mathematics. In *Third Biennial World Conference on Integrated Design and Process Technology (IDPT)*, 1998.

[10] M.A. Jackson. *Software Requirements and Specifications: A lexicon of practice, principles and prejudices.* Addison-Wesley, 1995.

[11] M.A. Jackson. *Problem Frames: Analysing and Structuring Software Development Problems.* Addison-Wesley, 2000.

[12] A. Mader, E. Brinksma, H. Wupper, and N. Bauer. Design of a PLC control program for a batch plant - VHS case study 1. *European Journal of Control*, 7(4):416–439, 2001.

[13] B. Meyer. On formalism in specifications. *IEEE Software*, pages 6–26, January 1985.

[14] M. von der Beeck. A comparison of statecharts variants. In *ProCoS: Proceedings of the Third International Symposium Organized Jointly with the Working Group Provably Correct Systems on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 128–148, London, UK, 1994. Springer-Verlag.

[15] H. Wupper. Taxonomy of computer science and Non-monotonic refinement - Wiki page. *http://www.cs.ru.nl/ wupper/wiki/index.php/Decomposition.*

[16] H. Wupper and A. Mader. System design as a creative mathematical activity. Technical report CSI-R9919, Univ. of Nijmegen, 1999. `http://www.cs.kun.nl/research/reports/full/CSI-R9919.ps.Z`.

[17] W. L. Yeung and Karl R.P.H. Leung. A synergistic interweaving of formal and informal methods. *compsac*, 00:257–?, 2003.